The Dissertation Committee for Maximilian Heimo Moritz Bremer certifies that this is the approved version of the following dissertation:

Task-Based Parallelism for Hurricane Storm Surge Modeling

Committee:

Clint Dawson, Supervisor

George Biros

Irene Gamba

Patrick Heimbach

Keshav Pingali

Task-Based Parallelism for Hurricane Storm Surge Modeling

by

Maximilian Heimo Moritz Bremer

Dissertation

Presented to the Faculty of the Graduate School
of the University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

The University of Texas at Austin
August 2020

In dedication to my family—old and new—for all of their love and support.

Acknowledgments

First and foremost I would like to thank Prof. Clint Dawson, my advisor, for all of his support over the many years. None of this would have been possible without his patience, encouragement, and guidance.

Secondly, I would like to gratefully acknowledge the funding that allowed me to pursue this research. Thank you to the Department of Energy's Computational Science Graduate Fellowship (Grant DE-FG02-97ER25308) not only for their financial support, but also all the friends I have made along the way. Additionally, I would like to thank the University of Texas at Austin's Donald D. Harrington Graduate Fellowship.

Finally, I would like to thank the many mentors that have made this journey wonderful: Craig Michoski, Cy Chan, John Bachan, Hartmut Kaiser, and many more unnamed friends made along the way.

Task-Based Parallelism for Hurricane Storm Surge Modeling

by

Maximilian Heimo Moritz Bremer, Ph.D.
The University of Texas at Austin, 2020
SUPERVISOR: Clint Dawson

Hurricanes are incredibly devastating events, constituting seven of the ten most costly U.S. natural disasters since 1980. The development of real-time forecasting models that accurately capture a storm's dynamics play an essential role in informing local officials' emergency management decisions. ADCIRC is one such model that is operationally active in the National Oceanic and Atmospheric Administration's Hurricane Surge On-Demand Forecast System. However, ADCIRC faces several limitations. It struggles solving highly advective flows and is not locally mass conservative. These aspects limit applicable flow regimes and can cause unphysical behavior. One proposed alternative which addresses these limitations is the discontinuous Galerkin (DG) finite element method. However, the DG method's high computational cost makes it unsuitable for real-time forecasting and has limited adoption among coastal engineers. Simultaneously, efforts to build an exascale machine and the resulting power constrained computing architectures have led to massive increases in the concurrency applications are expected to manage. These architectural shifts have in turn caused some groups to turn away from the traditional flat MPI or MPI+OpenMP programming models to more functional task-based programming models, designed specifically to be performant on these next generation architectures. The aim of this thesis is to utilize these new task-based programming models to accelerate DG simulations for coastal applications.

We explore two strategies for accelerating the DG method for storm surge simulation.

The first strategy addresses load imbalance caused by coastal flooding. During the simulation of hurricane storm surge, cells are classified as either wet or dry. Dry cells can trivially update, while wet cells require full evaluation of the physics. As the storm makes landfall and causes flooding, this generates a load imbalance. We present two load balancing strategies—an asynchronous diffusion-based approach and semi-static approach—to optimize compute resource utilization. These load balancing strategies are analyzed using a discrete-event simulation that models the task-based storm surge simulation. We find speed-ups of up to 56% over the currently used mesh partitioning and up to 97% of the theoretical speed-up.

The second strategy focuses on a first order adaptive local timestepping scheme for nonlinear conservation laws. For problems such as hurricane storm surge, the global CFL timestepping constraint is overly stringent for the majority of cells. We present a timestepping scheme that allows cells to stably advance based on local stability constraints. Since allowable timestep sizes depend on the state of the solution, care must be taken not to incur causality errors. The algorithm is accompanied with a proof of formal correctness that ensures that with a sufficiently small minimum timestep, the solution exhibits desired characteristics such as a maximum principle and total variation stability. The algorithm is parallelized using a speculative discrete event simulator. Performance results show that the implementation recovers 59%-77% of the optimal speed-up.

Contents

Chapter 1

Introduction

Since 1980, seven out of the ten most costly US climate disasters were hurricanes, with Hurricane Katrina being the most expensive [97], and Hurricanes Harvey, Maria, and Irma, which occurred in 2017, among the five most costly US natural disasters. The utilization of computational models can provide local officials with high fidelity tools to assist in evacuation efforts and mitigate loss of life and property. Due to the highly nonlinear nature of hurricane dynamics and stringent time constraints, high performance computing (HPC) plays a cornerstone role in providing accurate predictions of flooding. Because of the importance of fast, efficient models, there is a significant interest in improving the speed and quality of these computational tools.

Even as the speed of supercomputers is drastically increasing, the end of Moore's law and the introduction of many-core architectures represent a tectonic shift in the HPC community. In particular, the degree of hardware parallelism is increasing at an exponential rate and the cost of data movement and synchronization is increasing faster than the cost of computation. Additionally, hardware is becoming increasingly irregular due to the use of accelerators and susceptibility to failure [73]. In order to achieve good resource utilization on these machines, task-based programming and execution models are being developed to express increased software parallelism, introduce more flexible load balancing capabilities, and hide the cost of communication through task over-decomposition. Examples of major

task-based programming and execution models include Charm++, HPX [68], Legion [6], OCR [86], PaRSEC [14], StarPU [2], Galois [79, 106]. There are also domain-specific task-based programming systems, such as the Uintah AMR Framework [8], and task-based portability layers such as DARMA [127]. These programming models decouple the specification of the algorithm from the task scheduling mechanism, which determines where and when each task may execute and orchestrates the movement of required data between the tasks. Furthermore, lightweight, one-sided messaging protocols that support active messages have the potential to reduce the overheads associated with inter-process communication and synchronization, which will become even more important as parallelism increases.

The aim of this thesis is to identify means to improve the performance of the simulation of hurricane storm surge by using task-based runtimes. We accomplish this by identifying and introducing adaptivity into the simulation. These improvements historically suffer from complex implementations that are difficult to optimize using traditional programming models. Through proper mapping of the algorithms onto a task-based runtime, these performance gains are realized and the complexity of the implementation is managed by the runtime system.

1.1 Storm Surge Modeling

The motivating application for this work is the numerical simulation of large-scale coastal ocean physics, in particular, the modeling of hurricane storm surges. One of the leading simulation codes in this area is the Advanced Circulation (ADCIRC) model, developed by a large collaborative team including the author's supervisor [20, 36–38, 60, 61, 83, 126]. ADCIRC is a Galerkin finite element based model that uses continuous, piecewise linear basis functions defined on unstructured triangular meshes. The model has been parallelized using MPI and has been shown to scale well to a few thousand processors for large-scale prob-

lems [119]. While ADCIRC is now an operational model within the National Oceanographic and Atmospheric Administration's Hurricane Surge On-Demand Forecast System (HSOFS), its performance on future computational architectures is dependent on potentially restructuring the algorithms and software used within the model. Furthermore, ADCIRC provides a low-order approximation and does not have special stabilization for advection-dominated flows, thus requiring a substantial amount of mesh resolution. Extending it to higher-order or substantially modifying the algorithms within the current structure of the code is a challenging task.

With this in mind, our group has also been investigating the use of discontinuous Galerkin (DG) methods for the shallow water equations [21, 75, 76, 78, 90–93, 129, 130], focusing on the Runge-Kutta DG method as described in [28]. We have shown that this model can also be applied to hurricane storm surge [32]. DG methods have potential advantages over the standard continuous Galerkin methods used in ADCIRC, including local mass conservation, ability to dynamically adapt the solution in both spatial resolution and polynomial order (*hp*-adaptivity), and potential for more efficient distributed memory parallelism [77]. While DG methods for the shallow water equations have not yet achieved widespread operational use, recent results have shown that for solving a physically realistic tidal forecasts at comparable accuracies, the DG model outperformed ADCIRC in terms of efficiency by a speed-up of 2.3 and when omitting eddy viscosity, a speed-up of 3.9 [19]. We note that translating these results to hurricane storm surge remains an open problem. There remain outstanding stability issues for wetting and drying of high order DG methods. Furthermore, physically relevant problems may form discontinuities. The loss of smoothness will also require more computational work to match the accuracy of ADCIRC. Ultimately, we expect some accuracy improvements of high order methods compared to low order DG methods, however, given the stability issues, we do not consider high order methods in this thesis, but rather seek performance improvements from avenues that are "orthogonal" in the sense that we hope to

at a later point incorporate both high-order methods as well as the performance optimizations explored in this thesis.

To propose replacing the computational kernel inside ADCIRC with a DG kernel, it is important to understand the computational environment and needs of the simulation. During a hurricane event, the National Weather Service releases advisories every 6 hours updated with the newest best estimates of windspeeds and storm trajectories. Emergency management officials require a 2 hour turnaround of the simulation with the new data as inputs. This turns out to be a relatively stringent time constraint, and requires significant strong scaling of the simulation. Extrapolating the HPX-based results from [15], a 4 day forecast using a DG method would require 17.5 hours at the strong scaling limit. The required minimum task sizes for good runtime efficiency (i.e. fraction of time spent in the application versus the runtime) puts the HPX implementation well outside of the scalability regime needed for real-time forecasting of storm surge. From a work depth perspective, this is not an issue of available parallelism, but rather a depth issue. In considering task-based runtimes, we are highly sensitive to task-overheads as these ultimately determine how far we can scale out. However, the work required by the simulation is also an issue. Surge forecasts are run on shared compute resources, typically university clusters. While runs are given scheduling priority, we are certainly constrained by available compute resources. Furthermore, we would like to run ensembles of surge simulations. Hence, work reduction is a point of key emphasis. Lastly, a single surge simulation requires up to $\mathcal{O}(10^2)$ nodes. These node counts correspond to mesh resolutions of $\mathcal{O}(10)$ meters. We expect that higher resolution would only lead to marginal benefits, due to limitations of the model and uncertainties in the input.

The prediction of hurricane storm surge involves solving physics-based models that determine the effect of wind stresses pushing water onto land and the restorative effects of gravity and bottom friction. These flows typically occur in regimes where the shallow water approximation is valid [41, 92] . Taking the hydrostatic and Boussinesq approximations, the

governing equations can be written as

$$\partial_t \zeta + \nabla \cdot \mathbf{q} = 0,$$
$$\partial_t q_x + \nabla \cdot (\mathbf{u} q_x) + \partial_x g(\zeta^2/2 + \zeta b) = g\zeta \partial_x b + S_1,$$
$$\partial_t q_y + \nabla \cdot (\mathbf{u} q_y) + \partial_y g(\zeta^2/2 + \zeta b) = g\zeta \partial_y b + S_2,$$

where:

- ζ is the water surface height above the mean geoid,
- b is the bathymetry of the sea floor with the convention that downwards from the mean geoid is positive,
- $H = \zeta + b$ is the water column height,
- $\mathbf{u} = [u\,,\, v]^T$ is the depth-averaged velocity of the water,
- $\mathbf{q} = H\mathbf{u} = [q_x\,,\, q_y]^T$ is the velocity integrated over the water column height.

Additionally, g is the acceleration due to gravity, and S_1 and S_2 are source terms that introduce additional forcing associated with relevant physical phenomena, e.g. bottom friction, Coriolis forces, wind stresses, etc.

1.2 The Discontinuous Galerkin Finite Element Method

The discontinuous Galerkin (DG) kernel originally proposed by Reed and Hill [108] has achieved widespread popularity due to its stability and high-order convergence properties. For an overview on the method, we refer the reader to [29, 58] and references therein. For brevity, we forgo rigorous derivation of the algorithm, but rather aim to provide the salient features of the algorithm to facilitate discussion of the parallelization strategies.

We can rewrite the shallow water equations in conservation form

$$\partial_t \mathfrak{U} + \nabla \cdot \mathbf{F}(t, \mathbf{x}, \mathfrak{U}) = S(t, \mathbf{x}, \mathfrak{U}), \tag{1.1}$$

where

$$\mathfrak{U} = \begin{pmatrix} \zeta \\ q_x \\ q_y \end{pmatrix}, \quad \mathbf{F} = \begin{pmatrix} q_x & q_y \\ u^2 H + g(\zeta^2/2 + \zeta b) & uvH \\ uvH & v^2 H + g(\zeta^2/2 + \zeta b) \end{pmatrix}.$$

Let Ω be the domain over which we would like to solve Equation (1.1), and consider a mesh discretization $\Omega^h = \cup_e^{n_{el}} \Omega_e^h$ of the domain Ω, where n_{el} denotes the number of elements in the mesh.

We define the discretized solution space, $\mathcal{W}^h$ as the set of functions such that for each state variable the restriction to any element Ω_e^h is a polynomial of degree p. Note that we enforce no continuity between element boundaries. Let $\langle f, g \rangle_\Gamma = \int_\Gamma fg \, \mathrm{d}x$ denote the L^2 inner product over a set Γ. The discontinuous Galerkin formulation then approximates the solution by projecting $\mathfrak{U}$ onto $\mathcal{W}^h$ and enforcing Equation (1.1) in the weak sense over $\mathcal{W}^h$, i.e.

$$\langle \partial_t \mathbf{U} + \nabla \cdot F(t, \mathbf{x}, \mathbf{U}) - S(t, \mathbf{x}, \mathbf{U}), \mathbf{w} \rangle_{\Omega^h} = 0$$

for all $\mathbf{w} \in \mathcal{W}^h$, where $\mathbf{U} \in \mathcal{W}^h$ denotes the projected solution. Since the indicator functions over each element are members of $\mathcal{W}^h$, it suffices to satisfy the weak formulation element-wise. The discontinuous Galerkin method can be alternatively formulated as

$$\partial_t \langle \mathbf{U}, \mathbf{w} \rangle_{\Omega_e^h} = \langle \mathbf{F}, \nabla \mathbf{w} \rangle_{\Omega_e^h} - \langle \widehat{\mathbf{F} \cdot \mathbf{n}}, \mathbf{w} \rangle_{\partial \Omega_e^h} + \langle \mathbf{S}, \mathbf{w} \rangle_{\Omega_e^h} \tag{1.2}$$

for all $\mathbf{w} \in \bigoplus_{d=1}^3 \mathcal{P}^p(\Omega_e^h)$ and for all $e = 1, \ldots, n_{el}$, where $\mathcal{P}^p(\Omega_e^h)$ is the space of polynomials of degree p over Ω_e^h. Due to the discontinuities between elements in both trial and test spaces, particular attention must be given to the boundary integral term, which is not well-defined even in a distributional sense. For evaluation, the boundary integral's integrand is replaced with a numerical flux $\widehat{\mathbf{F} \cdot \mathbf{n}}(\mathbf{U}^{int}, \mathbf{U}^{ext}) \mathbf{w}^{int}$. To parse this term, let $\mathbf{U}^{int}$ and $\mathbf{w}^{int}$ denote the value of $\mathbf{U}$ and $\mathbf{w}$ at the boundary taking the limit from the interior of Ω_e^h, and

let $\mathbf{U}^{ext}$ denote the value of $\mathbf{U}$ at the boundary by taking the limit from the interior of the neighboring element. For elements along the boundary of the mesh, the boundary conditions are enforced by setting $\mathbf{U}^{ext}$ to the prescribed values. For the numerical flux, $\widehat{\mathbf{F}\cdot\mathbf{n}}$, we use the local Lax-Friedrichs flux,

$$\widehat{\mathbf{F}\cdot\mathbf{n}}(\mathbf{U}^{int},\mathbf{U}^{ext}) = \frac{1}{2}\left(\mathbf{F}(\mathbf{U}^{int}) + \mathbf{F}(\mathbf{U}^{ext}) + |\Lambda|(\mathbf{U}^{ext}-\mathbf{U}^{int})\right)\cdot\mathbf{n},$$

where $\mathbf{n}$ is the unit normal pointing from Ω_e^h outward, and $|\Lambda|$ denotes the magnitude of the largest eigenvalue of $\nabla_u\mathbf{F}\cdot\mathbf{n}$ at $\mathbf{U}^{int}$ or $\mathbf{U}^{ext}$.

In order to convey more clearly the implementation of such a kernel in practice, consider the element Ω_e^h. For simplicity of notation for the remainder of the subsection we drop all element-related subscripts, e. Over this element, we can represent our solution using a basis, $\{\varphi_i\}_{i=1}^{n_{dof}}$. Then we can let our solution be represented as

$$\mathbf{U}(t,x) = \sum_{i=1}^{n_{dof}} \tilde{\mathbf{U}}_i(t)\varphi_i(x),$$

where $\tilde{\mathbf{U}}$ are the basis-dependent coefficients describing $\mathbf{U}$. Following the notation of Warburton [47], it is possible to break down Equation (1.2) into a set of kernels as

$$\partial_t\tilde{\mathbf{U}}_i = \sum_{j=1}^{n_{dof}} \mathcal{M}_{ij}^{-1}\left(\underbrace{\langle\mathbf{F},\nabla\varphi_j\rangle_{\Omega_e^h}}_{\mathcal{V}_j} + \underbrace{\langle\mathbf{S},\varphi_j\rangle_{\Omega_e^h}}_{\mathcal{S}_j} - \underbrace{\langle\widehat{\mathbf{F}\cdot\mathbf{n}},\varphi_j\rangle_{\partial\Omega_e^h}}_{\mathcal{I}_j}\right), \tag{1.3}$$

where $\mathcal{M}_{ij} = \langle\varphi_i,\varphi_j\rangle_{\Omega_e^h}$ denotes the local mass matrix. Here we define the following kernels:

- $\mathcal{V}$: The volume kernel,
- $\mathcal{S}$: The source kernel,
- $\mathcal{I}$: The interface kernel.

To discretize in time, we use the strong stability preserving Runge-Kutta methods [49].

Letting

$$\mathcal{L}^h\left(\tilde{\mathbf{U}}\right) = \mathcal{M}^{-1}\left(\mathcal{V}\left(\tilde{\mathbf{U}}\right) + \mathcal{S}\left(\tilde{\mathbf{U}}\right) - \mathcal{I}\left(\tilde{\mathbf{U}}\right)\right),$$

we can define the timestepping method, for computing the i-th stage as

$$\tilde{\mathbf{U}}^{(i)} = \sum_{k=0}^{i-1} \alpha_{ik}\tilde{\mathbf{U}}^{(k)} + \beta_{ik}\Delta t\mathcal{L}^h\left(\tilde{\mathbf{U}}^{(k)}\right),$$

where $\tilde{\mathbf{U}}^{(k)}$ denotes the basis coefficients at the k-th Runge-Kutta stage. We denote the operator, which maps $\left\{\tilde{\mathbf{U}}^{(k)}, \mathcal{V}\left(\tilde{\mathbf{U}}^{(k)}\right), \mathcal{S}\left(\tilde{\mathbf{U}}^{(k)}\right), \mathcal{I}\left(\tilde{\mathbf{U}}^{(k)}\right)\right\}_{k=0}^{i-1}$ to $\tilde{\mathbf{U}}^{(i)}$ as the update kernel, $\mathcal{U}$.

Solutions to hyperbolic conservation laws may give rise to discontinuous solutions. In these regions, the underlying approximation theory breaks down, and the DG algorithm gives rise to spurious oscillations. These oscillations are treated in a post processing phase known as *slope-limiting*. Techniques roughly are categorized as limiting based on neighboring information [7, 27, 80], reducing high frequency modes via filtering [58, 84, 87], adding viscosity to smear out sharp gradients [51, 105], or projection to lower orders [40]. For an in-depth comparison of various slope limiting approaches, we refer the reader to [91]. For the shallow water equations, an additional instability occurs for small water column heights. The numerics may give rise to regions of negative water column height causing the shallow water equations to become meaningless both physically and mathematically. To address this, numerous approaches have been proposed [21, 24, 47, 107, 124, 131].

1.3 Exascale Computing–Novel Programming Models

Task-based programming models have arisen in response to increased hardware concurrency and irregularity. To identify what makes multithreaded code slow, the Ste||ar group—who develop HPX—use the acronym SLOW:

- **S**tarvation: cores idling due to insufficient parallelism exposed by the application,
- **L**atency: delays induced by waiting on dependencies, e.g. waiting on messages which are sent through a cluster's interconnect,
- **O**verhead: additional work performed for a multithreaded application which is unnecessary in a sequential implementation,
- **W**aiting for contention resolution: delays associated with the accessing of shared resources between threads.

These factors are prevalent in any multi-threaded code including those that use a task-based runtime. However task-based runtimes aim to outperform their more traditional counterparts by mitigating the impacts of SLOW, e.g. by hiding message latencies or aggressive work stealing to address starvation.

Even as there appears to be no convergence to a single task-based runtime in the HPC community, there appears to be broad agreement on the abstractions being used. Most runtimes favor some task-graph or dataflow based abstraction for algorithm design. Once the task graph has been specified it is given to the runtime to be executed. Another commonly used abstraction is that of globally addressable objects. By making actors or objects globally addressable, the application is able to specify dependencies between objects with similar syntax for both shared and distributed memory parallelizations.

While simply porting algorithms will certainly allow us to mitigate the impact of hardware irregularity, we have found that in practice this gives small to moderate improvement over existing optimized MPI implementations. In [15], we found that HPX—one such runtime—gives a speed-up of 1.2 over an MPI implementation on 256 Knights Landing nodes. Results suggest that the speed-up was mainly attributed to page faults for the MPI implementation, rather than avoiding message latencies or MPI overhead. In fact, we suspect that this gap could largely be closed through the use of a better memory allocator for the MPI implementation.

This limited improvement can be attributed to two main factors. Firstly, Moore's law has not yet ended, and we have not yet truly entered the age of extreme heterogeneity where managing the SLOW factors will become increasingly important. Furthermore and specifically related to storm surge, given the relatively modest computational resource requirement of a real-time forecast, the dependence on legacy codes, and the low arithmetic intensity of the storm surge codes (both ADCIRC and low-order DG solvers), there still remains a hesitation to adopt non-CPU based architectures that may provide more FLOPs but whose impact on time to solution is unclear.

Secondly, the test case used in [15] ultimately lacks irregularity. The communication profile of the code reduces to a stencil code on an unstructured mesh, which can be implemented efficiently using non-blocking point to point MPI messages. What is needed to generate a value proposal for using task-based runtime systems are sources of irregularity. Algorithms need to be more adaptive and perform compute only where needed. While doing so makes the implementation certainly more difficult, this complexity is passed off to the runtime.

Then, the aim of this work is to identify or introduce sources of adaptivity into the storm surge code, which are are then efficiently mapped onto a runtime. This thesis is split into two main chapters describing the impact of two such ideas:

1. *Chapter 2*: A key feature of storm surge simulation is the classification of cells as either wet or dry. The relevant computational impact of this classification is that while wet cells require the evaluation of the full physics to update, dry cells trivially update. As the storm makes landfall, inland cells wet and create a load imbalance. This chapter explores the impact of dynamic load balancing strategies, whereby we move cells across processor ranks during the simulation to ensure that each rank must update an equal number of cells.

2. *Chapter 3*: The timestep taken by these simulations is restricted by the Courant-

Friedrichs-Lewy (CFL) condition. The CFL condition forms the basis of the stability results which have led to the popularity of finite volume and DG methods. However, this condition is ultimately a condition that can be enforced locally, and in light of significant variations in mesh size and advection speeds, local CFL enforcement can lead to dramatic reduction in work. This chapter develops a timestepping method that locally enforces the CFL condition and recovers the same stability results for the global timestepping case. The proposed timestepping scheme is then parallelized using an optimistic (speculative) parallel discrete event simulator.

Chapter 2

Semi-Static and Dynamic Load Balancing for Asynchronous Hurricane Storm Surge Simulations[1]

This chapter examines the potential benefits of implementing DGSWEM, a discontinuous Galerkin (DG) finite-element storm surge code, on a task-based asynchronous execution model, and investigates various load balancing strategies for the resulting program. One of the key aspects of DGSWEM is its ability to simulate coastal inundation during hurricane landfall. The DG kernel can be implemented in a manner such that dry regions of the simulation require no computational work; however, as the hurricane inundates the coast, this optimization introduces significant dynamic load imbalance. In order to address the load imbalance while still fitting the problem in machine memory, two constraints (load and memory) must be accounted for simultaneously.

While multi-constraint graph partitioning tools have been used to obtain good static partitions for these scenarios, they perform sub-optimally for irregular applications such as ours. On the other hand, fully dynamic load balancing strategies are typically based on balancing a single constraint, which is insufficient for hurricane simulation. In order to overcome these shortcomings, we investigate dynamic and semi-static multi-constraint load

[1]This chapter is based on the following publication: **Maximilian Bremer**, John Bachan, and Cy Chan. "Semi-Static and Dynamic Load Balancing for Asynchronous Hurricane Storm Surge Simulations." In *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, pages 44-56, 2018. Maximilian Bremer contributed 60% of the work, John Bachan contributed 20% of the work, and Cy Chan contributed 20% of the work in this chapter.

balancing approaches that can be leveraged to run hurricane simulations efficiently on future supercomputing platforms.

To circumvent the software engineering effort to port DGSWEM to a task-based model and to allow extremely rapid evaluation of the resulting execution under a variety of parameters (e.g. choice of load balancing algorithm), we have developed DGSim, a task-based execution model and discrete-event simulation tool that features asynchronously migratable, globally addressable objects. DGSim was designed to natively model lightweight, one-sided, active interprocess messaging along with distributed, asynchronously migratable objects. Together, these allow the simulation to fully overlap both the computation of the new tile placements and the resulting movement of the tile data with the application's main computation. These features are not readily available in other simulation tools but are key features of the fully asynchronous load balancing algorithms introduced here. The development of DGSim allows us to estimate the performance of hurricane simulations running with these advanced features in a variety of configurations. Crucially, DGSim allows us to simulate the computational behavior of DGSWEM in an asynchronous task-based runtime using skeletonization, eliminating the need to execute the costly computational kernels, resulting in a lightweight approach that allows for rapid algorithm prototyping and evaluation. The main contributions of this section are:

1. The development of multi-constraint dynamic load balancing strategies specifically geared for the irregularity associated with the simulation of hurricane storm surge. In doing so, we present a dynamic and semi-static algorithm.

2. The *trellis* approach for semi-static repartitiong strategies which fully leverages the asynchronous nature of the run-time. This approach can be extended to ensure efficient semi-static load balancing for a wide range of problems.

3. The development and validation of DGSim, a discrete-event simulator that enables

rapid prototyping and evaluation of *fully asynchronous* load balancers for task-based parallel programs.

The outline of the chapter is as follows. Section 2.1 presents related work. In Section 2.2, we discuss the DG kernel and the irregular nature of the inundation problem. Thereafter, we outline DGSim, which simulates the performance of an asynchronous task-based implementation of DGSWEM. Section 2.5 presents formalism for load balancing in an asynchronous context and outlines a distributed diffusion-based and a semi-static load balancing algorithm. Lastly, in Section 2.6, we present our model validation and experimental results, demonstrating the viability of these load balancing approaches.

2.1 Related Work

In adaptive mesh refinement (AMR) codes and *hp*-adaptive finite elements, dynamic load balancing typically relies on one of three approaches: (1) graph partitioning using space filling curves, known as geometric partitioning [11, 22, 35, 43] (2) graph partitioning algorithms, such as those provided in the METIS and SCOTCH libraries [3, 9, 35, 70, 104], or (3) diffusion or refinement based approaches [111]. The simulation of coastal inundation introduces irregularity, which decouples the memory and load balancing constraints. To the authors' knowledge, the only paper that uses dynamic load balancing to address this issue is [34]. However, their approach only balances load on structured grids, which may result in memory overflow. Local timestepping methods introduce similar irregularity. Seny et al. have proposed a static load balancing scheme using multi-constraint partitioning in [112]. However, they note the dynamic load balancing problem as an open one. Some examples of load balancing algorithm evaluations in the context of task-based execution models include the use of cellular automata [62], hierarchical partitioning [135], and gossip protocols [88].

There has been much previous work on the use of system-level hardware simulation to

evaluate how existing applications will behave on future architectures (e.g. [5,64,66,94,132]). For example, the SST-macro simulator [65] allows performance evaluation of existing MPI programs on future hardware by coupling skeletonized application processes with a network interconnect model. Previous work has investigated the impact of various static task placement strategies for AMR multigrid solvers using simulation techniques [26]. Evaluation of various load balancing strategies using discrete event simulation has been conducted in [133], and the use of particular asynchronous load balancing algorithms has been discussed in [103,134]. However, [134] examines only a simple greedy algorithm that ignores communication costs, and [103] examines an offload model that still requires synchronization to enter and exit the load balancing phase. Another framework, StarPU with SimGrid [115, 116], allows estimation of task-based execution on a parameterized hardware model, emphasizing the simulation of heterogeneous nodes with GPUs. However, these papers leave modeling of distributed memory simulations as a subject of future work. These existing simulators do not natively model one-sided, active message communication or asynchronously migratable objects, where the migration of objects happens simultaneously during the execution of computational tasks.

Our simulation approach combines an application task dependency graph with a performance model to enable the evaluation of an application (DGSWEM) that *does not yet have a task-based implementation*, allowing us to forecast its performance with different load balancing strategies and estimate the benefits on large-scale runs. Furthermore, the interplay between the multi-constraint nature of the storm surge application and the effectiveness of the load balancing strategies has not been previously investigated, to the best of our knowledge.

2.2 Forecasting Hurricane Storm Surge

The details of storm surge modeling were presented in Section 1.1, and details of the DG method were presented in Section 1.2. In order to understand the communication pattern of the kernel, recall that Runge-Kutta stages are updated according to the following update rule,

$$\tilde{\mathbf{U}}^{(i)} = \sum_{k=0}^{i-1} \alpha_{ik} \tilde{\mathbf{U}}^{(k)} + \beta_{ik} \Delta t \mathcal{L}^h \left(\tilde{\mathbf{U}}^{(k)} \right),$$

where

$$\mathcal{L}^h \left(\tilde{\mathbf{U}} \right) = \mathcal{M}^{-1} \left(\mathcal{V} \left(\tilde{\mathbf{U}} \right) + \mathcal{S} \left(\tilde{\mathbf{U}} \right) - \mathcal{I} \left(\tilde{\mathbf{U}} \right) \right). \tag{2.1}$$

The DG method is parallelized by distributing elements across a set of concurrent processes (referred to as *ranks*) that cooperate by message passing. In (2.1), the volume $\mathcal{V}$, and source $\mathcal{S}$ kernels can be computed entirely locally (independent of other elements). The interface kernel is the only portion of the DG algorithm that may require non-local information, so elements not on the same rank must communicate over the network. With explicit timestepping, these equations can be thought of as a stencil code over an irregular graph.

One of the key aspects of a storm surge code is its ability to simulate inundation. Due to the numerical discretization, regions of negative water column height may occur throughout the simulation, rendering the shallow water equations meaningless, both mathematically and physically. To remedy this, an additional slopelimiter is applied after the update kernel. We use the limiter proposed in [21], which locally examines elements after each update and fixes problematic regions. One of the key features of this algorithm is its ability to classify elements as either wet or dry. The performance implication of this classification is that dry elements require almost no work, thus as the hurricane inundates the coast, elements become wet in localized regions, causing load imbalance.

2.3 The DGSim Simulator

Our DGSim simulator utilizes discrete event simulation to model the execution of the parallel application. It is expected to only run *skeletonized* applications, where heavy computation and large data allocations are omitted for efficiency. Every thread in the simulation schedules events into a global priority queue keyed on *virtual time*, so that events are processed in the correct simulation order. Threads "burn" virtual time to simulate the execution of heavy computational tasks, and message arrival times are delayed to capture communication costs. This method permits efficient large-scale simulation while retaining the salient computation and communication characteristics of the program execution. With DGSim, we are able to rapidly evaluate hundreds of simulation trials with different parameterizations in 4,000 core-hours compared to over 21,000,000 core-hours had we run DGSWEM itself, roughly translating to a 5,200x speed-up.

DGSim is designed from the ground up to model a very aggressive asynchronous execution framework, which is comprised of three layers. The lowest layer of our framework, the parallel machine, mirrors the communicating multi-threaded processes prevalent in current extreme scale computing. All inter-processes communication is expressed through one-sided, point-to-point active messages, each containing a bound function. Active messages impose no synchronization, and thus encourage a very asynchronous methodology. Each process contains a *master* thread responsible for executing active messages, and all other threads belong to a pool of *worker* threads. Messages from other processes are only executed by the master thread, but any thread may send messages to other processes. The dedication of the master thread to handling incoming messages and managing worker threads may seem wasteful, but this fits asynchronous execution well as it ensures there is a core that will remain attentive to servicing incoming messages. Production asynchronous runtimes such as Charm++ usually dedicate at least one CPU core for communication and scheduling.

The second layer of the framework's stack is our interpretation of Active Global Address Space (AGAS) [68]. This is a software layer that allows the user to place parts of their application state in a named-object store without having to explicitly manage where objects reside. Users simply `visit` objects by sending an active message to a name (as opposed to a process id) and the function will be executed on the process currently holding the object. To migrate an object, there is a `relocate` function that takes a name and a target process id and will cause the named object to migrate from wherever it currently resides to the target. Relocates and visits can be issued concurrently from any number of processes without any synchronization whatsoever. This design is similar to the chare/actor model of Charm++. They too express parallel computation as asynchronous method invocations between relocatable objects.

The top layer is a distributed tasking layer, which allows the application to describe named units of non-blocking work that are executed on worker threads. Tasks mainly communicate via satisfaction of another task's dependencies. The task graph has no explicit hierarchy (no parent or sub-tasks), and all dependencies are managed through task names. When a task has computed data required by other tasks, that task sends a satisfaction containing the data to its successors. All task satisfactions are sent through AGAS `visit`s, allowing task migrations to be issued independently of the task graph scheduler. This separation allows the scheduling logic to reside entirely in the application code, giving it easy access to pertinent metadata, and enables task migration to occur concurrently with task execution. Thus we have no need to explicitly enter or exit a load balancing phase.

2.4 Performance Model Calibration

2.4.1 Compute Cost Model

Since DGSim utilizes a skeletonized form of the kernel, we developed a model to estimate the time to execute each task. As there is a limitation associated with the amount of fine-grain parallelism we can expose due to scheduling overhead, we agglomerate elements together into *tiles*, whose size is sufficiently large such that the performance improvement obtained via exposing more parallelism to the system amortizes the scheduling overhead. Furthermore, more tiles than worker threads are assigned to each rank. This oversubscription of compute resources allows the scheduler to hide message latencies.

Given a tile $\mathcal{T}$, that consists of the elements $\{\Omega_e\}$, we approximate the execution time to advance $\mathcal{T}$ by one RK stage, $t_{\mathcal{T}}$ by $t_{\mathcal{T}} = \sum_{\Omega_e} \omega_e \tau_e$ where ω_e is 1 if the element is wet and 0 if it is dry, and τ_e is the time required to advance 1 element by 1 RK stage. Timing DGSWEM using polynomial order $p = 2$ with a modal filter and the wetting and drying limiter on a single Edison node, we measured τ_e to be $2.62\,\mu$s.

To determine the wet/dry status of elements at each time step, we use a hurricane simulation of a synthetic storm from a FEMA flood insurance test suite throughout this chapter, which we refer to as *Storm36*. The test problem is a 4 day simulation with a timestep of 0.25 seconds using a 2-stage RK method on a 3.6 million element unstructured mesh. Using this benchmark, the wet/dry state of each element is recorded every 1200 time steps in DGSWEM, and the recorded states are then interpolated for intermediate time steps in DGSim.

Due to constraints on the length of simulation, DGSim only simulates every tenth timestep. This conservatively approximates the available work to hide the load balancing costs, but still models the full imbalance generated by the simulated surge. The other contributors to compute time are the task scheduler overhead and the cost to run the load

balancer. We use timers to measure the actual costs of these computations on the host machine.

2.4.2 Communication Cost Model

In order to model the delay incurred by messages sent between ranks in the machine, we defined a hierarchical communications model with varying costs associated with sending messages across different memory levels. On many current and future memory architecture designs, the memory on a compute node is split into separate partitions. In such a configuration, the cost of accessing data from the closest partition is lower than for distant partitions, thus creating non-uniform memory access (NUMA) domains. For example, on the NERSC Edison supercomputer [96], each node contains two NUMA domains, one for each CPU socket containing 12 compute cores.

Our execution model instantiates one rank per NUMA domain, each with eleven *worker threads* dedicated to executing compute tasks and one *communication thread* dedicated to send and receive active messages. We also model messages between nodes, resulting in three message categories: intra-NUMA, inter-NUMA, and inter-node. Given a message source, destination, and size, our simulator's performance model estimates the delivery delay by summing per-message (latency/overhead) and per-byte (bandwidth) costs over the path connecting the ranks. For our experiments, we utilize performance model parameters simulating a machine similar to Edison [96]. Table 2.1 summarizes the parameters used in our model to estimate message delivery costs. We conservatively assume that communication links are utilized by all cores that share them (e.g. multiples cores sharing the DDR3 bus).

All messages are subject to two memory copies over the DDR3 memory bus: one to copy the message from application memory to a communications buffer, then an additional copy at the destination from the communications buffer to the final address. For intra-NUMA messages, these two copies constitute the entire delivery cost. The measured STREAM

Table 2.1: Message transmission costs

Communications Performance Model		
Link	Latency (per-message)	Bandwidth (per-core)
1866MHz DDR3	77 ns	4.3 GiB/s
Intel QuickPath	132 ns	11.5 GiB/s
Cray Aries	1.2 - 1.5 μs	0.2 - 9 GiB/s

bandwidth on Edison is 103 GiB/s [96], thus the per-core bandwidth is 4.3 GiB/s.

For inter-NUMA messages, messages must additionally traverse the inter-socket communications bus. On Edison, the sockets are connected via the Intel QuickPath interconnect [63]. Since no worker threads may send or receive messages, the dedicated communications thread can use the full uni-directional bandwidth (11.5 GiB/s) for sending messages. The latency for intra-/inter-NUMA messages is measured using Intel®Memory Latency Checker–v3.5 with a 200 MB buffer on NERSC's Edison.

For inter-node messages, we parameterize our model using data from performance benchmarks conducted using the Cray Aries interconnect [42]. Although the Aries interconnect has a dragonfly topology, for simplicity, we assume a uniform cost of communication between ranks. Figures 7 and 8 in that paper specify how the message latency and bandwidth costs vary with message size. Since we simulate two communicating ranks per node (one per socket), we conservatively halve the reported bandwidth. Section 2.6.1 presents a validation of our performance model, comparing our modeled execution time versus empirically observed execution time of a skeletonized DGSWEM implementation.

2.5 Balancers

2.5.1 Theoretical Preliminaries

In order to express our load balancing algorithms, we first present the precise load balancing problem and then introduce our model terminology and the trellis load balancing concept.

As mentioned in Section 2.4, elements are agglomerated into tiles to amortize the scheduling overhead. The simulation consists of many tasks, each one responsible for advancing a single tile by one RK stage. Based on the DG kernel (2.1), any edge between two elements on different tiles constitutes a dependency. In order to minimize the number of dependencies and expose the largest amount of asynchrony, we group elements into tiles using the METIS k-way graph partitioning algorithm [70].

Each tile i will have a memory space requirement m_i, and an amount of work required to advance the tile by one RK stage will be given by t_i. We now introduce the assignment variables, χ_{ik}, where

$$\chi_{ik} = \begin{cases} 1 & \text{if tile } i \text{ is on rank } k \\ 0 & \text{otherwise} \end{cases},$$

and χ_{ik} is subject to the following constraints:

$$\sum_{k=1}^{n_{\text{ranks}}} \chi_{ik} = 1 \quad \forall\, i = 1, \ldots, n_{\text{tiles}}. \tag{2.2}$$

Ideally, we would solve for time-dependent values of χ_{ik} that minimize the total application execution time. However, for asynchronous applications, this is an NP-hard mixed integer optimal scheduling problem, which is not feasible to solve in situ. Since the compute cost of the tiles dominates the execution time, we make the simplifying assumption that the execution time is approximately proportional to

$$T = \max_k \sum_{i=1}^{n_{\text{tiles}}} t_i \chi_{ik} \tag{2.3}$$

Additionally, we also have a memory constraint, namely, if m_i is the memory required for

tile i and M_k is the available memory on rank k, then we obtain the additional constraints:

$$\sum_{i=1}^{n_{\text{tiles}}} m_i \chi_{ik} \leq M_k \quad \forall k = 1, \ldots, n_{\text{ranks}}. \tag{2.4}$$

Our optimization problem is defined by (2.2), (2.3), and (2.4). Note that since the tile's compute cost changes as the simulation progresses, the optimal assignment $\{\chi_{ik}\}$ is also a function of the simulation's progress. Furthermore, the irregularity associated with the wetting and drying algorithm requires that the memory constraint (2.4) and the execution time (2.3) are accounted for separately. Before we discuss approaches to approximate optimal assignments $\{\chi_{ik}\}$, we introduce some formalism upon which we will base our load balancing strategies.

The trellis approach

The main idea of this approach is to execute a second, parallel task dependency graph to handle load balancing that is completely independent of the application task graph. The motivation for this approach is twofold: it decouples the load balancer from the application execution, avoiding the need for costly control synchronization and it accurately accounts for the improvement in load balance with the cost of making load balancing decisions.

First, a common strategy taken by semi-static load balancers is to periodically pause application execution, issue data and task migrations (rebalancing phase), and resume application execution. This strategy is convenient because it decouples the times during which tasks and data are allowed to be in transit from the times that tasks are executing and sending messages to each other. Unfortunately, in order to enter and exit the rebalancing phase, control dependencies are added to the application task graph (often in the form of global synchronization), which can severely impact the performance of a highly asynchronous execution model. The trellis approach introduces an ancillary set of tasks that collect the

application state, make rebalancing decisions, and launch tile migrations, all without adding synchronization to the application execution.

Another advantage of the trellis approach is the fixed frequency and natural amortization of the cost of rebalancing. Were the frequency of rebalancing instead linked to application progress, the frequency of rebalancing would *decrease* as the load balance *worsens* since each iteration would take longer to complete. Additionally, assuming reasonable strong scaling, doubling the simulation core count would approximately double the frequency of the load balancer even though the cost of the load balancer would remain the same. In practice, there is a trade-off between the improvement in run-time due to improved load balance and the cost of rebalancing. The trellis approach allows us to simplify our cost model, reducing the numbers of parameters that need to be tuned in order to maintain effective load balancing.

Local models

The second bit of formalism we would like to introduce, are *local models*. These models capture local states of the system that can be used to make load balancing decisions. The use of local models reduces the need to aggregate global information; however, it is infeasible to keep all model information up-to-date at all times, thus the information in these models can become stale. If a rank steals a tile based on stale information, the tile migration could in fact negatively impact the load balance. Since this information is exclusively used for load balancing, the correctness of the numerical simulation remains unaffected. For our problem, we consider 3 types of models:

1. *Tile model*: Information such as the location and compute load of neighboring tiles.

2. *Rank model*: Information such as how much work is located on a rank (may aggregate local tile models).

3. *World model*: Information about the global state of the simulation (may aggregate tile

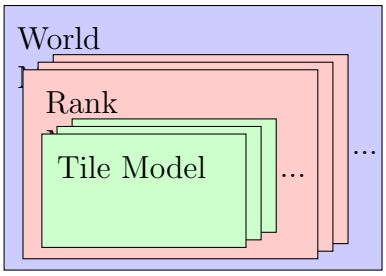


Figure 2.1: Each world model is able to aggregate rank models, which aggregate tile models. This allows for representations of the system with various levels of completeness.

and rank models).

The nested nature of these models is represented in Figure 2.1. These models provide representations of the simulation upon which the load balancers make relocation decisions.

2.5.2 Static load balancing

By re-imagining the wetting and drying algorithm as a type of multirate timestepping, where the dry elements have an infinitely large timestep and the wet elements advance at the implemented timestep, we use the static load balancing approach outlined in [112]. Here the goal is to assign an equal number of dry elements and wet elements to each rank. This is accomplished by using the multi-constraint graph partitioning outlined in [71]. Note that balancing the memory and load constraints and balancing the wet and dry elements are equivalent formulations of the same load balancing problem.

2.5.3 Dynamic load balancing

Dynamic load balancers in our context are defined as load balancing strategies that operate in an entirely *distributed, asynchronous* manner. The methods relocate tiles based on a "load pressure"—the local difference in loads. The idea is to issue many individual relocation requests based on these local observations, and thereby achieve a global balance.

To accommodate both memory and load balance objectives, we implement the refinement

Listing 2.1: Asynchronous Diffusion Tile Model

```
struct TileModel {
  int my_rank;
  map<int,int> neigh_rank;
  double wet_fraction;
  double get_gain(int rank);
  void local_broadcast();
};
```

Listing 2.2: Asynchronous Diffusion Rank Model

```
struct RankModel {
  map<int,pair<double,double>> rank_weight;
  StealQ boundary_data;
  void local_broadcast();
  void update_boundary(int leaving);
  void steal_one_tile();
};
```

procedure used in [71]. Specifically, our asynchronous diffusion approach includes no world model, but we include tile models and rank models outlined in Listings 2.1 and 2.2.

The tile model consists of the tile's wet fraction, the location of neighboring tiles, and a communication gain function, which determines the net impact on inter-rank communication were the tile to be relocated. The rank model aggregates its tile models to include the tiles bordering the rank, changes in the rank's load balance, and a list of neighboring ranks. These neighboring tiles are stored in the `StealQ`. A rank periodically broadcasts its load balancing information to its neighbors, allowing ranks to aggregate the load state of neighboring ranks. The execution model does not guarantee message ordering. In order to ensure correctness, both tile and rank models call a `local_broadcast` function, which updates neighbors' rank and tile states at regular intervals. The `StealQ` contains two STL maps for each rank. This data structure allows us to prioritize, which tile to steal to balance an imbalance in wet or dry tiles from a given rank. The maps are sorted by the amount of extra communication that would be required after stealing a given tile; we prioritize stealing tiles that minimize

Algorithm 1 Asynchronous Diffusion Thresholding

```
Set r to current rank ID
Set Σ_W ← W(r) + Σ_{n∈neighbors(r)} W(n)
Set Σ_D ← D(r) + Σ_{n∈neighbors(r)} D(n)
for all n ∈ neighbors(r) do
    if W(r) + αG < W(n) then
        Insert n into PQ(r) with weight W(n)/Σ_W.
    end if
    if D(r) + βG < D(n) then
        Insert n into PQ(r) with weight D(n)/Σ_D.
    end if
end for
```

this additional communication.

During the execution of a task, the tile checks to determine if the wet fraction has switched between wet and dry states. In the case this happens, the tile's local broadcast updates neighboring tile models. AGAS `visit` is utilized to ensure that updates are delivered to the tiles no matter where they are located. The rank model periodically calls `steal_one_tile()`. Based on the rank model's estimation of the neighboring ranks' work and memory loads, the rank assembles a priority queue $\mathcal{PQ}$ sorted by normalized wet- and dry-element counts. Only ranks that are overworked or have too many tiles will be inserted into this queue; overburdened ranks will never steal tiles. Karypis and Kumar note this stealing heuristic may lead to chatter, which is when tiles get passed back and forth between the same ranks. To mitigate this, we introduce admissible pressure gradients, i.e. a required minimum imbalance to warrant being inserted into the pool of candidate ranks to be stolen from. Specifically, let the granularity $\mathcal{G}$ be the largest number of elements per tile. For a given rank r, let $\mathcal{D}(r)$ be the number of dry elements on rank r and $\mathcal{W}(r)$ the number of wet elements on rank r. The precise heuristic for inserting elements is shown in Algorithm 1.

Then, we attempt to find the tile on the most imbalanced neighboring rank, which borders

the stealing rank and would result in the best communication gain. If there is no satisfactory tile to select, we attempt to improve the most imbalanced constraint associated with the next most-imbalanced rank.

2.5.4 Semi-static load balancing

While the dynamic load balancing procedure allows for fully distributed load balancing decisions, it is unclear that this will result in a good global load balance. The multi-constraint partitioning problem may result in unconnected partitions [71]. Additionally, the thresholding heuristic is unable to correct gradual–yet large–load imbalances.

To ensure, that we have a good load balance, we consider a semi-static approach which periodically rebalances the global model. In our formalism, the tile model updates the world model with its wet fraction at a fixed frequency. Therefore, we can trigger a load balance when all the tiles have sent their information without having to perform any synchronization. The global model incorporates all the information required to perform load balancing, i.e. the current tile partition and an approximation to the current wet fractions of the respective tiles. The global model is the only entity which may issue `relocate`s, as such the tile partition stored in the global model is always accurate.

Semistatic rebalancing involves constructing an updated tile-to-rank assignment based on the constraints associated with the updated wet fractions using the multi-constraint graph partitioning algorithm [71]. To keep the master threads available to process messages, we offload the relatively expensive rebalancing operations onto the worker threads where the global model is stored. After obtaining this new partition, we potentially have the means to load balance the system. However, the multi-constraint graph partitioner is unable to take into account the previous location of the tiles. As a worst case scenario, the graph partitioner could return the same partition with permuted partition IDs. The resulting relocation would require migration of every tile, whereas simply maintaining the old partition we would achieve

Table 2.2: DGSim parameters for load balance comparison.

Machine		DGSWEM	
NUMA domains/node	2	Runge-Kutta Stages	273600
Threads/NUMA domain	12	Polynomial order (p)	2
CPU clock-speed	2.4 GHz	Tiles/worker thread	4
Asynchronous Diffusion		Semi-Static	
Rebalance Frequency	40 ms	Rebalance Frequency	5 s
α	1		
β	2		

the same load balance without disruption. In order to remedy this, we solve a minimization problem which determines a permutation in the global rank ID names, which minimizes the number of tiles to be migrated. Using a greedy method, we construct a priority queue of old ID and new ID pairs weighted by the number of tiles that reside in both partitions. We then pop the members of the priority queue until all of the ranks have been assigned new IDs. This approach can have a significant impact on the number of tiles migrated: for example, during a 1200 core run with 4,400 tiles, the greedy assignment reduced the total number of tiles migrated from roughly 317,000 to 106,000.

2.6 Numerical Experiments

2.6.1 Empirical Validation of DGSim

To ensure that our simulator is accurate, we compared the DGSim execution times for the Storm36 synthetic hurricane simulation to a skeletonized version of DGSWEM run on NERSC's Edison. The skeletonized DGSWEM implements the programming model outlined in Section 2.3. Inter-process communication is achieved through one-sided asynchronous remote procedure calls using UPC++ [4], and tasks are executed using a dataflow execution model with a master-worker thread organization. One feature not implemented in the skeletonized DGSWEM is the AGAS layer. However, for the statically balanced problems considered for this validation, AGAS overhead is negligible due to caching of neighbors' ranks. By burning identical worker thread execution times, the validation examines whether the

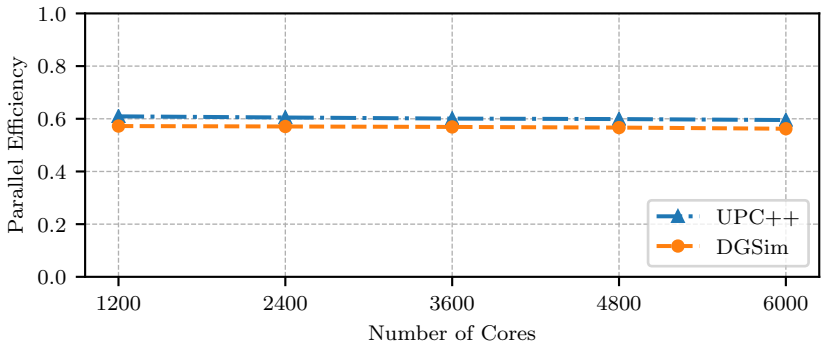


Figure 2.2: Comparison between the parallel efficiency of the simulated performance (DGSim) and the skeletonized application code (DGSWEM) on NERSC's Edison supercomputer for 1200 to 6000 cores for the Storm36 hurricane using Machine and DGSWEM parameters as outlined in Table 2.2. Each configuration was run five times. The standard deviations for a given core count were each below 1%.

messaging and threading overhead models as described in Section 2.4.1 are reasonable. The parallel efficiencies for the two simulations are shown in Figure 2.2. Differences in execution times between the simulation and the skeletonized DGSWEM application do not exceed 7% of the runtime. Furthermore, the DGSim execution time was slower than the DGSWEM execution time for all simulations, reflecting the conservative design of our cost models. These validation results demonstrate that despite our relatively simple network model, DGSim predicts execution times for this particular hurricane simulation with accuracies comparable to more sophisticated approaches, e.g. [64, 116].

2.6.2 Load Balance Comparison

In order to compare the load balancing algorithms outlined in Section 2.5, we compare performance for the hurricane used in the previous subsection using a fixed machine and algorithmic configuration outlined in Table 2.2. The load balancer parameters are based on parameter sweeps done at 1200 and 3600 cores.

To quantify the quality of our load balancing algorithms, we use two performance met-

Table 2.3: Performance of load balancers for Storm36

	Static	Multi-constraint static		Asynchronous diffusion					Semi-static				
Cores	T_{ST}	T_{MC}	S_{ST}	T_{AD}	TM	σ_{TM}	S_{ST}	S_{MC}	T_{SS}	TM	σ_{TM}	S_{ST}	S_{MC}
1200	2,613	2,048	1.28	1,686	3,654	145	1.55	1.21	1,675	723,109	2,325	1.56	1.22
2400	1,310	1,058	1.24	849	9,532	189	1.54	1.25	856	681,654	557	1.53	1.24
3600	876	698	1.26	571	16,614	319	1.53	1.22	585	641,950	568	1.50	1.19
4800	659	553	1.19	431	24,402	319	1.53	1.28	452	593,709	3,932	1.46	1.22
6000	528	444	1.19	348	36,955	761	1.52	1.28	373	574,109	1,255	1.41	1.19

Note: All times are reported in seconds. For each case, 5 runs were performed. The standard deviation of all execution times was found to be below 2% of the mean. T corresponds to the execution time in seconds. TM refers to the number of relocated tiles. The speed-up S_X is the speed-up of the simulation relative to strategy X at the given core count. Due to non-determinism in our simulation, we've reported standard deviations of tiles moved σ_{TM} for a sample size of 5.

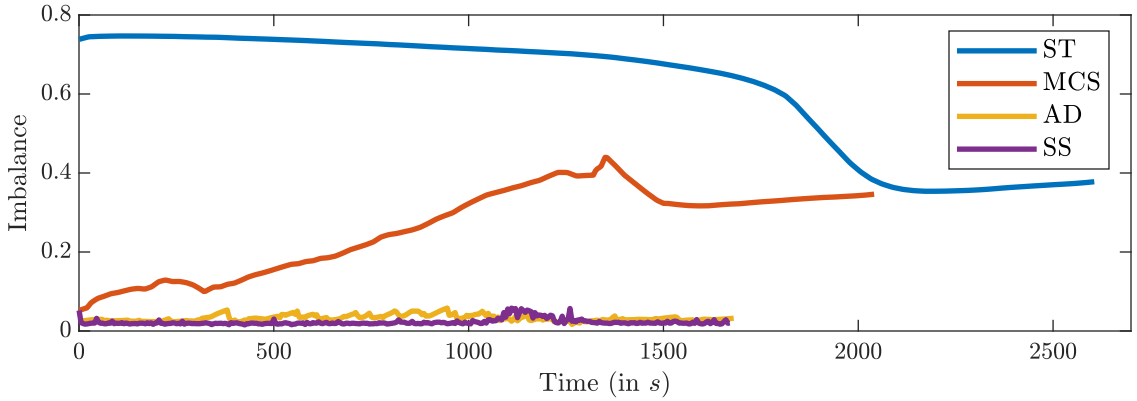


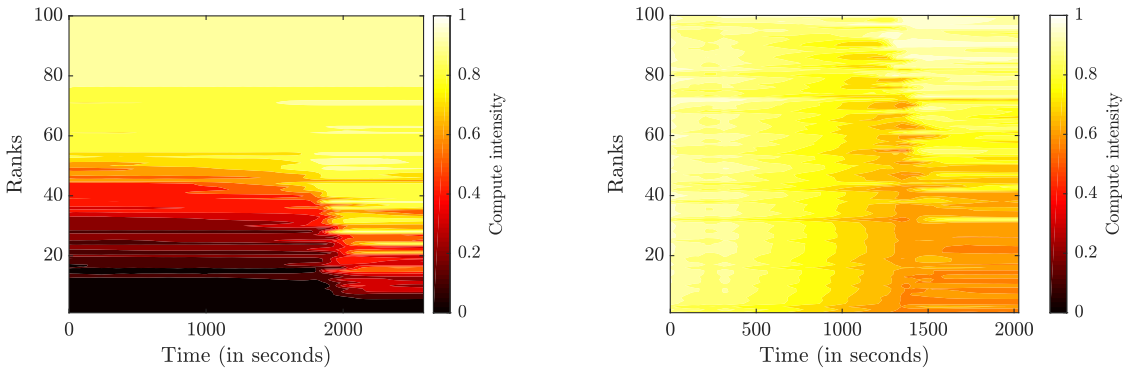
Figure 2.3: Load imbalance for Storm 36 using 1200 cores with the configuration outlined in Table 2.2. The following load balancing strategies were evaluated: static (ST), multi-constraint static (MCS), asynchronous diffusion (AD), and semi-static (SS).

rics: the *compute intensity*, which is defined for a given rank as the fraction of time spent computing at a given instant in the simulation, and the *imbalance*, which is defined as

$$I = \frac{\max T - \overline{T}}{\overline{T}},$$

where $\max T$ is maximum load on a given rank and $\overline{T}$ is the average load across the system. The combined performance results are shown in Figure 2.3 and the elapsed times and speed-ups in Table 2.3.

Since the traditional static load balancing approach solely distributes tiles to ranks to satisfy the memory constraint, it is expected that the static curve in Figure 2.3 has a high imbalance. This is reflected in Figure 2.4a, where roughly half the ranks are underutilized until the storm inundates the coast. This load profile is consistent with the 60% parallel efficiency observed in Figure 2.2. The multi-constraint static load balancer takes load as well as memory into account. This results in much better load balance until the hurricane makes landfall. Once this happens, the dynamic nature of the computational load causes a strong increase in imbalance and a corresponding decrease in computational intensity. Since



(a) Static

(b) Multi-constraint static

(c) Asynchronous diffusion

(d) Semi-static

Figure 2.4: Compute intensities of the various load balancing strategies for a 1200 core simulation. For clarity, the ranks have been sorted according to average compute intensity.

the majority of the time DGSWEM simulates is prior to landfall, the multi-constraint static mesh partitioning achieves a speed-up of 1.28 versus the original static partitioning.

Both the asynchronous diffusion and semi-static load balancers are very effective at reducing load imbalance. The compute intensities shown in Figure 2.4d and 2.4c illustrate the ranks are being well utilized throughout the simulation. With roughly equal execution times, both load balancers achieve 96% of the ideal parallel efficiency. While the semi-static strategy migrates significantly more tiles than asynchronous diffusion, we have observed a small dependence of execution time in DGSIM on tiles moved. AGAS's ability to overlap computation and tile migration minimizes resource starvation. Furthermore, the execution time is mainly determined by the simulation's critical path. Thus migrating tiles not on the critical path will not impact execution time.

2.6.3 Strong Scaling Study

To demonstrate that these algorithms are scalable for operationally-relevant core counts, we present a strong scaling study for up to 6000 cores. To obtain an understanding of the quality of the load balancers relative to achievable performance, we define the parallel efficiency E for load balancer LB as $E_{LB} = \frac{T^*}{n_c T_{LB}}$ where T^* is the serial execution time, n_c is the core count, and T_{LB} is the execution time at a given core count. As the master threads do not compute tasks themselves, the best attainable parallel efficiency is $91.7\% = 11/12$.

Using the parameters Table 2.2, the parallel efficiencies for the four partitioning strategies are shown in Figure 2.5. Note that by fixing tiles per worker thread, the task granularity decreases as we scale out to higher core counts. Firstly, we note that the code scales well: the static partitioning strategy loses less than 1% parallel efficiency across the range of core counts. The execution time here should be similar to that of a fully wetted mesh, and demonstrates the parallelizability of the DG method. Next, the multi-constraint static partitioner experiences a slight degradation in performance; $E^{MC}_{6000}/E^{MC}_{1200} = 92.1\%$. As the

number of cores increases each NUMA domain is assigned a smaller fraction of the mesh. Scaling out, the imbalance becomes more sensitive to local inundation, causing degradation of the strong scaling performance.

The semi-static load balancer also loses efficiency at scale with $E^{SS}_{6000}/E^{SS}_{1200} = 89.6\%$. Since the rebalance frequency is fixed by the trellis approach, as we scaled out to larger core counts, the number of times the balancer is called decreases. Furthermore, the cost of repartitioning the tile graph increases. At 1200 simulated cores, the METIS calls take approximately 700 ms. These timings increase to around 3700 ms at 6000 simulated cores. We suspect that the performance degradation is due to both the faster rate that imbalance is introduced, as well as a mismatch between the load balance used to compute the repartitioning and the load balance at the time when the AGAS migrates get issued. Ultimately, the performance appears to remain robust, with the semi-static partitioner maintaining a speed-up of 1.19 over the multi-constraint static partitioner at 6000 cores.

The asynchronous diffusion load balancing scales the best with $E^{AD}_{6000}/E^{AD}_{1200} = 96.9\%$. Since the cost of rebalancing is entirely local, the computational complexity of load balancing decisions does not grow with the number of cores. Furthermore, since the rebalancing frequency is roughly two orders of magnitude higher than the semi-static rebalancing frequency, the asynchronous diffusion approach does not struggle with increased irregularity due to higher core counts. A common problem with strong scaling of diffusion-based algorithms is the persistence of small gradients. Even with finer tiles at large core counts, as the rank graph grows, the maximum allowable imbalance grows as well. This did not appear to be a problem here. However, we do not claim that this approach would work equally well for other applications. Ultimately, the asynchronous diffusion worked very well providing a speed-up of 1.28 over the multi-constraint static partitioning strategy at 6000 cores, and achieved 93.7% of the maximum attainable speed-up.

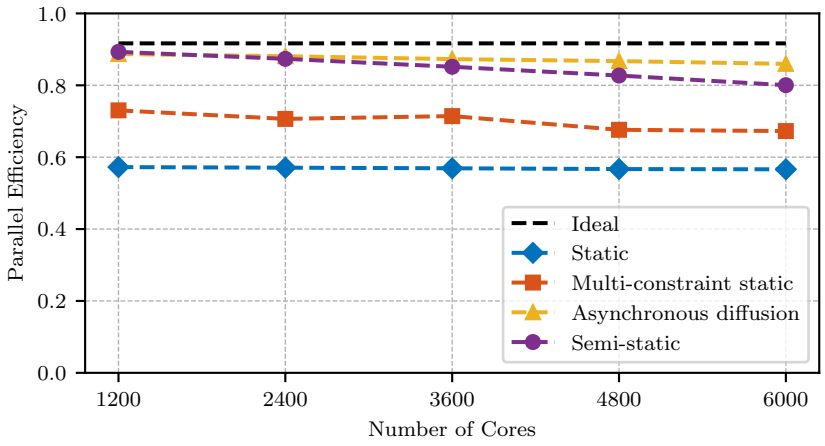


Figure 2.5: Parallel efficiency of the various load balancing strategies on Edison for 1200 to 6000 cores.

2.7 Conclusion and Future Work

Hurricane storm surge simulation stands to benefit greatly from the dynamic load balancing techniques outlined in this paper. The irregularity introduced by flooding requires load balancers to simultaneously account for both memory and compute load. We have presented a dynamic diffusion-based and a semi-static load balancing approach for an asynchronous, task-based implementation of DGSWEM. To enable rapid prototyping and evaluation of these load balancing strategies, we simulated DGSWEM using a discrete event simulation approach, which allowed us to evaluate configurations 5,000 times faster than running the actual code (in terms of core-hours). We found that static multi-constraint partitioning gives a speed-up of 1.28 over static single-constraint balancing. Both semi-static and asynchronous diffusion approaches worked very well, achieving speed-ups of 1.2 over the static multi-constraint partitioning strategy at 1200 cores. The asynchronous diffusion approach scaled best, maintaining a speed-up of 1.52 over the original static partitioning approach at 6000 cores. Furthermore, the asynchronous diffusion approach migrated an order of magnitude fewer tiles than the semi-static approach.

The irregularity of the hurricane limits the need for more sophisticated approaches. By only simulating every tenth timestep, we have exaggerated the rate of change of the imbalance by a factor of ten. In practice, we expect the methods proposed here to completely balance the compute load. Implementing these load balancing strategies in DGSWEM is a topic of future work.

The nature of simulating DGSWEM opens up many interesting avenues of future research. We plan to model parallel performance behavior of DGSWEM on future supercomputer architectures, e.g. x86 many-core, GPU, and RISC cores. Additionally, we would like to perform a co-design exploration of algorithmic and software optimizations in conjunction with the hardware parameter space. Increased communication costs on future architectures will likely play a role in choosing between load balancing strategies that trade between load balance quality, application communication costs, and tile migration costs.

Chapter 3

Adaptive Total Variation Stable Local Timestepping for Conservation Laws[1]

3.1 Introduction

The total variation stability results for scalar conservation laws form the basis of the robustness which has led to the popularity of finite volume and discontinuous Galerkin finite element methods. Roughly, the Courant-Friedrichs-Lax (CFL) condition stipulates that under a restriction on the timestep as a function of wave speed $|\Lambda|$ and mesh size Δx,

$$\Delta t \leq \frac{C_{CFL}\Delta x}{|\Lambda|}, \tag{3.1}$$

the numerical solution will weakly converge to a weak solution [81]. However, for large scale simulations, it is computationally too expensive to check that (3.1) is enforced due to high communication overhead. In practice, timesteps are set to sufficiently small values.

Enforcing a global CFL condition or using a sufficiently small global timestep tends to be overly conservative for large portions of the mesh. For example, in the case of hurricane storm surge simulations, meshes are used with length scales ranging from $\mathcal{O}(10\,\mathrm{m})$ to $\mathcal{O}(1\,\mathrm{km})$ [32].

[1]This chapter is based on the following publication currently under review: **Maximilian Bremer**, John Bachan, Cy Chan, Clint Dawson. "Adaptive Total Variation Stable Local Timestepping for Conservation Laws." Preprint available at arXiv:2003.09020 [math.NA]. Maximilian Bremer contributed 65% of the work, John Bachan contributed 15% of the work, Cy Chan contributed 10% of the work, and Clint Dawson contributed 10% of the work in this chapter.

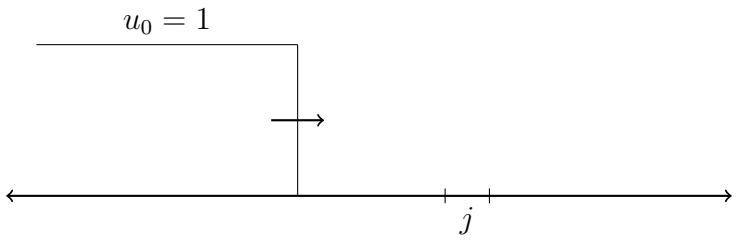


Figure 3.1: Riemann problem for Burgers' equation with initial conditions, Eqn. (3.2).

Local timestepping relaxes the global CFL constraint (3.1) by allowing different regions of the mesh to advance with different local timesteps. Similar to synchronous timestepping, existing methods are unable to account for significant temporal variations in the wave speed $|\Lambda|$.

As a motivating example, consider (inviscid) Burgers' equation ($f(u) = u^2/2$), with initial conditions,

$$u_0(x) = \begin{cases} 1 & u < 0, \\ 0 & u > 0. \end{cases} \tag{3.2}$$

The solution to this problem is a shockwave moving with speed $1/2$ to the right, i.e. $u(x,t) = u_0(x - t/2)$. The local wavespeed is given by $|\Lambda|(x) = |u|(x,t)$. For clarity, we have depicted the schematic in Figure 3.1. Consider the cell j downstream of the shock. Were one to naively evaluate the CFL condition (3.1), they would incorrectly determine that cell j is able to step arbitrarily far. In doing so, as the shock arrives at cell j, it would be unable to pass through the cell, and mass would accumulate at the interface. Such a scheme is unable to converge.

Current theoretical results require knowing the entire temporal history of the flux, before being able to determine that a timestep will be stable. Therefore, one cannot look only at neighboring values, but must consider all flux values between the last update and the next update. While existing works can assess whether or not a timestep will be stable, they fail to describe an algorithm that will advance the system in a manner that guarantees stability.

This work addresses this issue.

The main result of this chapter is a novel adaptive local timestepping algorithm, which is provably total variation stable. The algorithm recasts local timestepping as a discrete event simulation, which allows cells to dynamically coarsen and refine their timesteps. A proof of correctness is supplied to verify that under a sufficiently small minimum timestep, the algorithm will stably advance the system to an arbitrary final time, t_{end}. While the presented algorithm is first order in time, this work makes two fundamental contributions, which will enable local timestepping to become feasible for nonlinear hyperbolic problems in a massively parallel computing environment:

1. *The application of loop invariants in the proof of correctness*: Loop invariants are a proof technique that guarantee a program is formally correct, and have been used with great success in the linear algebra community [10]. Given the highly asynchronous context in which our timestepping method is executed, it is extremely difficult to debug such a program. By using these formal correctness techniques we have machinery which ensures that the algorithm achieves desired mathematical properties, e.g. the CFL condition. This not only strengthens the confidence in the results produced by the algorithm, but provides a systematic way to manage the complexity associated with higher order timestepping methods. We expect this proof technique to be indispensable in the extension of this timestepping scheme to higher orders and multiple dimensions.

2. *A discrete event formulation which removes artificial synchronizations*: Many current timestepping formulations consider only two timestepping groups. Extending these formulations to more timestepping groups is done in a recursive fashion. However, parallelizing these methods then requires a synchronization between each fine timestep to allow for timestepping adapativity. These approaches are fundamentally limited by the parallelism in the finest timestepping group. Using hurricane storm surge as an

example, there exist $\mathcal{O}(10^4)$ elements in the finest timestepping group in a mesh that consists of $\mathcal{O}(10^6)$ elements [33]. Amdahl's law severely restricts the scalability of any such formulation. By using a discrete event simulation, we consider arbitrarily many timestepping groups. Furthermore, we can leverage the extensive work done for parallel discrete event simulation to arrive at an efficient parallel implementation. Parallel performance is demonstrated for a single node on Stampede2's Skylake architecture using Devastator, one such parallel discrete event simulator [25]. We also introduce a performance model for the adaptive, locally time-stepped method, which is used to both load balance the execution and explain the observed performance gains relative to theoretical speedup bounds. These results highlight the ability of the algorithm to capitalize on adaptivity on mesh size as well as adaptivity in local wave speeds.

The remainder of this chapter is structured as follows. In Section 3.2, we discuss the current state of the art. Section 3.3 generalizes existing local timestepping results to account for arbitrary local timestepping. Section 3.4 presents the timestepping algorithm, which we accompany with a proof of correctness. Section 3.5 introduces the Devastator runtime along with performance optimizations for the local timestepping algorithm. Finally, numerical results are presented in Section 3.6, where we present one-dimensional results for Burgers' equation and the shallow water equations on a variety of meshes.

3.2 Previous Work

Relaxing the CFL condition for conservation laws by allowing different regions of the mesh to advance with different timesteps has been the subject of numerous studies. Osher et al. presented a first-order total variation diminishing timestepping scheme in [101]. Dawson and Kirby developed a second order method, which reduces to first order at the interface between elements stepping with different timesteps [31]. Kirby generalized this approach to

high resolution methods in [72]. Constantinescu developed a second order total variation stable method for conservation laws [30], leveraging the P-series analysis of Hairer [54]. Third order methods were presented in the work of Schlegel et al [110]. These methods can conserve mass, however are unable to demonstrate total variation stability.

Other methods have relied on high-order interpolation strategies to approximate coupled terms with larger timesteps [52, 59, 74]. Another set of local timestepping algorithms for conservation laws rely on single step methods with high-order space-time representations [82, 89, 120]. These methods are conservative and high-order, however fail to recover the total variation stability results of the partitioned Runge-Kutta approaches.

The main thrust of this chapter is a reformulation of the algorithm of Osher et al [101] to improve parallel performance. Previous work parallelizing local timestepping approaches has had mixed results. In cases, where the CFL number is fixed, e.g. for linear conservation laws with static meshes, task-based programming models have been extremely effective, with local timestepping ADER-DG methods scaling up to full system runs [17, 123].

While task-based approaches have proven successful for linear conservation laws, their implementations fundamentally rely on timestepping groups being statically determined, and thus are unsuitable for timestepping adaptivity. Ignoring dynamic changes in timestepping groups results in CFL violations and associated instabilities, e.g. [48, 121]. Other parallelization attempts for conservation laws have been made by recursively updating finer and finer timestepping groups. These methods would allow for adaptive timestepping by synchronizing after each fine timestep to allow for cells to change their timestepping groups. However, load balancing these methods requires balancing the work in each timestepping group across all ranks [109, 112]. The scalability of these approaches is limited by the amount of work in the timestepping group with the smallest timestep. Other parallelization approaches have treated the adaptive local timestepping problem as a discrete event simulation [69, 98–100, 114, 117, 122]. While these approaches achieve tremendous speed-ups,

these methods typically provide only heuristics and examples as proofs of robustness. To the authors' knowledge, no method has presented a provably total-variation stable adaptive timestepping scheme.

3.3 Theoretical Results for Scalar Conservation Laws

In this section, we are concerned with solving problems of the following form: Find $u \in L^\infty((0, t_{\text{end}}); \mathbb{T})$ such that

$$\begin{cases} \partial_t u + \partial_x f(u) = 0 \\ u(x, 0) = u_0(x), \end{cases} \tag{3.3}$$

where f is a Lipschitz flux, $u_0(x) \in L^\infty(\mathbb{T}) \cap BV(\mathbb{T})$, and $\mathbb{T}$ is the unit torus. Consider a finite partitioning of the torus, $\Omega^h = \cup_{j=0}^{n_{el}}(x_j, x_{j+1})$ as defined by the strictly increasing sequence $x_j \subset (0, 1)$. Let U_j denote the average value on the interval (x_j, x_{j+1}), and let Δ denote the forward difference operator, i.e.

$$\Delta U_k = U_{k+1} - U_k.$$

We specify an approximation to the conservation law as a set of pairs,

$$\mathcal{E} = \left\{ (t_j^n, U_j^n) \,:\, 0 \le j \le n_{el},\, n \in \mathbb{R}_+ \right\},$$

where U_j^n is the average value over the cell (x_j, x_{j+1}) at time t_j^n. We refer to this collection of space-time points as the *event trace*. We define the timestamps $\mathcal{T}_j$ on cell j, as

$$\mathcal{T}_j = \left\{ t_j^* \,:\, \exists\, U_j^* \text{ s.t. } (t_j^*, U_j^*) \in \mathcal{E} \right\}.$$

We define for cell j, the nearest previous timestep for time t as

$$\lfloor t \rfloor_j = \max\{\tau \in \mathcal{T}_j \,:\, \tau \leq t\}.$$

We similarly define the next timestep at time t as

$$\lceil t \rceil_j = \min\{\tau \in \mathcal{T}_j \,:\, \tau > t\}.$$

To achieve the desired theoretical results, we impose two constraints on the event traces. Firstly, we assume that there are only a finite number of events. Secondly, we need to restrict when timestamps are able to step relative to one another.

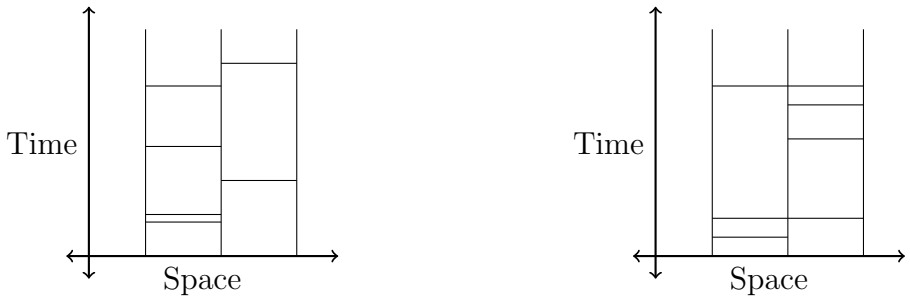
We call the set of timestamps $\cup_j \mathcal{T}_j$ *locally ordered* if for each cell j there exists a sequence of pairwise synchronization times $\{s^{\mu}_{j,j+1}\}_{\mu=0}^{n_s} \subset \mathcal{T}_j \cap \mathcal{T}_{j+1}$ such that for all consecutive synchronization times $s^{\mu}_{j,j+1}$ and $s^{\mu+1}_{j,j+1}$, either

$$\mathcal{T}_j \cap (s^{\mu}_{j,j+1}, s^{\mu+1}_{j,j+1}) = \emptyset \quad \text{or} \quad \mathcal{T}_{j+1} \cap (s^{\mu}_{j,j+1}, s^{\mu+1}_{j,j+1}) = \emptyset.$$

That is to say that one cell must step strictly faster than its neighbor between synchronization times. To illustrate this definition, we've drawn a sketch of potential local timestepping solutions in Figure 3.2. Additionally, we define the value of an approximation at any point in time and space as the most recent average value of the cell containing the given point, i.e.

$$U(t,x) = \begin{cases} U_j^n & \text{if } x \in (x_j, x_{j+1}) \quad \text{and} \quad t = t_{j^n}, \\ U(\lfloor t \rfloor_j, x) & \text{where } x \in (x_j, x_{j+1}). \end{cases}$$

We also introduce $U_j(t)$ as the value of the solution $U(x,t)$ inside cell j at time t. To approximate the flux exchange at the boundary, we define a numerical flux $\hat{F}(\cdot,\cdot)$. Additionally,



(a) Straddling (b) Locally Ordered

Figure 3.2: Comparison of two event traces.

we require that the numerical flux be:

1. *Consistent*, i.e. $\hat{F}(u^*, u^*) = f(u^*)$,

2. *Monotone*, i.e. $\hat{F}$ is increasing in the first argument, and decreasing in the second argument,

3. *Lipschitz continuous* in both arguments.

We now define the Euler approximation to the scalar conservation law as follows.

Definition 3.3.1 (Forward Euler Approximation)**.** An event trace $\mathcal{E}$ is an Euler approximation to the scalar conservation law if

$$U_j^{n+1} = U_j^n + \frac{1}{\Delta x_j} \int_{t_j^n}^{t_j^{n+1}} \left[\hat{F}(U_{j-1}(\tau), U_j^n) - \hat{F}(U_j^n, U_{j+1}(\tau)) \right] \mathrm{d}\tau, \tag{3.4}$$

for all cells j, and between all updates t_j^n and t_j^{n+1}, where $\hat{F}$ is a numerical flux.

The remainder of this section assumes the existence of forward Euler approximations. Under this assumption, we present proofs that under CFL-like constraints, a maximum principle and total variation stability can be obtained. We note that the proofs follow closely the proof presented in [72, 101]. In the next section, we will present an algorithm which satisfies the assumptions of the theorems presented in this section.

Following the spirit of the analysis proposed in [55], define

$$C_j = -\left(\frac{\hat{F}(U_j, U_{j+1}) - \hat{F}(U_{j-1}, U_{j+1})}{\Delta x_j (U_j - U_{j-1})}\right) \text{ and } D_j = \frac{\hat{F}(U_{j-1}, U_{j+1}) - \hat{F}(U_{j-1}, U_j)}{\Delta x_j (U_{j+1} - U_j)}.$$

We remark that due to the Lipschitz continuity of the numerical flux both C_j and D_j are bounded, and due to the monotonicity of the numerical flux $C_j \leq 0 \leq D_j$ for all $U_{j-1}, U_j, U_{j+1} \in \mathbb{R}$. We reformulate the update rule (3.4) as

$$U_j^{n+1} = U_j^n + \int_{t_j^n}^{t_j^{n+1}} \left[C_j(\tau)\Delta U_{j-1}(\tau) - D_j(\tau)\Delta U_j(\tau)\right] \mathrm{d}\tau. \tag{3.5}$$

Note that this representation is slightly different than the analyses presented in [55, 72, 101]. We do not multiply our C_j and D_j coefficients by Δt.

Theorem 3.3.1 (Maximum Principle)**.** *Consider an event trace $\mathcal{E}$ satisfying the forward Euler approximation* (3.4)*. If given any two events (t_j^n, U_j^n) and (t_j^{n+1}, U_j^{n+1}),*

$$1 + \int_{t_j^n}^{t_j^{n+1}} \left[C_j(\tau) - D_j(\tau)\right] \mathrm{d}\tau \geq 0, \tag{3.6}$$

then

$$|U_j^n| \leq \sup_j |U_j^0|.$$

Proof. Using the update criterion (3.5),

$$U_j^{n+1} = U_j^n + \int_{t_j^n}^{t_j^{n+1}} \left[C_j(\tau)\Delta U_{j-1}(\tau) - D_j(\tau)\Delta U_j(\tau)\right] \mathrm{d}\tau$$

Using the CFL condition (3.6), and $C_j(\tau) \le 0 \le D_j(\tau)$ for all τ,

$$\begin{aligned}
\left|U_j^{n+1}\right| &\le \left| U_j^n + \int_{t_j^n}^{t_j^{n+1}} \left[C_j(\tau)U_j^n - D_j(\tau)U_j^n\right] \mathrm{d}\tau \right| \\
&\quad + \left| \int_{t_j^n}^{t_j^{n+1}} C_j(\tau)U_{j-1}(\tau)\mathrm{d}\tau \right| + \left| \int_{t_j^n}^{t_j^{n+1}} D_j(\tau)U_{j+1}(\tau)\mathrm{d}\tau \right|, \\
&\le \left(1 + \int_{t_j^n}^{t_j^{n+1}} C_j(\tau) + D_j(\tau)\,\mathrm{d}\tau \right) |U_j^n| \\
&\quad - \int_{t_j^n}^{t_j^{n+1}} C_j(\tau)|U_{j-1}(\tau)|\,\mathrm{d}\tau + \int_{t_j^n}^{t_j^{n+1}} D_j(\tau)|U_{j+1}(\tau)|\,\mathrm{d}\tau.
\end{aligned}$$

Let $U^* = \sup_{\tau\in[t_j^n, t_j^{n+1})}\{|U_j^n|, |U_{j-1}(\tau)|, |U_{j+1}(\tau)|\}$, then

$$\begin{aligned}
|U_j^{n+1}| &\le \left(1 + \int_{t_j^n}^{t_j^{n+1}} \left[C_j(\tau) - D_j(\tau)\right] \mathrm{d}\tau \right) U^* \\
&\quad - \left(\int_{t_j^n}^{t_j^{n+1}} C_j(\tau)\mathrm{d}\tau \right) U^* + \left(\int_{t_j^n}^{t_j^{n+1}} D_j(\tau)\mathrm{d}\tau \right) U^* \\
&\le U^*.
\end{aligned}$$

Lastly, we define the minimum time between events as

$$\varepsilon = \inf_{j,k,n} \left\{ t_j^n - t \,:\, t \in \mathcal{T}_k \wedge t < t_j^n \right\}.$$

Since we've assumed a finite number of events, $\varepsilon > 0$. Letting $\mathcal{U}(t)$ be defined as the largest value up till time t, i.e.

$$\mathcal{U}(t) = \max_{j,\,\tau\le t} |U_j(\tau)|.$$

By the definition of ε, $U^* \le \mathcal{U}(t_j^{n+1} - \varepsilon/2)$. Finally, we claim $\mathcal{U}((n+1)\varepsilon/2) \le \mathcal{U}(n\varepsilon/2)$. Pick

any event at t_i^m such that $n\varepsilon/2 < t_i^m \leq (n+1)\varepsilon/2$. By the above analysis,

$$|U_i^m| \leq \mathcal{U}(t_m^i - \varepsilon/2) \leq \mathcal{U}(n\varepsilon/2).$$

Taking the maximum over all events occurring between $n\varepsilon/2$ and $(n+1)\varepsilon/2, \mathcal{U}((n+1)\varepsilon/2) \leq \mathcal{U}(n\varepsilon)$. Arguing inductively, for all events (t_j^n, U_j^n), $|U_j^n| \leq \mathcal{U}(0) = \sup_j |U_j^0|$. □

Remark 3.3.1. This proof did not require that the solution be locally ordered.

3.3.1 TVD Analysis

The main result of this section will be:

Theorem 3.3.2. *A locally ordered forward Euler solution $\mathcal{E}$ subject to the following CFL constraint: for all cells j and for all times between consecutive synchronization times, $t \in (s_{j,j+1}^{\mu}, s_{j,j+1}^{\mu+1})$,*

$$1 + (\lceil t \rceil_{j+1} - s_{j,j+1}^{\mu})C_{j+1}(t) - (\lceil t \rceil_j - s_{j,j+1}^{\mu})D_j(t) \geq 0, \tag{3.7}$$

is total variation diminishing (TVD), i.e.

$$TV(U(t)) = \sum_j |U_{j+1}(t) - U_j(t)| \leq TV(U(0)).$$

Remark 3.3.2. The CFL condition presented here looks different than the one presented in Kirby [72]. We've simply opted to use a less stringent inequality by making the time interval in the CFL condition span from the last synchronization time to the next update time rather than the next synchronization time.

Before we begin the proof, we introduce the following Lemma:

Lemma 3.3.1. *Given cell j, and two update points, t_1, $t_2 \in \mathcal{T}_j$ and $t_1 < t_2$,*

$$\int_{t_1}^{t_2} \big[U_j(t_1) - U_j(\tau)\big]\,\mathrm{d}\tau = -\int_{t_1}^{t_2} (t_2 - \lceil\tau\rceil_j)\big[C_j\Delta U_{j-1}(\tau) + D_j\Delta U_j(\tau)\big]\,\mathrm{d}\tau.$$

Proof of Lemma 3.3.1. Using the update relation,

$$\int_{t_1}^{t_2} \big[U_j(t_1) - U_j(\sigma)\big]\,\mathrm{d}\sigma = -\int_{t_1}^{t_2}\int_{t_1}^{\lfloor\sigma\rfloor_j} \big[C_j\Delta U_{j-1}(\tau) + D_j\Delta U_j(\tau)\big]\,\mathrm{d}\tau\,\mathrm{d}\sigma.$$

Changing the order of integration, yields the result

$$\begin{aligned}\int_{t_1}^{t_2} \big[U_j(t_1) - U_j(\sigma)\big]\,\mathrm{d}\sigma &= -\int_{t_1}^{t_2}\int_{\lceil\tau\rceil_j}^{t_2} \big[C_j\Delta U_{j-1}(\tau) + D_j\Delta U_j(\tau)\big]\,\mathrm{d}\sigma\,\mathrm{d}\tau,\\ &= -\int_{t_1}^{t_2} (t_2 - \lceil\tau\rceil_j)\big[C_j\Delta U_{j-1}(\tau) + D_j\Delta U_j(\tau)\big]\,\mathrm{d}\tau.\end{aligned}$$

□

Proof of Theorem 3.3.2. The proof closely follows that of [72]. We've included it here for completeness. Given an interface between cells j and $j+1$ with consecutive synchronization times $s^{\mu}_{j,j+1}$ and $s^{\mu+1}_{j,j+1}$, the update is given by

$$\Delta U_j(s^{\mu+1}_{j,j+1}) = \Delta\left(U_j(s^{\mu}_{j,j+1}) + \int_{s^{\mu}_{j,j+1}}^{s^{\mu+1}_{j,j+1}} \big[C_j\Delta U_{j-1}(\tau) + D_j\Delta U_j(\tau)\big]\,\mathrm{d}\tau\right)$$

Let $\Delta s^{\mu}_{j,j+1} = s^{\mu+1}_{j,j+1} - s^{\mu}_{j,j+1}$, we can then re-write the update rule as

$$\begin{aligned}\Delta U_j(s^{\mu+1}_{j,j+1}) = \frac{1}{\Delta s^{\mu}_{j,j+1}}\Bigg(&\int_{s^{\mu}_{j,j+1}}^{s^{\mu+1}_{j,j+1}} \Delta U_j(s^{\mu}_{j,j+1}) - \Delta U_j(\tau)\,\mathrm{d}\tau\\ &+ \Delta\int_{s^{\mu}_{j,j+1}}^{s^{\mu+1}_{j,j+1}} U_j(\tau) + \Delta s^{\mu}_{j,j+1}\big(C_j\Delta U_{j-1}(\tau) + D_j\Delta U_j(\tau)\big)\,\mathrm{d}\tau\Bigg).\end{aligned}$$

By Lemma 3.3.1,

$$\begin{aligned}\Delta U_j(s_{j,j+1}^{\mu+1}) &= \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} \Delta\left(U_j(\tau) + (\lceil\tau\rceil_j - s_{j,j+1}^{\mu})\left(C_j\Delta U_{j-1}(\tau) + D_j\Delta U_j(\tau)\right)\right) \mathrm{d}\tau \\ &= \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} \Big(1 + (\lceil\tau\rceil_{j+1} - s_{j,j+1}^{\mu})C_{j+1}(\tau) - (\lceil\tau\rceil_j - s_{j,j+1}^{\mu})D_j(\tau)\Big)\Delta U_j(\tau)\,\mathrm{d}\tau \\ &\quad + \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} (\lceil\tau\rceil_{j+1} - s_{j,j+1}^{\mu})D_{j+1}\Delta U_{j+1}(\tau) - (\lceil\tau\rceil_j - s_{j,j+1}^{\mu})C_j\Delta U_{j-1}(\tau)\,\mathrm{d}\tau.\end{aligned}$$

Taking the absolute value of both sides, and using the CFL condition and $C_j \le 0 \le D_j$ for all j and τ, we obtain

$$\begin{aligned}|\Delta U_j(s_{j,j+1}^{\mu+1})| &\le \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} \Big(1+(\lceil\tau\rceil_{j+1}-s_{j,j+1}^{\mu})C_{j+1}(\tau)-(\lceil\tau\rceil_j-s_{j,j+1}^{\mu})D_j(\tau)\Big)|\Delta U_j(\tau)|\,\mathrm{d}\tau \\ &\quad + \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} (\lceil\tau\rceil_{j+1} - s_{j,j+1}^{\mu})D_{j+1}|\Delta U_{j+1}(\tau)| - (\lceil\tau\rceil_j - s_{j,j+1}^{\mu})C_j|\Delta U_{j-1}(\tau)|\,\mathrm{d}\tau.\end{aligned} \tag{3.8}$$

Lastly, we establish bounds for $|\Delta U_j(\tau)|$. Consider the sequence of update points $\{t_k\}$ for either cell j or $j+1$ in $[s_{j,j+1}^{\mu}, s_{j,j+1}^{\mu+1}]$, i.e. $t_k \in (\mathcal{T}_j \cup \mathcal{T}_{j+1}) \cap [s_{j,j+1}^{\mu}, s_{j,j+1}^{\mu+1}]$. Importantly, $\Delta U_j(\tau)$ is constant for all $\tau \in (t_k, t_{k+1})$. Then, for r, $t_{k+1} \le r < t_{k+2}$,

$$\Delta U_j(r) = \Delta U_j(t_k) + \Delta\left(\int_{\lfloor t_k\rfloor_j}^{\lfloor r\rfloor_j} C_j\Delta U_{j-1}(\sigma) + D_j\Delta U_j(\sigma)\,\mathrm{d}\sigma\right).$$

Due to the local ordering constraint, either cell j or cell $j+1$ will only update at $s_{j,j+1}^{\mu}$ and $s_{j,j+1}^{\mu+1}$. Let us assume $\lfloor t_k\rfloor_j = \lfloor r\rfloor_j$. Taking the absolute value of both sides and grouping like terms, we obtain

$$|\Delta U_j(r)| \le |\Delta U_j(t_k)|\left|1 + \int_{\lfloor t_k\rfloor_{j+1}}^{\lfloor r\rfloor_{j+1}} C_{j+1}(\sigma)\,\mathrm{d}\sigma\right| + \int_{\lfloor t_k\rfloor_{j+1}}^{\lfloor s\rfloor_{j+1}} D_{j+1}|\Delta U_{j+1}|(\sigma)\,\mathrm{d}\sigma.$$

Using the CFL condition (3.7), it follows that

$$1 + \int_{\lfloor t_k \rfloor_{j+1}}^{\lfloor r \rfloor_{j+1}} C_{j+1}(\sigma)\,\mathrm{d}\sigma \geq 0.$$

Summing over the update points $\{t_k\}$, for $r < s_{j,j+1}^{\mu+1}$, we obtain

$$\begin{aligned}|\Delta U_j(r)| \leq |\Delta U_j(s_{j,j+1}^{\mu})| &+ \int_{s_{j,j+1}^{\mu}}^{\lfloor r \rfloor_{j+1}} C_{j+1}|\Delta U_j|(\sigma) + D_{j+1}|\Delta U_{j+1}|(\sigma)\,\mathrm{d}\sigma \\ &- \int_{s_{j,j+1}^{\mu}}^{\lfloor r \rfloor_{j}} C_j|\Delta U_{j-1}|(\sigma) + D_j|\Delta U_j|(\sigma)\,\mathrm{d}\sigma.\end{aligned}$$

Substituting this expression into (3.8),

$$\begin{aligned}|\Delta U_j(s_{j,j+1}^{\mu+1})| \leq\ & |\Delta U_j(s_{j,j+1}^{\mu})| \\ &+ \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} (\lceil \tau \rceil_{j+1} - s_{j,j+1}^{\mu}) \left(C_{j+1}|\Delta U_j|(\tau) + D_{j+1}|\Delta U_{j+1}|(\tau)\right)\,\mathrm{d}\tau \\ &- \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} (\lceil \tau \rceil_{j} - s_{j,j+1}^{\mu}) \left(C_{j}|\Delta U_{j-1}|(\tau) + D_{j}|\Delta U_j|(\tau)\right)\,\mathrm{d}\tau \\ &+ \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} \int_{s_{j,j+1}^{\mu}}^{\lfloor \tau \rfloor_{j+1}} C_{j+1}|\Delta U_j|(\sigma) + D_{j+1}|\Delta U_{j+1}|(\sigma)\,\mathrm{d}\sigma\,\mathrm{d}\tau \\ &- \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} \int_{s_{j,j+1}^{\mu}}^{\lfloor \tau \rfloor_{j}} C_{j}|\Delta U_{j-1}|(\sigma) + D_{j}|\Delta U_j|(\sigma)\,\mathrm{d}\sigma\,\mathrm{d}\tau\end{aligned}$$

Reversing the order of integration for the double integrals, we obtain

$$
\begin{aligned}
|\Delta U_j(s_{j,j+1}^{\mu+1})| \leq\ & |\Delta U_j(s_{j,j+1}^{\mu})| \\
& + \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} (\lceil \tau \rceil_{j+1} - s_{j,j+1}^{\mu}) \left(C_{j+1} |\Delta U_j|(\tau) + D_{j+1} |\Delta U_{j+1}|(\tau) \right) \mathrm{d}\tau \\
& - \frac{1}{s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} (\lceil \tau \rceil_{j} - s_{j,j+1}^{\mu}) \left(C_{j} |\Delta U_{j-1}|(\tau) + D_{j} |\Delta U_{j}|(\tau) \right) \mathrm{d}\tau \\
& + \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} (s_{j,j+1}^{\mu+1} - \lceil \tau \rceil_{j+1}) \left(C_{j+1} |\Delta U_j|(\tau) + D_{j+1} |\Delta U_{j+1}|(\tau) \right) \mathrm{d}\tau \\
& - \frac{1}{\Delta s_{j,j+1}^{\mu}} \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} (s_{j,j+1}^{\mu+1} - \lceil \tau \rceil_{j}) \left(C_{j} |\Delta U_{j-1}|(\tau) + D_{j} |\Delta U_{j}|(\tau) \right) \mathrm{d}\tau \\
=\ & |\Delta U_j(s_{j,j+1}^{\mu})| \\
& + \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} C_{j+1} |\Delta U_j|(\tau) + D_{j+1} |\Delta U_{j+1}|(\tau) \, \mathrm{d}\tau \\
& - \int_{s_{j,j+1}^{\mu}}^{s_{j,j+1}^{\mu+1}} C_{j} |\Delta U_{j-1}|(\tau) + D_{j} |\Delta U_{j}|(\tau) \, \mathrm{d}\tau
\end{aligned}
$$

Summing over all synchronization times, we obtain

$$
|\Delta U_j(t)| \leq |\Delta U_j(0)| + \Delta \int_0^t C_j |\Delta U_{j-1}| + D_j |\Delta U_j| \, \mathrm{d}\tau.
$$

Summing across all cells j gives the result. □

3.4 An Adaptive Local Timestepping Algorithm

The previous proofs dealt with certain classes of event traces and demonstrated that they were TVD. In this section, we present an algorithm that generates an event trace satisfying the hypotheses of Theorem 3.3.2. To allow for an efficient implementation, we couch the

algorithm in the language of discrete event simulations, which will allow us to rely on the extensive work done in that field for the parallelization. The main result of this section will demonstrate that with a sufficiently small minimum timestep, our algorithm will generate a total variation stable event trace.

3.4.1 Discrete Event Simulation

Discrete event simulation as a tool has been used extensively for the simulation of numerous phenomena [46]. Fundamentally, the simulation is represented as a set of actors denoted here as $\mathcal{A}$, and a set of events to be executed on an actor at a given point in time. When executed, each event may then schedule more events with later timestamps. The discrete event simulator guarantees that events will be executed in order of their timestamps.

At a high level, the actors used for the approximation of solutions to conservation laws will consist of submeshes. For this section, we require that at least two elements be assigned to each submesh. However, given an update rule with a $2n+1$-sized stencil per cell as is the case for high order finite volume methods, we require that the submesh contain at least $n+1$ cells. In practice, achieving performance requires balancing the overhead of the simulator against the amount of useful work being done during each task [15]. Therefore, submeshes should contain significantly more cells to balance simulation overheads. Each submesh can schedule one of two events: (1) *Update* ($\mathcal{U}$) the submesh to the time at which the update is scheduled, and (2) *Push Flux* ($\mathcal{PF}$), wherein the submesh sends relevant metadata to the neighboring cell to allow advancing along the shared boundary. We quantize our time to be an integer multiple of a smallest timestep, $\Delta t_{\min}$. In this section, we use t, $\lfloor t \rfloor$, $\lceil t \rceil$ as variable names and τ is used to specify timestamps in the discrete event simulation.

The actor states are described in Figure 3.3a. The actor's members correspond to

- *id*: The submesh id,
- t: The current time of the submesh,

class Submesh	class Interface
$id \in \mathbb{Z}$	$id \in \mathbb{Z}$
t, $\lfloor t \rfloor$, $\lceil t \rceil \in \mathbb{Z}$	$\lfloor t \rfloor$, $t_{\text{sync}} \in \mathbb{Z}$
u, $\Delta x \in \mathbb{R}^{n_{el}}$	u, $\Sigma \hat{F}$, $\Delta x \in \mathbb{R}$
ℓ, $r \in$ Interface	K^{int}, $K^{ext} \in \mathbb{R}$
(a) Submesh class	(b) Interface class

Figure 3.3: Local timestepping data structures

- $\lfloor t \rfloor$: The last time at which the submesh was updated,
- $\lceil t \rceil$: The next time at which the submesh is scheduled to be updated,
- u: The average density on each cell,
- Δx: The sizes of the cells for the submesh,
- ℓ, r: Representations of the states to the left and right of the submesh, respectively.

The interface class in Figure 3.3b describes neighboring cell metadata and has members describing:

- id: The id of the neighboring submesh,
- $\lfloor t \rfloor$: The last time at which the neighbor was updated,
- t_{sync}: The last time at which the neighbor and submesh both updated,
- u: The neighbor's density at time $\lfloor t \rfloor$,
- K^*: The largest Lipschitz constant of the numerical flux between time t_{sync} to $\lfloor t \rfloor$ on the internal interface (K^{int}) or external interface cell (K^{ext}),
- $\Sigma \hat{F}$: The integral of the numerical flux between the submesh and its neighbor from time $\lfloor t \rfloor$ to the current time,
- Δx: is the size of the neighboring cell.

We note that the Lipschitz constant K is used to bound C_j and D_j terms simultaneously. For commonly used fluxes like the Godunov flux or Lax-Friedrichs flux, K will look like $|\Lambda|/\Delta x$ yielding the commonly seen version of the CFL condition (3.1). Member fields are denoted using a period, e.g. $S.\ell.\lfloor t \rfloor$ denotes the previous update of the left interface of submesh S.

We next turn to describing our events. Before we define our two main events, UPDATE and PUSH_FLUX, we define some helper functions.

Definition 3.4.1 (Helper Functions)**.** • ADVANCE(T, SUBMESH): Advance the submesh one timestep from submesh.$\lfloor t \rfloor$ to t according to the update rule (3.4). Note that this will also cause corresponding updates in boundary terms such as K^* and $\Sigma\hat{F}$.

• COMPUTE_T_NEXT_BDRY(SUBMESH, NEIGH): Compute the timestep size for interfaces which depend on neigh.u. This evaluates two CFL conditions at: (1) the external interface between the submesh and its neighbor, which is synchronized at neigh.t_{sync} and (2) the internal interface nearest to the external interface, which is synchronized at submesh.$\lfloor t \rfloor$.

• COMPUTE_T_NEXT(SUBMESH, T): Compute the allowable timestep size for the submesh. This is taken to be the minimum of the largest timesteps for the values returned by COMPUTE_T_NEXT_BDRY for both neighbors and the largest allowable internal timestep.

• MAKE_MSG(SUBMESH, NEIGH): Generate a buffer with the required information to update the neighbor's corresponding interface.

• ACCUMULATE(T, SUBMESH, NEIGH): Update neigh.$\Sigma\hat{F}$ to integrate from the previous integration point neigh.$\lfloor t \rfloor$ to time t.

• UPDATE_K_BDRY(SUBMESH, NEIGH): Update Lipschitz constants K^* according to the new updated values at the boundary.

The helper functions serve as an API between the application and the timestepping algorithm. Features like which set of conservation laws or choice of discretization are encapsulated into the above function calls. In addition to the helper functions, we also define our scheduling primitives. These are the function calls with which events are scheduled in the discrete event simulator.

Definition 3.4.2 (Scheduling Primitives)**.** • SCHEDULE(T, EVENT): Schedule EVENT at time T in the discrete event simulator.

```
function update(id, update_forced)
  τ ← get_time()
  submesh ← get_submesh(id)

  if ( τ ≠ submesh.⌈t⌉ ∧ ¬force_update ) return
  if ( τ = submesh.⌊t⌋ ) return

  advance(τ, submesh)
  submesh.⌈t⌉ ← compute_t_next(submesh, τ)

  for neigh ∈ submesh.bdry
    forced_update ← neigh.⌊t⌋ > neigh.t_sync
                  ∧ compute_t_next_bdry(submesh, neigh) ≤ τ

    schedule( τ, push_flux( neigh.id, id,
                            make_msg(submesh, neigh),
                            force_update) )
    if ( forced_update ) neigh.t_sync ← τ

  if ( submesh.⌈t⌉ > τ )
    schedule(submesh.⌊t⌋, update(id, false))
```

Figure 3.4: Update function

- SCHEDULE_INLINE(EVENT): Execute EVENT in the current execution context, i.e. on that actor at that time.

With these helper functions, we now define our two main events: update ($\mathcal{U}$) and push flux ($\mathcal{PF}$). Shown in Algorithm 2, the main loop schedules initial updates at time 0 in a priority queue ($\mathcal{Q}$). As events execute, they may cause other events to be enqueued in the priority queue as pairs with the desired timestamp of evaluation as the key. The lowest timestamps receive the highest priority, and in case of ties, we impose no ordering.

Algorithm 2 Main timestepping loop

```
Initialize actors A = {S_1, ..., S_{n_sbmsh}}
Initialize Q := {}
for sbmsh ∈ A do
    schedule( 0, update(sbmsh.id, false))
end for
while ∃ q ∈ Q do
    current_event ← Q.pull_highest_priority()
    evaluate(current_event)
end while
```

We now arrive at the main result for this section.

```
function push_flux(id, id_from, msg, force_update)
  τ ← get_time()
  submesh ← get_submesh(id)
  neigh ← get_neigh(submesh, id_from)

  accumulate( τ, submesh, neigh )
  neigh.u ← msg
  neigh.⌊t⌋ ← τ
  if ( neigh.⌊t⌋ =submesh.⌊t⌋ ) neigh.t_sync ← τ
  update_K_bdry(submesh, neigh)

  submesh.⌈t⌉ ← compute_t_next(submesh, τ)
  if ( force_update ∨ submesh.⌈t⌉ ≤ τ )
    schedule_inline(update(id, true))
    return

  if ( submesh.⌈t⌉ > τ )
    schedule(submesh.⌈t⌉, update(id, false))
```

Figure 3.5: Push flux function

Theorem 3.4.1. *Given a minimum timestep size of*

$$\Delta t_{\min} < \inf \left\{ \frac{\Delta x_{\min}}{2 \max(K_1(\xi),\, K_2(\xi))} \,:\, \xi \in \text{range}(u_0) \right\}, \tag{3.9}$$

where $\Delta x_{\min}$ *is the smallest element size,* $K_1(\xi)$ *is the local Lipschitz constant of* $\hat{F}(\cdot, \xi)$ *and* $K_2(\xi)$ *is the local Lipschitz constant of* $\hat{F}(\xi, \cdot)$*. The discrete event simulator as defined in Algorithm 2 with a minimum timestep of* $\Delta t_{\min}$ *will produce a TVD solution to the scalar conservation law in* (3.3)*.*

3.4.2 Proof of Theorem 3.4.1

The proof of the theorem requires demonstrating that the discrete event simulation generates an event trace that satisfies the conditions of Theorem 3.3.2. The main machinery for this proof will be loop invariants. Loop invariants are a formal correctness technique whereby we specify a set of "correct" states. By showing that every event maps a correct state onto the set of correct states, we are able to ensure that the following state satisfies certain desired criteria. Once all events have executed, we will have proven that the simulation terminates in the correct state and the algorithm behaves as desired. This proof technique has been

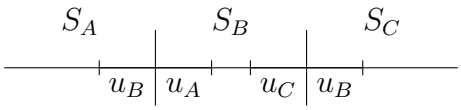


Figure 3.6: Three neighboring submeshes S_A, S_B, and S_C with bordering cells named.

utilized with great success for systematic design of algorithms for numerical linear algebra as part of the FLAME project [10, 95]. To demonstrate that the algorithm satisfies the conditions of Theorem 3.3.2, we need invariants to satisfy:

1. the local ordering principle,

2. the CFL condition,

3. and the update rule as expressed in (3.4).

Additionally, we will require two further invariants to show that the computed values are correct, and that duplicated metadata in the boundaries is consistent with the values on the neighboring submesh. To simplify, the notation in these invariants, we assume three submeshes are enumerated as, S_A, S_B, and S_C. The value of u at the cell neighboring another submesh, will be indexed with that neighbors name, e.g. the cell in submesh S_B that neighbors S_A we will reference as $S_B.u_A$. For clarity, we have depicted this naming convention in Figure 3.6.

Local ordering implies that one of two neighboring submeshes must have last updated at the last synchronization time. To ensure that the event trace will be locally ordered, we define the local ordering invariant for the left interface as

$$LO_\ell(S_B, \tau) = (S_B.\lfloor t \rfloor = S_B.\ell.t_{\text{sync}} \vee S_B.\ell.\lfloor t \rfloor = S_B.\ell.t_{\text{sync}}) \tag{3.10}$$

Note that we similarly define $LO_r(S_B, \tau)$ for the right interface. The τ in the arguments of the loop invariants refers to the simulation time at which we evaluate the invariant.

Next, we define the CFL-related invariants. These invariants will ensure that our algorithm satisfies the CFL condition (3.7). Define the invariant for the left boundary cell as

$$CFL_\ell(S_B, \tau) = (S_B.\lceil t \rceil - S_B.\ell.t_{\text{sync}}) S_B.\ell.K^{\text{ext}} \leq 1/2 \tag{3.11}$$

$$\wedge\ (S_B.\lceil t \rceil - S_B.\lfloor t \rfloor) S_B.\ell.K^{\text{int}} \leq 1/2, \tag{3.12}$$

where K is used to bound the respective C_j and D_j terms. Note that (3.11) corresponds to interface between submeshes S_A and S_B, thus the CFL condition is relative to $S_B.\ell.t_{\text{sync}}$. The other CFL condition (3.12) corresponds to the interior interface of the cell associated with $S_B.u_a$. Since internally the internal cells step synchronously, the last synchronization time is $S_B.\lfloor t \rfloor$, but since $S_B.\ell.K^{\text{int}}$ still depends on $S_B.\ell.u$, this condition must be re-evaluated after each push flux. We define a CFL invariant for the right interface, $CFL_r(S_B, \tau)$ similarly. We also define an internal CFL invariant, CFL_{int}, which can be determined by solely examining the internal state of the submesh.

$$CFL_{\text{int}} = \bigwedge_{i=2}^{n_{el-1}} \Big((S_B.\lceil t \rceil - S_B.\lfloor t \rfloor) \big[D_i(S_B.u_{i-1}, S_B.u_i, S_B.u_{i+1}) - C_i(S_B.u_{i-1}, S_B.u_i, S_B.u_{i+1}) \big] \leq 1 \Big)$$

We then define our CFL invariant as

$$CFL(S_B, \tau) = (CFL_{\text{int}} \wedge CFL_\ell \wedge CFL_r). \tag{3.13}$$

The correctness invariant CR checks that all computations are being done correctly. Here,

we use $\tilde{S}_B$ to denote the state before executing an event.

$$CR(S_B, \tau) = \big((S_B.\lfloor t \rfloor = \tilde{S}_B.\lfloor t \rfloor \wedge S_B.u = \tilde{S}_B.u) \tag{3.14}$$

$$\vee\, (S_B.\lfloor t \rfloor \neq \tilde{S}_B.\lfloor t \rfloor \wedge S_B \text{ updated according to Equation (3.4)}\big) \tag{3.15}$$

$$\wedge \left(S_B.\ell.\Sigma\hat{F} = \int_{S_B.\lfloor t \rfloor}^{\max(S_B.\ell.\lfloor t \rfloor, S_B.\lfloor t \rfloor)} \hat{F}(S_B.\ell.u(\sigma), S_B.u_A)\, \mathrm{d}\sigma \right) \tag{3.16}$$

$$\wedge \left(S_B.r.\Sigma\hat{F} = \int_{S_B.\lfloor t \rfloor}^{\max(S_B.r.\lfloor t \rfloor, S_B.\lfloor t \rfloor)} \hat{F}(S_B.u_C, S_B.r.u(\sigma))\, \mathrm{d}\sigma \right) \tag{3.17}$$

$$\wedge\ S_B.\ell.K^{\text{int}} \geq \max_{\sigma \in (S_B.\lfloor t \rfloor, \tau)} D_j(S_B.\ell.u(\sigma), S_B.u_A, S_B.u_{A+1}) \tag{3.18}$$

$$\wedge\ S_B.\ell.K^{\text{ext}} \geq \max_{\sigma \in (S_B.\ell.t_{\text{sync}}, \tau)} -C_j(S_B.\ell.u(\sigma), S_B.u_A, S_B.u_{A+1}) \tag{3.19}$$

$$\wedge\ S_B.r.K^{\text{int}} \geq \max_{\sigma \in (S_B.\lfloor t \rfloor, \tau)} -C_j(S_B.u_{C-1}, S_B.u_C, S_B.r.u(\sigma)) \tag{3.20}$$

$$\wedge\ S_B.r.K^{\text{ext}} \geq \max_{\sigma \in (S_B.r.t_{\text{sync}}, \tau)} D_j(S_B.u_{C-1}, S_B.u_C, S_B.\ell.u(\sigma)). \tag{3.21}$$

Equations (3.14) and (3.15) require that if the state has been updated, $S_B.\lfloor t \rfloor$ will reflect that, and that the state is updated according to the update rule. We check that the flux buffers are being correctly integrated at the boundary with (3.16) and (3.17). The last set of equations (3.18)–(3.21) check that the K terms used to enforced the CFL condition correctly bound the Lipschitz constants of the numerical flux. Internal values of $S_B.u$ in (3.18)–(3.21) have no dependence on the time variable σ since the solution remains constant on a cell between updates. For this invariant to be well-defined, we require that each submesh contain at least two cells. Otherwise, terms like $S_B.u_{A+1}$ would not exist.

The previous three invariants, LO, CFL, and CR, have all used information available on submesh S_B. However, when duplicating information between neighbors, we need to ensure

that the information is consistent. Thus, we define a consistency invariant,

$$CI(S_B, S_A, \tau) = \big(S_B.u_A = S_A.r.u_B \wedge S_B.\lfloor t \rfloor = S_A.r.\lfloor t \rfloor) \tag{3.22}$$

$$\vee\ \exists\,(\tau, \mathcal{PF}(S_A.id, S_B.id, S_B.u_a, \cdot)) \in \mathcal{Q}\big) \tag{3.23}$$

$$\wedge\ \big(S_B.\ell.t_{\text{sync}} = S_A.r.t_{\text{sync}} \tag{3.24}$$

$$\vee\ (S_B.\ell.t_{\text{sync}} = \tau \wedge \exists\,(\tau, \mathcal{PF}(S_A.id, S_B.id, \cdot, \cdot)) \in \mathcal{Q} \tag{3.25}$$

$$\vee\ (S_A.r.t_{\text{sync}} = \tau \wedge \exists\,(\tau, \mathcal{PF}(S_B.id, S_A.id, \cdot, \cdot)) \in \mathcal{Q}\big) \tag{3.26}$$

Here we use the dot, $\cdot$ to denote any argument. Note that we similarly define consistency for the other interface between S_B and S_C.

The last invariant we define is a progress invariant to ensure that the simulation can make progress,

$$P(S_B, \tau) = (S_B.\lfloor t \rfloor = t_{end}) \tag{3.27}$$

$$\vee\,(\exists\,(s, \mathcal{U}(s, S_B, false)) \in \mathcal{Q} : \{s = S_B.\lceil t \rceil \geq \tau > S_B.\lfloor t \rfloor\}) \tag{3.28}$$

$$\vee \exists\,(\tau, \mathcal{PF}(S_B.id, \cdot, \cdot, \cdot)) \in \mathcal{Q} \tag{3.29}$$

$$\vee \exists\,(\tau, \mathcal{PF}(\cdot, S_B.id, \cdot, true)) \in \mathcal{Q}. \tag{3.30}$$

Combining these five loop invariants, we arrive at

$$I(S_B, S_A, S_C, \tau) = LO_\ell(S_B) \wedge LO_r(S_B) \wedge CFL(S_B, \tau) \wedge CR(S_B, \tau) \tag{3.31}$$

$$\wedge\ CI(S_B, S_A, \tau) \wedge CI(S_B, S_C, \tau) \wedge P(S_B, \tau). \tag{3.32}$$

As the aim is to ensure the invariants hold for all submeshes, we will refer to $I(\tau)$ (without

the submesh arguments) as

$$I(\tau) = \bigwedge_{k=1}^{n_{\text{sbmsh}}} I(S_k, S_{k-1}, S_{k+1}, \tau).$$

Proposition 3.4.1. *A discrete event simulation which satisfies the invariant I between all events executed before time τ, will execute a finite number of events at τ and satisfy I between each of these events.*

Assuming Proposition 3.4.1 holds, the remainder of the proof of Theorem 3.4.1 follows by induction. At the start of time τ , we note that no updates have executed, and so $S.\lfloor t \rfloor < \tau$. Furthermore, since push fluxes may only be scheduled at the time at which the spawning updates are executed, there are no outstanding push fluxes. Thus, the invariant I before any events have been executed at time τ is

$$\begin{aligned} I(\tau, S_B, S_A, S_C) = {} & LO_\ell \wedge LO_r \wedge CFL_{\text{int}} \wedge CFL_\ell \wedge CFL_r \wedge CR \\ & \wedge S_B.u_A = S_A.r.u_B \wedge S_B.\lfloor t \rfloor = S_A.r.\lfloor t \rfloor \\ & \wedge S_B.\ell.t_{\text{sync}} = S_A.r.t_{\text{sync}} \\ & \wedge S_B.u_C = S_C.\ell.u_B \wedge S_B.\lfloor t \rfloor = S_C.\ell.\lfloor t \rfloor \\ & \wedge S_B.r.t_{\text{sync}} = S_C.\ell.t_{\text{sync}} \\ & \wedge \exists\, (s, \mathcal{U}(S_B.id, false)) \in \mathcal{Q} \text{ for which } \tau \leq s \leq S_B.\lceil t \rceil. \end{aligned}$$

After all events at time τ have finished executing, we end in the same state, with the exception that the progress invariant implies that every element must have an update scheduled at a time strictly greater than τ. Due to consistency and correctness, every cell which updated at time τ, was updated according to the update rule (3.4). Furthermore, consistency and correctness along with the CFL condition, imply that the event trace up until time τ satisfies the CFL condition (3.7). Once all the events have executed, remaining in I implies that

updates must be scheduled for time $\tau + \Delta t_{\min}$ or greater, since otherwise the discrete event simulation would not advance past time τ in a finite number of events. Lastly, the event trace remains locally ordered. Were it not, one of the locally ordered invariants would be violated after all events at τ had executed. Remarking that the solution remains constant from τ to $\tau + \Delta t_{\min}$, we have established the inductive hypothesis. Arguing inductively, the event trace at t_{end} is locally ordered, satisfies the CFL condition, and obeys the forward Euler update rule. Therefore, the discrete event simulation will generate a TVD solution.

3.4.3 Proof of Proposition 3.4.1

The remainder of the proof relies on demonstrating that the simulation state satisfies the invariants after each event is evaluated at time τ. However, before we proceed, we prove a progress guarantee.

Proposition 3.4.2. *Given a discrete event simulation that satisfies invariant I up until time τ, $u(\tau)$ will satisfy a maximum principle.*

Proof. Pick any cell j and consider two update points t_j^n and t_j^{n+1}. Let $s_{j-1,j}^{\mu}$ and $s_{j,j+1}^{\eta}$ represent the most recent left and right synchronization times, respectively, and let u_{j-1}, u_j, u_{j+1} represent the associated cell densities. Since the total variation CFL condition (3.7) is satisfied for all time up until τ,

$$
\begin{aligned}
&1 + \int_{t_j^n}^{t_j^{n+1}} \left[C_j(\sigma) - D_j(\sigma) \right] \mathrm{d}\sigma \\
&\quad \geq 1 + (t_j^{n+1} - s_{j-1,j}^{\mu}) \min_{\sigma \in (s_{j-1,j}^{\mu}, t_j^{n+1})} C_j(u_{j-1}, u_j, u_{j+1}) \\
&\qquad - (t_j^{n+1} - s_{j,j+1}^{\eta}) \max_{\sigma \in (s_{j,j+1}^{\eta}, t_j^{n+1})} D_j(u_{j-1}, u_j, u_{j+1}) \\
&\quad \geq 0.
\end{aligned}
$$

Thus, the event trace satisfies the maximum principle, Theorem 3.3.1. □

Lemma 3.4.1 (Progress Guarantees)**.** *Given a simulation that satisfies the invariant I up until time τ, if a submesh S_B and its neighbors S_A and S_B are both updated at time τ, then*

$$compute\_t\_next(submesh,\ \tau) \geq \tau + \Delta t_{\min}.$$

Proof. Pick any cell of S_B and consider the two neighboring cells. Enumerate the densities as u_{j-1}, u_j and u_{j+1}. By definition of the Lipschitz constant

$$-C_j(u_{j-1}, u_j, u_{j+1}) \leq K_1(u_{j+1})/\Delta x_{\min}$$

for all u_{j-1}, u_j, $u_{j+1} \in \text{range}(u_0)$. By Proposition 3.4.2, $u_{j-1}, u_j, u_{j+1} \in \text{range}(u_0)$. Therefore,

$$-\Delta t_{\min} C_j(u_{j-1}, u_j, u_{j+1}) \leq \frac{K_1(u_{j+1})}{\Delta x_{\min}} \frac{\Delta x_{\min}}{2K_1(u_{j+1})} = \frac{1}{2}$$

The proof for the right interface of cell j follows similarly. Thus, taking a timestep of size $\Delta t_{\min}$ satisfies (3.7). Since this holds for every cell in submesh S_B,

$$\text{get\_t\_next}(\text{submesh},\ \tau) \geq \tau + \Delta t_{\min}.$$

□

To see that there are at most a finite number of messages sent at each τ, note that each update will only execute the main body at most once for a given timestep. If multiple updates are scheduled for the same time τ, $S_B.\lfloor t \rfloor = \tau$ after the first update, and subsequent update events would cause the function to become a no-op. Furthermore, since push fluxes can only be scheduled when an update has executed at that timestep, we bound the total number of events scheduled at a given timestep by $3n_{\text{sbmsh}}$.

Table 3.1: All possible states as a submesh S processes various messages. Without loss of generality the first push flux is assumed to arrive from the left.

	b_1	b_2	b_3	b_4	b_5	b_6
q_a	True	True	True	True	True	True
q_b	True	False	True	True	True	True
q_c	False	True	True	True	True	True
q_d	False	True $\vee$ False	True	False	True	True
q_e	False	True $\vee$ False	True	True	True	False
q_f	False	True $\vee$ False	True	False	True	False
q_g	True	True	False	True	True	True
q_h	False	True	False	True	True	True
q_i	False	True $\vee$ False	False	True	True	False
q_j	True	True	False	True	False	True
q_k	False	True	False	True	False	True

Completing the proof requires showing that as each event at time τ is processed I is not violated. To do so, we will enumerate some states of the submesh and show that execution of any event will not lead to an invariant violation. In order to determine the state of a submesh S, we require 6 Boolean variables:

$$b_1 = (S.\lfloor t \rfloor < \tau)$$

$$b_2 = (S.\lceil t \rceil > \tau)$$

$$b_3 = (S.\ell.\lfloor t \rfloor < \tau)$$

$$b_4 = (S.\ell.t_{\text{sync}} \leq S.\ell.\lfloor t \rfloor)$$

$$b_5 = (S.r.\lfloor t \rfloor < \tau)$$

$$b_6 = (S.r.t_{\text{sync}} \leq S.r.\lfloor t \rfloor).$$

Our aim will be to derive a finite state machine and show that all transitions preserve the loop invariants. With 6 binary variables, we have 64 possible states. To reduce the complexity of the system we assume that if a submesh receives a message, the first message it processes will come from the left. All states that are attained are shown in Table 3.1

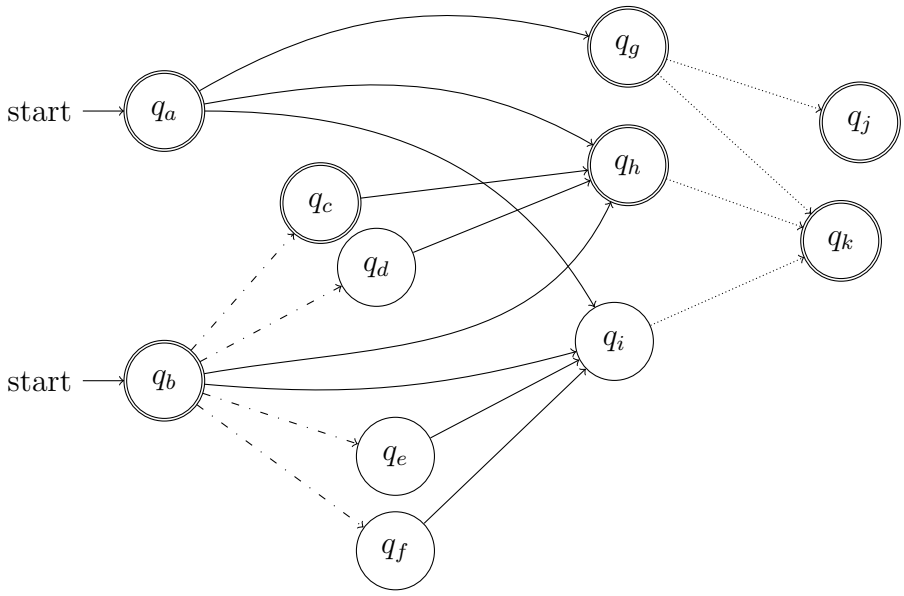


Figure 3.7: Possible state transitions during a single timestamp. Accepting states are denoted with double circles. The line styles indicate the type of message: the solid line denotes a push flux from the left neighbor, the dotted line denotes a push flux from the right neighbor, the dash-dotted line denotes an update. Without loss of generality we assume that the first push flux processed arrives from the left.

and their transitions are pictorially shown in Figure 3.7. We note the non-standard notation depicting multiple arrows coming out of given states. These multiple arrows leaving a state are due to the fact that our binary state representation doesn't fully capture the internal state of any given submesh. In particular, when any given message is processed, the algorithm deterministically computes the appropriate transition based on the values at the cells, e.g. does the submesh need to execute an inlined update. This representation could be expanded to recover the traditional finite state machine representation at significantly greater complexity.

Before we assess any state transitions,we note that the CFL invariant simply checks whether or not the proposed $S_B.\lceil t \rceil$ satisfies the invariant. The function COMPUTE_T_NEXT, which exclusively sets $S_B.\lceil t \rceil$ is constructed to always satisfy the CFL invariant, i.e. we never propose a timestep that if executed would cause a CFL violation. Thus, we will omit

the invariant in the following proof. However, the proposed timesteps may be unphysically small, i.e. zero or even negative. Equation (3.28) of the progress invariant will be used to ensure that all executed updates occur between intervals larger than $\Delta t_{\min}$.

We begin by noting that before any submesh has processed any event at time τ, the last events must have been processed at a time strictly less than τ, implying that the submesh must satisfy $b_1 \wedge b_3 \wedge b_5$. In addition, since the synchronization times are only updated when two neighboring submeshes update at the same time, both b_4 and b_6 must be true. Therefore, any submesh must begin in states q_a or q_b. These states differ based on whether or not there is an update scheduled at τ.

Processing Unforced Updates

We refer to unforced updates as updates for which the update_forced argument of the update function is false. All update events scheduled during the simulation are unforced updates. On the other hand, forced updates only occur as inlined events during a push flux evaluation. An unforced update will only be applied to submeshes for which $S_B.\lceil t \rceil = \tau$ and $S_B.\lfloor t \rfloor < \tau$. Otherwise, the update becomes a no-op. Thus, non-trivial update events can only be applied to submeshes in state q_b.

CR: Satisfying the consistency invariant before the update implies the values of the neighbors are correct until τ. Therefore, the update will satisfy (3.15). After the update, the values of the flux buffers at the edges trivially satisfy (3.16) and (3.17) since $S_B.\lfloor t \rfloor = \tau > S_B.\ell.\lfloor t \rfloor$. Assuming K is updated appropriately in ADVANCE, q_b will satisfy the correctness invariant after the execution of an update.

LO: To be locally ordered either the submesh or its neighbor must have last updated at the synchronization time. Consider the left side. If $S_B.\ell.\lfloor t \rfloor = S_B.\ell.t_{\text{sync}}$, the submesh will update without causing a local ordering violation. Otherwise, the neighbor has

updated since the last synchronization, and updating the submesh must be treated as a local ordering violation. In that case, $S_B.\ell.\lfloor t \rfloor > S_B.\ell.t_{\text{sync}}$ and the push flux scheduled during the update will force the neighbor to update at τ, meaning at some point we expect the neighbor to update at τ. Inside the submesh, the synchronization time $S_B.\ell.t_{\text{sync}}$ is set to τ in anticipation of the scheduled synchronization thus satisfying the local ordering invariant.

P: The scheduled push fluxes satisfy (3.30).

CI: Updating a submesh will cause the consistency invariant (3.22) to fail on the neighbor. However, the submesh will send a push flux. Thus, Equation (3.23) holds. If the synchronization times remain unchanged, $S_B.\ell.t_{\text{sync}} = S_A.r.t_{\text{sync}}$ continues to hold. Otherwise, the update causes a push flux, which implies that $S_B.\ell.t_{\text{sync}} = \tau$ and there exists a forced push flux to S_A.

Thus, q_b will satisfy I after processing the update. The state transition depends solely on, which sides need to receive a push flux from their neighboring submesh in order to satisfy local ordering or allow scheduling the next non-trivial update. State q_b transitions to q_c if no side requires updating, to q_d if the left side requires updating, q_e if the right side requires updating, and q_f if the both sides require updating. Since the last message was received before time τ, and a forced update would set $S_B.\ell.t_{\text{sync}} = \tau$. Precisely, b_4 and b_6 correspond to waiting on these missing flux updates. Lastly, any state that caused a forced update to be scheduled (q_d, q_e, q_f) cannot be terminal. The forced push flux implies that the neighbor to which that message was sent must send a push flux back if it hasn't already sent a flux. Therefore, these states will process at least one push flux.

Processing the Push Flux from the Left

Without loss of generality we assume that the first message comes from the left. Since no push fluxes have been processed, we only consider cases for which $b_3 \wedge b_5$ is true. Since the submesh must begin in state q_a or q_b and can transition to states q_c-q_f after executing an update, it suffices to show that the invariants remain unviolated after executing the push flux on a submesh in any one of the states q_a-q_f. There are three broad cases of states to consider: (i) submeshes that have not updated, but execute an inlined update during the push flux, (ii) submeshes that have not updated and do not update during the push flux, and (iii) submeshes that have already updated.

CR: For case (i), $\tilde{S}_B.\lfloor t \rfloor < \tau$ and the push flux causes the submesh to update. In this case, *CR* follows by the same reasoning mentioned in the previous section. For cases (ii) and (iii), $\tilde{S}_B.\lfloor t \rfloor = S_B.\lfloor t \rfloor$. In the case of (iii), even if the push flux executed an inlined update, the update would be a no-op as $S_B.\lfloor t \rfloor = \tau$. Since the push flux function doesn't modify $S_B.u$, (3.14) holds. The call to ACCUMULATE correctly updates $S_B.\ell.\Sigma\hat{F}$ so that (3.16) holds, and UPDATE_K_BDRY ensures that (3.18) and (3.19) are true. For the right side, since no modifications occur to $S_B.u$ or $S_B.r$ and the submesh is assumed to have satisfied *CR* before the push flux, (3.17), (3.20), and (3.21) must hold.

LO: For case (i), the processing of the push flux and inlined update implies S_A and S_B are synchronized $S_B.\ell.\lfloor t \rfloor = S_B.\lfloor t \rfloor = S_B.\ell.t_{\text{sync}}$. Thus, LO_ℓ holds. On the right side, LO_r will be satisfied using the same logic to ensure that LO_r holds when updating a submesh. For case (ii), processing the push flux and not updating, implies that $S_A.r.\lfloor t \rfloor = S_A.r.t_{\text{sync}}$. Since $S_B.\lfloor t \rfloor < \tau$, the consistency invariant implies $S_B.\lfloor t \rfloor = S_B.\ell.t_{\text{sync}}$. Therefore LO_ℓ is true. Since $S_B.\lfloor t \rfloor$ and $S_B.r$ are unchanged LO_r must hold, since it held before the push flux by assumption. Lastly, for case (iii), once the

push flux has been processed at τ, LO_ℓ holds by the same logic used for case (i), and since $S_B.\lfloor t \rfloor$ and $S_B.r$ remain unchanged, LO_r holds following the same logic used in case (ii).

P: For case (i), the push fluxes scheduled in the inlined update satisfy (3.30). For case (ii), since the message was processed and did not lead to an inlined update, $S_B.\lceil t \rceil > \tau$, and an update has been scheduled satisfying (3.28). For case (iii), if after the push flux we are able to schedule an update greater than τ, (3.28) is satisfied. Otherwise, the progress guarantee and the fact that after the push flux the submesh and its left neighbor are synchronized imply that being unable to schedule a future update must arise due to the CFL condition at the right boundary, i.e. $\textsc{compute\_t\_next}(S_B, S_B.r) \leq \tau$. When the submesh updated it must have sent a forced push flux to the right neighbor, satisfying (3.30). If that push flux has already been consumed, this implies that the right neighbor has updated at τ and sent a push flux back to the submesh, satisfying (3.29). Thus, (3.29) or (3.30) must hold.

CI: For case (i), S_B updates, and the push flux sent to S_A as well as the the fact that $S_B.\ell.t_{\text{sync}} = \tau$ implies that (3.23) and (3.25) hold. On the right side, $CI(S_B, S_C, \tau)$ holds following the same logic used during the unforced update. For case (ii), $CI(S_B, S_C, \tau)$ continues to hold since it held before the push flux was executed. Considering $CI(S_B, S_A, \tau)$, (3.22) or (3.23) held before the push flux. Without changes to $S_B.u_A$ or $S_B.\lfloor t \rfloor$, one of the two statements continues to hold after the push flux. Since S_B did not update, S_A could not have scheduled a forced update, and $S_A.r.t_{\text{sync}}$ remains unchanged from the state before S_A last updated. Satisfying the consistency invariant on S_B before any events executed at time τ implies $S_A.r.t_{\text{sync}} = \tilde{S}_B.\ell.t_{\text{sync}}$. Since the synchronization time on S_B also remains unchanged (3.24) must hold. For case (iii), $CI(S_B, S_C, \tau)$ holds following the same logic used for case (ii). On the left interface, $S_B.\ell.t_{\text{sync}} = \tau$

after the push flux. When S_B updated at τ it sent a message to S_A. If this message has not been processed (3.25) holds. Otherwise, S_A has updated and processed the push flux for S_B (in no particular order). This implies that $S_A.r.t_{\text{sync}} = \tau = S_B.\ell.t_{\text{sync}}$. Therefore, (3.24) holds.

It is necessary to check that consuming the push flux does not lead to invariant violations of P or CI on S_A. If S_A is able to schedule a future update or still requires a message from its left neighbor (i.e. not S_B), consuming the push flux has no impact on $P(S_A, \tau)$. However, if S_A is unable to schedule a future update due to required synchronization with S_B, the push flux must require an inlined update on S_B. If S_B updates during the push flux, it will send a message to S_A, thereby satisfying (3.29). If S_B has already updated, there must be an outstanding push flux from S_B to S_A. This event cannot have been consumed yet, since otherwise S_A and S_B would be synchronized on S_A and the progress guarantee implies that this boundary could not hinder the scheduling of a future update. The existing push flux from S_B to S_A at time τ satisfies (3.29). Thus $P(S_A, \tau)$ continues to hold after processing the push flux on S_B. Next, we check that $CI(S_A, S_B, \tau)$ is not violated. Since S_A must have updated at τ to send the push flux and will only update once at τ, we are guaranteed that S_A will not have updated since sending the push flux. Therefore Equation (3.22) holds. If a push flux from S_B to S_A has already been processed, we know that $S_B.\ell.t_{\text{sync}} = \tau = S_A.r.t_{\text{sync}}$, satisfying (3.24). Otherwise, the outstanding push flux from S_B to S_A implies that (3.26) holds. Thus, processing the push flux will not violate I on S_A.

The transitions following the execution of a push flux from the left (or right) neighbor depend on whether or not the submesh updated and whether or not it can make progress on the right side. If the submesh did not update at τ—which is only the case for q_a—and it is able to schedule an update at a time greater than τ, this will transition to q_g. If the mesh previously updated or is updated during the push flux, the state depends on whether or not the mesh is able to make progress on the right hand side, i.e. whether or not b_6 is true.

We note that the value of b_6 remains unchanged for submeshes updated before the push flux. Therefore, states q_c and q_d will transition to q_h, and states q_e and q_f will transition to q_i. Submeshes which execute an inlined update transition depending on the value of b_6. If the submesh has issued a force push flux to its right neighbor to enforce local ordering or enable scheduling a future update, the submesh transitions to q_i. Otherwise, the submesh is able to make progress and the submesh transitions to q_h.

Processing the Push Flux from the Right

Since by assumption the first message arrives from the left, we only need to consider the right push flux applied to states after processing the push flux from the left neighbor, i.e. states q_i-q_f.

CR: The correctness of the algorithm relies on arguments similar to those use for the push flux from the left.

LO: If the push flux did not cause an update, then both sides must satisfy the local ordering constraint. Otherwise, the submesh has synchronized with both of its neighbors and satisfies the local ordering constraint on either side.

P: If no update is required, that implies $S_B.\lceil t \rceil > \tau$. Thus, P is satisfied by (3.28). Otherwise, if the submesh updated at time τ, it will have already processed both neighboring push fluxes. The progress guarantee implies that the simulation will be able to schedule an update at a time greater than τ, satisfying (3.28).

CI: The consistency invariant holds making the same arguments used to show the consistency invariant holds after processing the push flux from the left.

In the case that push fluxes have been processed on both sides, $S_B.\ell.\lfloor t \rfloor = S_B.r.\lfloor t \rfloor = \tau$. Implying both b_3 and b_5 are false. In the case in which no inlined update occurred, $S_B.\lfloor t \rfloor =$

$S_B.\ell.t_{\text{sync}} < \tau = S_B.\ell.\lfloor t \rfloor$. Otherwise, the neighbors are synchronized at time τ. Therefore, $S_B.\ell.\lfloor t \rfloor = S_B.\ell.t_{\text{sync}}$. Either way, b_4 must be true. Arguing similarly for the right side, b_6 must also be true. Finally, since in either case the progress guarantee was able to schedule an update for a time greater than τ, b_2 must be true. The state transition can then be determined by whether or not the submesh updated at τ, if it did not the submesh must have started in state q_g and transitions to q_j. Otherwise, the submesh transitions to q_k.

Remark 3.4.1. We note that this proof imposed no restriction on scheduling order of events with the same timestamp τ. That is to say that neither update nor push flux event with the same timestamp are required to be executed before the other. However, what we have not shown here is that the events commute, i.e. given any state q_a and q_b, any order of execution would result in the same final state along with the same messages being scheduled. Thus, while any ordering of events would provide a total variation solution, we have not shown that it would provide a unique event trace. Taking advantage of this commutativity might have performance benefits and is a topic of future work. However, for our implementation, we have enforced deterministic execution by ordering events which share a timestamp on a submesh. This is achieved by bit shifting timestamps to the left and using the extra trailing bits to break ties in timestamps.

3.5 Implementation Details

By expressing the algorithm as a discrete event simulation, we can use existing parallelization infrastructure to rapidly and efficiently parallelize the proposed local timestepping algorithm. This section introduces Devastator, the parallel discrete event simulator upon which our implementation is based. Additionally, we outline several performance optimizations made to the algorithm to improve the performance as well as load balancing strategies to achieve good compute resource utilization.

3.5.1 Devastator Simulation Framework

Devastator (publication forthcoming) is the parallel discrete event simulation (PDES) framework we have used to implemented this work. As with other PDES frameworks, Devastator expects the simulation to be modeled as a discrete number of logical processes (i.e. actors) producing and consuming timestamped events. Once the logical processes and their event processing behaviors have been defined, Devastator handles the task of progressing and maintaining consistency of the distributed parallel execution. To do this, Devastator employs the TimeWarp algorithm [67] with an asynchronous algorithm for bounding global virtual time (GVT). Devastator was chosen for this work for its emphasis on performance in HPC environments as well as its productive C++14 interface.

PDES frameworks generally fall in one of two categories in how they maintain consistency of the distributed state, these are named optimistic and conservative. Consider the following scenario: a CPU A in the simulation wants to execute event E having timestamp T. How can the CPU be sure that no other event E' with timestamp T' where $T' < T$ is going to be generated by some other CPU B and sent to A? Conservative methods require all CPUs to synchronize heavily to ensure that E is only ever executed after it can be proven that no such E' is possible. Optimistic methods, like TimeWarp, instead execute events optimistically before knowing that it is safe to so. To deal with the inevitable causality violations (discovering E' only after executing E), TimeWarp performs a *rollback* to revert the logical process's state to before execution of E, then executes E', then E, and carries on. Once GVT passes T, the CPU knows that no such further events E' can occur, and the event E is *committed*.

Optimistic execution enjoys the ability to operate with high performance in regimes of low communication, dynamically, simply by virtue that it always assumes no communication is needed before executing the next event. This makes it an excellent choice for domains

where absolute bounds on the needed inter-CPU synchronization are far tighter than what is required in the average case. Simply put, for an application that rarely needs to communicate, it's best to assume it never needs to and then only pay a heavy cost when that assumption fails, rather than synchronizing frequently (as in a conservative execution) only to learn communication isn't required.

In the case of nonlinear conservation laws, determining whether an event is able to execute without incurring a CFL violation requires determining the domain of dependence for that submesh. Due to pathological examples such as the shockwave for Burgers' equation considered in the introduction, this is an inherently non-local problem. In situ computation of the domain of dependence would greatly increase required communication between submeshes. The locality of this problem can be limited by considering smaller timesteps at the cost of available parallelism. However, in practice we assume that the probability of a remote high-speed wave dramatically increasing $|\Lambda|$ and causing CFL violations is very small. Using optimistic execution, we are able to maintain our local communication stencil without limiting parallelism, and only incur significant overhead when timesteps require refining.

3.5.2 Performance related optimizations

While implementing the above presented push flux and update functions specify a TVD timestepping algorithm, three performance optimizations were necessary to obtain good parallel performance.

1. *Timestep binning:* While the COMPUTE_T_NEXT allows us to take optimally large timesteps, the local ordering constraint makes this approach suboptimal. As an example consider two neighboring submeshes, which are able to take respective timesteps of 16 (the left submesh) and 17 (the right submesh) time units. During the simulation, the left submesh advances to time 16, and the right submesh advances to time 17. Once the right submesh has advanced, it forces its neighbor to update due to a local

ordering violation, and so the left submesh has updated at both times 16 and 17. Using a work-depth analysis, at time 272, this approach requires $3 \cdot 16$ (48) updates, and has a depth of $2 \cdot 16$ (32). Instead, we propose binning timesteps to the nearest power of two. Specifically, we require that given a certain timestep size (Δt), we step to largest multiple of the largest power of two multiplied by $\Delta t_{\min}$ less than the given timestep. This naturally synchronizes submeshes and avoids extra updates due to local ordering violations. In the given example, both submeshes would take timesteps of 16. Thus, the work would be $17 \cdot 2$ (34) and the depth would be 17. By restricting allowable timestep sizes, we potentially reduce both work and the depth of the simulation.

2. *Reducing unnecessary speculation:* While TimeWarp allows us to speculatively update the submeshes, it is not always wise to do so. If a submesh updates and sends a forced push flux to one neighbor, we know that at that time the neighbor would have to send a push flux back. Therefore, any events executed on the submesh before the neighboring push flux arrives must be rolled back. Therefore, whenever scheduling new updates the submesh inspects its state to determine whether it is still waiting for a message from one of its neighbors. If so, it will simply return without scheduling any further events. Once the messages the submesh is waiting on arrive, it will schedule the next events without having to roll back events.

3. *Avoiding small timesteps due to binning:* While timestep binning reduces the number of synchronizations required due to local ordering violations, it introduces another problem due to the fact that timesteps at submesh boundaries are computed relative to the previous synchronization time. Considering two submeshes that both could step at 7 time units. The timestep binning would make both of these submeshes update at time 4. However, assuming that the push flux from one submesh is delayed, the other submesh would take timesteps of sizes 2 and 1 before being unable to make progress

and sending a forced update to the neighbor. Since the timestep is taken relative to the previous synchronization time, the submesh will try to enumerate the bits of its maximum timestep before waiting on a message from its neighbor. This phenomena is highly problematic for parallel discrete event simulation, since once the neighboring message at time 4 arrives, the updates at times 6 (timestep of 2) and 7 (timestep of 1) would have to be rolled back along with any events scheduled due to those later updates. Furthermore, since the timesteps become smaller and smaller, the associated updates are enqueued with a high priority into the event queue and the simulator is more likely to spend time executing events which will be rolled back. To remedy this, we introduce a heuristic whereby we examine the ratio between the timestep taken if the submesh were synchronized with its neighbors divided by the computed timestep due to a single neighbor, i.e. the value of COMPUTE_T_NEXT_BDRY. If this ratio is greater than or equal to 2, we force that neighbor to update at the current time. The previous optimization (reducing unnecessary speculation) then causes the submesh to wait until the neighbor's push flux has been processed. While this may increase the critical path of simulation depending on the latency associated with sending messages, we have found that in practice this significantly reduces bad speculation.

3.5.3 Performance Modeling and Load Balancing

To understand the performance results as well as load balance the problem, we estimate the work for a given problem as follows. At a given simulation time τ, the timestep dt taken by cell j can be approximated as

$$dt_j(\tau) = \Delta t_{\min} 2^{\left\lfloor \log_2 \frac{\Delta x_j}{2|\Lambda(\tau, x_j)|\Delta t_{\min}} \right\rfloor},$$

where $\Delta t_{\min}$ is the smallest timestep, Δx_j is the cell size, and $|\Lambda|$ corresponds to the wave speed at time τ in the midpoint of the cell. The $\log_2$ appropriately bins the timesteps.

The mesh partitioning problem follows a 2-phase process of aggregating cells into submeshes and assigning the submeshes to ranks. Cells within each submesh step synchronously enabling efficient utilization of CPU architectures. Although stability requirements necessitate that cells step at the most stringent timestep of cells associated with that submesh, unstructured finite element meshes typically exhibit small variations of timestep sizes in a given neighborhood. Therefore, the synchronous stepping inside a submesh marginally decreases the maximum attainable speed-up.

The mapping of cells to submeshes is denoted by $\pi : \mathbb{Z} \to \mathbb{Z}$. Since in this chapter, we exclusively consider one dimensional problems, we define our partition using a set of ordered splitters $\{s_i\} \subset \mathbb{Z}$ where the submesh i has been assigned the cells with indices $[s_i, s_{i+1})$. The relationship between the splitters and partition function is then given by: for cell j, $\pi(j) = i^*$ such that $s_{i^*} \leq j < s_{i^*+1}$. For the generation of π, we assume no prior knowledge on Λ i.e. $|\Lambda| \equiv 1$, and partition solely based on variation in cell sizes. Let w_j be the weights assigned to each cell. Each w_j is determined by the smallest allowable timestep on a given submesh, i.e.

$$w_j = \frac{1}{\min_{1 \leq r \leq n_{el}} \{dt_r(\tau) \,:\, \pi(j) = \pi(r)\}}.$$

Since the partitioner balances work across submeshes, but the work depends on the partitioning, there exists a circular dependency. Hence, we generate π using an iterative procedure. Given weights $\{w_j\}$, we create a partition π, ensuring each submesh has the same amount of work. With the new partition, update the weights w_j, which may change based on elements added or removed from a submesh. Repeat this process until some terminating condition is satisfied. In this thesis, we stop iterating after 100 iterations. Throughout the iterations, we track the amount of work assigned to the most overworked submesh. At the end of the

iterative procedure, we return the partition with the least overworked submesh, i.e. if π_ℓ is the ℓ-th iteration of the submesh partitioner,

$$\pi = \underset{\{\pi_\ell\}}{\operatorname{argmin}} \max_{0<i<n_{\text{sbmsh}}} \sum_{j=1}^{n_{el}} \chi_{\{\pi_\ell(j)=i\}}(j) w_j$$

where χ_A denotes an indicator function over set A. The objective of the first partitioning phase is to specify a problem that can be efficiently executed by the parallel discrete event simulator. The proposed scheme attempts to generate submeshes such that the number of cells updated is approximately equal across all submeshes. However, due to the dependence of work on partitioning, we must also pay attention to the total work. A proposed partitioning may be perfectly load balanced but require a lot more work and therefore be less preferable than a partitioning which is imbalanced but more work optimal, i.e. for which the discrepancy between w_j and dt_j^{-1} is smaller.

The second partitioning phase assigns submeshes to ranks. The objective of this phase is to minimize the runtime of the simulation. Let ρ map a submesh to its assigned rank. Since the most overworked rank will determine the rate at which global virtual time is advanced, we estimate the wall-clock time as

$$T = \int_0^{t_{\text{end}}} \max_{0\leq k<n_{ranks}} \sum_{j=1}^{n_{el}} w_j(\tau) \chi_{\{(\rho\circ\pi)(j)=k\}}(j)\, \mathrm{d}\tau. \tag{3.33}$$

We remark that for performance results presented in Section 3.6.3, we consider analytic solutions, thereby yielding a good approximation to $|\Lambda|$ and hence w_j. We solve (3.33) using a Gauss-Lobatto quadrature to approximate the integral in time. Given the submesh partition π and our assumed wavespeed Λ, we can formulate this problem as mixed integer programming problem, which we solve using Gurobi [53]. The incorporation of $|\Lambda|$ in (3.33) introduces a discrepancy between the $|\Lambda|$ used for the two partitioning phases. The decision

to use different $|\Lambda|$ for the partitioning phases arises from two considerations. First, exactly knowing $|\Lambda|$ is an impractical constraint. Since $|\Lambda|$ is a function of the solution, knowing $|\Lambda|$ implies already knowing the solution to the problem we are interested in solving. Using approximations or even analytic representations of $|\Lambda|$ for determining π can be problematic. If we consider one giant submesh and a single cell underestimates $|\Lambda|$, the entire submesh will incur extra work. By ignoring $|\Lambda|$ in during the computation of π, we inoculate ourselves against adverse effects caused by poor estimates of $|\Lambda|$. The second consideration has to do with partitioner performance. The number of cells is 2-3 orders of magnitude larger than the number of submeshes. Attempting to solve (3.33) for the cell graph would be intractable for large cell counts. By ignoring $|\Lambda|$ in the first partitioning phase, we can use existing fast graph partitioners to generate submeshes and use more sophisticated means to achieve good load balance at the submesh level.

Lastly, we derive upper limits on the speed-up achievable as the ratio of work (i.e. number of cell updates) executed by a standard synchronous timestepping implementation using an MPI runtime divided by the local timestepping Devastator implementation. We calculate the total work for the Devastator runtime as

$$W_{deva}^{th} = \sum_{j=1}^{n_{el}} \int_0^{t_{\text{end}}} w_j(\tau)\, \mathrm{d}\tau.$$

Note that this estimate is a conservative work estimate for the Devastator implementation as it doesn't account for factors such as taking intermediate timesteps between finer and coarser timesteps as well as extra timesteps due to forced updates and rollbacks. For an MPI-based implementation, which steps with uniform timesteps, we compute the work W_{MPI}^{th} as the number of timesteps times the number of elements. We bound the largest theoretical speed-up by

$$S^{th} = \frac{W_{MPI}^{th}}{W_{deva}^{th}}.$$

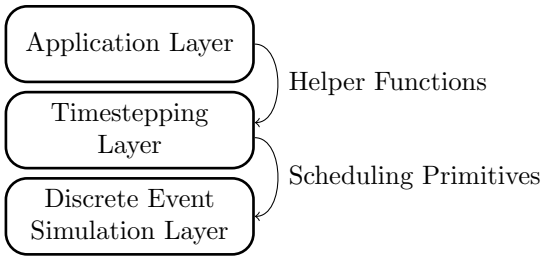
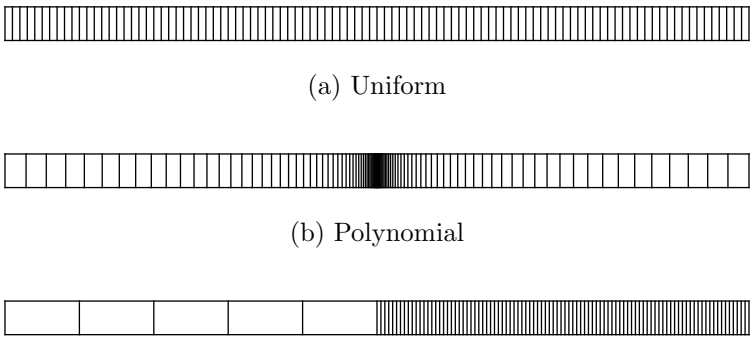


Figure 3.8: Stack Diagram for the Local Timestepping Implementation

This will provide a baseline to assess how efficient the implementation is.

3.5.4 Ease of Implementation

We conclude this section by remarking on the complexity of the implementation. The proposed algorithm can be segmented into three main software layers shown in Figure 3.8. At the lowest level is the discrete event simulation layer for which we are using Devastator. Devastator handles all parallelism via the scheduling primitives (Definition 3.4.2). The layers on top of it schedule events and Devastator ensures that the events are executed in the correct order in a parallel setting. On top of the Devastator layer is the timestepping layer. This layer contains the timestepping logic introduced in Section 3.4. The complexity of running in an asynchronous context makes this layer difficult to reason about without a tool like loop invariants. However, through judicious specification of helper functions (Definition 3.4.1) we can prevent the complexity of the timestepping algorithm from seeping into the topmost layer, the application layer. In practice, the timestepping algorithm should be wrapped into a library, and the user would solely need to implement the helper functions. At this highest level, features of the algorithm such as numerical discretization and the form of the conservation laws are specified. Programming at this highest level is effectively identical to programming flat MPI code. The ADVANCE function is essentially the main kernel of an MPI rank. The only additional calls that need to be supplied deal with assessing the appropriate timestepping size and what to do once a submesh update is either committed or rolled back.



(a) Uniform

(b) Polynomial

(c) Piecewise

Figure 3.9: Meshes used for numerical experiments

This segmentation facilitates the introduction of local timestepping into existing scientific applications without requiring extensive rewriting of code or impeding productive software development in the application layer.

3.6 Results

In this section we present results for the one dimensional Burgers' equation and the shallow water equations. Since the timestepping method is first order, we only consider first order finite volume schemes. To demonstrate the robustness of the timestepping method for different types of meshes, we consider three meshes: a uniform mesh, a polynomial mesh, and a piecewise mesh. These meshes are generated by warping uniformly distributed nodes along $(-1, 1)$ onto the interval $(-1, 1)$. The base case is the uniform mesh for which the warp function is the identity, i.e. $w(x) = x$. The polynomial mesh refines the mesh around the origin. This type of mesh is common for finite element applications where local refinement is required to resolve flow around fine features. The warp function for this mesh is given as

$$w(x) = \frac{1}{1/3 + \varepsilon} \left(\frac{x^3}{3} + \varepsilon x \right),$$

where we set $\varepsilon = 0.02$ in order the bound the ratio of largest to smallest cells. Lastly, we consider a mesh with a large jump in refinement. The warp function is then defined so that the ratios of cell sizes is at least 16 to 1. Assuming the nodes are enumerated x_j for $0 \leq j \leq n_{el}$, define $j^* = \lfloor n_{nodes}/17 \rfloor$. We then define the warp function for the piecewise mesh as

$$w(x) = \begin{cases} \frac{x+1}{1+x_{j^*}} - 1 & \text{for } j \leq j^* \\ \frac{x-1}{1-x_{j^*}} + 1 & \text{for } j > j^* \end{cases}.$$

For clarity the meshes used are depicted in Figure 3.9. To illustrate the behavior of the timestepping algorithm, we consider meshes with 100 cells and 20 submeshes in the next two sections. In practice, the number of cells per submesh needs to be significantly larger to amortize runtime overheads with useful work. Section 3.6.3 showcases performance results with meshes consisting of 500,000 cells.

3.6.1 Burgers' Equation

Consider Burgers' equation on the line,

$$\partial_t u + \partial_x u^2/2 = 0. \tag{3.34}$$

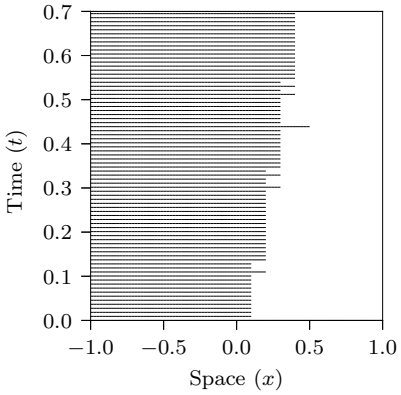
We consider two sets of initial conditions: firstly, the shockwave, which is initialized

$$u_0(x) = \begin{cases} 1 & x < 0 \\ 0 & x > 0, \end{cases}$$

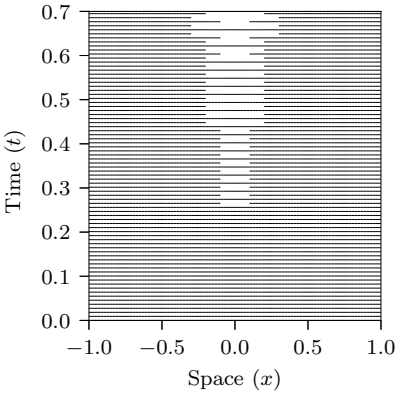
and secondly a rarefaction wave,

$$u_0(x) = \begin{cases} -1 & x < 0 \\ \\ 1 & x > 0. \end{cases}$$

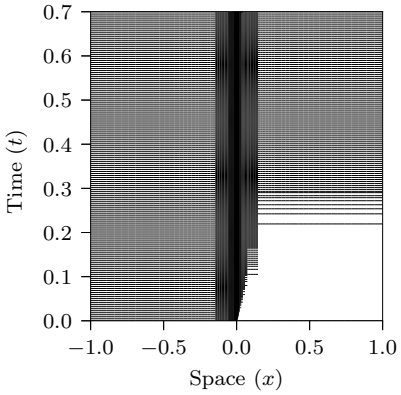
For both cases, we enforce the boundary conditions by setting the boundary value to the analytic solution. We note that these test cases highlight the ability of the local timestepping algorithm to refine the timestep—in the case of the shockwave—and the ability to coarsen the timestep—in the case of the rarefaction. To visualize the timesteps taken by these algorithms, we present space-time event traces similar to those shown in Figure 3.2. In these plots, the x-axis corresponds to the domain Ω, and the y-axis represents simulation time, each line corresponds to an update having executed at that given time. The space-time plots are shown in Figure 3.10. For the shockwave and rarefaction problems on the uniform polynomial mesh, the L^1- and L^2-errors are bounded by 0.055. For the piecewise mesh, the errors are bounded by 0.15. Nevertheless, results for the piecewise mesh remain stable and show that the under resolved region is able to step with much larger timesteps. Looking at Figures 3.10a and 3.10e, we see that submeshes only update behind the shock front. For the shockwave initial conditions on the polynomial mesh—shown in Figure 3.10c—after $t = 0.2$ the entire mesh begins stepping. This is due to the fact that cells in the interval $(0.146, 1)$ belong to a single submesh. Since these cells take larger timesteps the submesh partitioner places more cells into this submesh. In later results with more cells and submeshes, the inactive regions of the polynomial mesh do not update. For the rarefaction problem, due to timestep binning, timestep coarsening happens only once the submesh is able to take a timestep twice as large as the previous timestep. For the analytic solution to the rarefaction wave at $t = 0.7$–the end of the simulation—$|u| < 0.5$ for $|x| < 0.35$. This is consistent with the observed timestep coarsening seen in Figure 3.10b. Similarly, the polynomial and



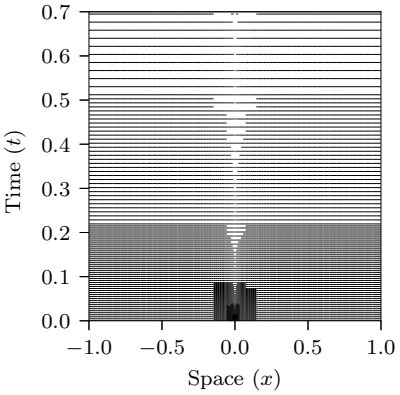
(a) Uniform-Shockwave



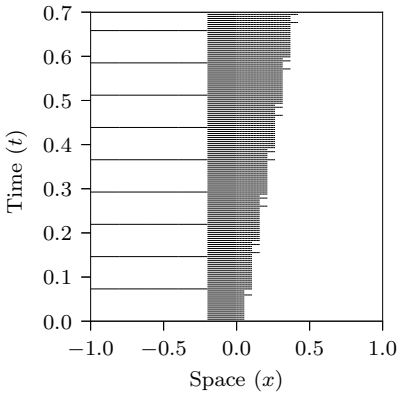
(b) Uniform-Rarefaction



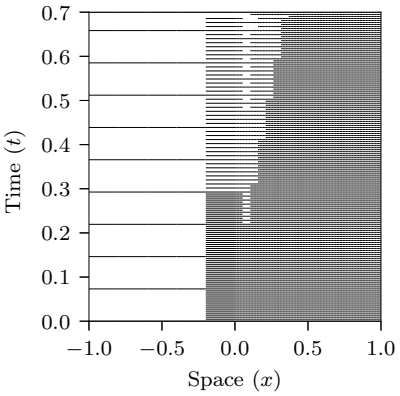
(c) Polynomial-Shockwave



(d) Polynomial-Rarefaction



(e) Piecewise-Shockwave



(f) Piecewise-Rarefaction

Figure 3.10: Space-time plots for Burgers' equation with various meshes and problem configurations.

piecewise meshes are able to adapt their timesteps appropriately. In these cases, the submesh partitions are not symmetric about $x = 0$, giving rise to the asymmetry in the event traces.

3.6.2 Shallow Water Equations

In order to show that the local timestepping scheme is robust for more non-linear problems and in the absence of the theoretical guarantees, we provide two problems for the shallow water equations. Consider the system of conservation laws,

$$\begin{cases} \partial_t h + \partial_x q_x = 0 \\ \partial_t q_x + \partial_x (hu^2 + gh^2/2) = -gh\partial_x z \end{cases}$$

where $q_x = hu$, z is the bathymetry, and $g = 1$. We consider here the dam break Riemann problem, for which,

$$h_0(x) = \begin{cases} 1 & \text{if } x < 0, \\ 1/16.1 & \text{if } x > 0, \end{cases}$$
$$q_{x,0}(x) = 0,$$
$$z = 0.$$

For the shallow water equations the maximum advection speed, $|\Lambda|$ is $\sqrt{gh} + |u|$. The initial conditions have been chosen to allow a 4-to-1 timestepping ratio between downstream and upstream of the dam break for the uniform mesh. The second shallow water test case we consider is the analytic problem of Carrier and Greenspan [23]. This problem considers water moving up and down a shoreline with uniform slope in a periodic manner. We follow the set-up outlined in [12] with a phase shift of $\varphi = -\pi$.

For the numerical discretization, we use a local Lax-Friedrichs flux along with the first-

order local hydrostatic reconstruction proposed in [1]. The space time plots are shown in Figure 3.11. We note that for the shallow water flow, the theoretical guarantees no longer hold. In fact, the timestepping region between the two waves emanating from the dam break problem in Figure 3.11a requires a finer timestep than observed during the beginning of the simulation, and thereby violates the progress guarantee, Lemma 3.4.1. For the dam break problem, the refinement of the timesteps looks similar to that of the shockwave problem for Burgers' equation. For the uniform and piecewise meshes, we see expected refining of timestep sizes, and for the polynomial mesh, regions far from the shock waves prematurely begin taking fine timesteps due to the large submeshes generated by the submesh partitioner.

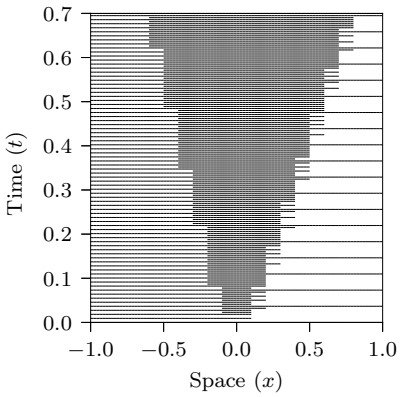
For the Carrier-Greenspan problem on the uniform mesh, shown in Figure 3.11b, we see the wave moving in and out of the domain. Analytically, the water front never goes past $x = 0.25$, and the simulated event trace reflects this as no submesh updates occur for in regions for which $x > 0.3$. For the polynomial mesh in Figure 3.11d, we again begin stepping everywhere due to the submesh graph partitioning. For both of these cases, we note hysteretic effects in regions which are mesh drying. The final simulation time of 2π contains two periods of the Carrier-Greenspan solution. At time π, the submeshes participating in updates would ideally look identical to the start of the simulation, i.e. only submeshes which contain cells located at $x < -0.25$ would update. Rather we see this slow "draining" of mass as submeshes try to coarsen their timesteps, resulting in updates occurring in regions of the mesh that are dry in the analytic solution. The impact of behavior on performance will be touched upon in the next section. For the piecewise mesh, the incoming wave is so under resolved that it is unable to cause significant wetting and drying on the beach. Hence, this configuration fails to reproduce the periodic wetting and drying event trace seen for the uniform and polynomial meshes.

To conclude, the proposed timestepping method remains stable for simulations with dramatic temporal variations in Λ as shown for Burgers' equation in Figure 3.10 and the shallow

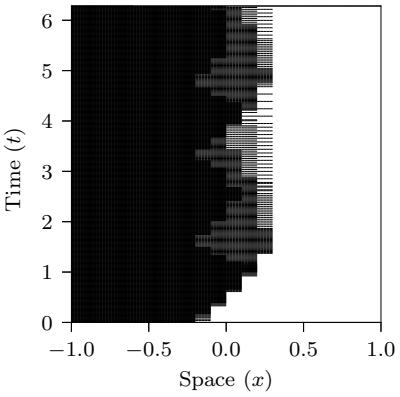
water equations in Figure 3.11. For the latter problem, we fail to satisfy the assumptions for the proof of correctness and even observe a violation of the progress guarantee, and yet the algorithm is able to stably compute the correct solution. Important in both problems is the ability for the algorithm to locally refine or coarsen timesteps. The proposed formulation dispenses with complicated book keeping relating to which timestepping level submeshes are in and how to appropriately define buffer zones, but simply computes and adjusts the timestep based on locally available information.

3.6.3 Performance Comparison

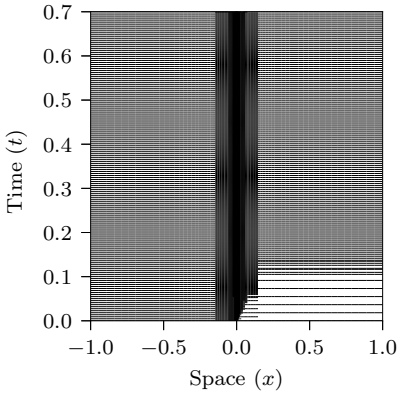
In this section, we compare the performance of our local timestepping implementation to a flat MPI implementation. For the MPI implementation, we use non-blocking point to point messaging and hide message latencies with internal work. For more details, we refer the reader to implementation in [15]. One key detail is that the MPI implementation does not compute the CFL condition, but rather uses a fixed timestep. Since dynamically updating a CFL condition would require an all-reduce at each timestep, the communication overhead makes an adaptive CFL condition non-viable for large-scale simulation. For the performance comparison, we consider the uniform and polynomial mesh with 500,000 cells on one Skylake node with 48 cores on TACC's Stampede2. We partition the mesh into 48 (one per core) uniform submeshes for the MPI implementation, and 288 (six per core) submeshes for the Devastator implementation. We partition the mesh into 48 (one per core) uniform submeshes for the MPI implementation, and 288 (six per core) submeshes for the Devastator implementation based on the heuristic described in Section 3.5.3. The over-decomposition factors of submeshes to ranks have been optimized for each runtime. Over-decomposition of the mesh for Devastator enables three performance optimizations. Firstly, over-decomposition allows a rank to hide message latencies. While one actor may be waiting on a message to arrive, the rank may execute events scheduled for other actors. Secondly, over-decomposition reduces



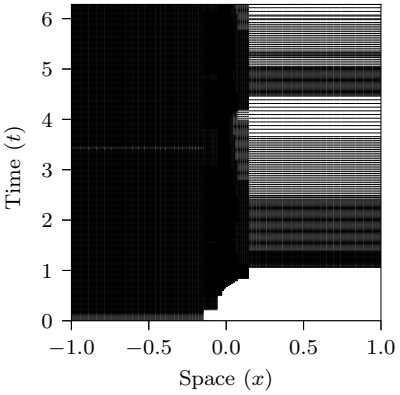
(a) Uniform-Dam break



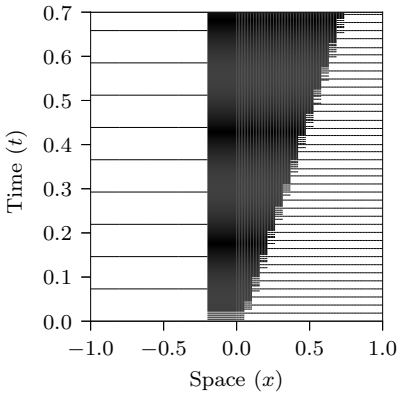
(b) Uniform-Carrier-Greenspan



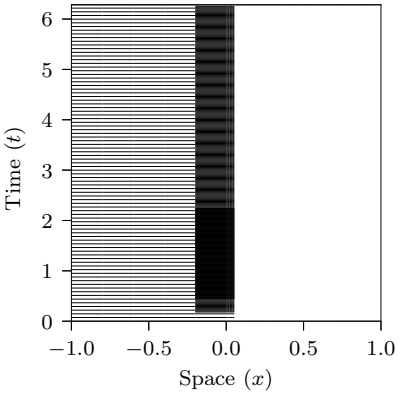
(c) Polynomial-Dam break



(d) Polynomial-Carrier-Greenspan



(e) Piecewise-Dam break



(f) Piecewise-Carrier-Greenspan

Figure 3.11: Space-time plots for the shallow water equations with various meshes and problem configurations.

the total amount of work required. Since each cell must step synchronously with other cells on that submesh, creating more submeshes implies that more cells are able to step with different timesteps. Lastly, more submeshes improves the performance of the load balancer. With more submeshes, the load balancing algorithm is able to more easily balance the simulation throughout all time points in the simulation. However, over-decomposition comes at the cost of higher runtime overhead. More events with less work implies that a larger fraction of event execution time is spent on scheduling and control flow. The over-decomposition factor of 6 was chosen based on observed performance. The MPI implementation does not benefit from over-decomposition. We explicitly hide message latencies by performing all internal work after posting sends and receives. Furthermore, the total number of cell updates—and thus work—remains independent of the over-decomposition factor and the load balancer is able to balance work across MPI ranks easily.

To analyze the impact of the dynamic CFL condition versus the local timestepping due to mesh refinement, we consider two problems for the shallow water equations. Firstly, we consider a lake at rest problem, where

$$\begin{aligned} h_0(x) &= 1, \\ q_{x,0}(x) &= 0, \\ z &= 0. \end{aligned}$$

For this problem, differences in the CFL condition are solely a function of differences in cell size, Δx. The second problem we consider is the Carrier-Greenspan solution, outlined in the previous section. As the water front moves across the mesh, large portions of the mesh wet and dry throughout the simulation, causing these regions to experience large changes in allowable timestep sizes. For the lake at rest problem, the work in each submesh is approximately the same. Thus, we assign submeshes $[6 \cdot k, 6 \cdot (k+1))$ to rank k. In the case

of the Carrier-Greenspan problem, due to the periodic nature of $|\Lambda|$, we load balance over a quarter of the period, i.e. $t_{\text{end}} = \pi/2$ in (3.33). For simulations of the Carrier-Greenspan problem on either mesh, we run Gurobi for 6 minutes and then use the best solution found. There are significant variations in the difficulty of the load balancing the problem. For the uniform mesh, the partitioner returns with a solution with an imbalance of 4%. For the solution to the polynomial mesh, Gurobi returns a solution with an imbalance of 25%. However, Gurobi also reports that a lower bound on the best possible partitioning is 20% indicating that the solution is within 5% of the optimal partitioning.

The performance data is shown in Table 3.2. For the static uniform Lake at Rest problem, we include a Devastator configuration—labeled as (no CFL)—without the overhead of computing the local CFL in order to provide a more direct comparison with the MPI version, which does not compute the CFL in any configuration. Each configuration is run 10 times with the average time elapsed, $\bar{T}$ reported in seconds. The standard deviation over the mean $\sigma_T/\bar{T}$ is given in percentages. Lastly, the number of updates and rollbacks are taken from a single run.

We begin by noting that the number of cells rolled back is limited. At most 3.5% of updates are rolled back suggesting that the performance related optimizations do a good job of preventing unnecessary rollback. The lack of rolled back updates for the uniform mesh lake at rest problem shows that we are able to completely avoid bad speculation for this configuration. For the lake at rest problem on the polynomial mesh, it should be possible to infer when messages are incoming since $|\Lambda|$ and Δx are fixed. The existence of rollbacks implies that there is still future work to be done to prevent bad speculation.

The speed-up achieved via local timestepping is attained through work reduction. To assess the quality of our implementation, we contextualize execution times with configuration-dependent metrics to determine what fraction of unnecessary work was skipped. Recall the theoretical speed-up S^{th}, which estimates the maximum achievable speed-up based on ana-

Table 3.2: Execution times for the shallow water equations on Stampede2.

Problem	Mesh	Runtime	$\bar{T}$	$\sigma_T/\bar{T}$	# of Updates	# of Rollbacks
Lake at Rest	Uniform	Devastator (no CFL)	139.4	0.8%	$252.5 \cdot 10^9$	0
Lake at Rest	Uniform	Devastator	173.2	0.8%	$252.5 \cdot 10^9$	0
Lake at Rest	Uniform	MPI	133.2	0.1%	$252.5 \cdot 10^9$	–
Lake at Rest	Polynomial	Devastator	192.6	3.3%	$239.7 \cdot 10^9$	$3.5 \cdot 10^9$
Lake at Rest	Polynomial	MPI	478.4	2.0%	$892.2 \cdot 10^9$	–
Carrier-Greenspan	Uniform	Devastator	729.4	1.1%	$856.2 \cdot 10^9$	$12.0 \cdot 10^9$
Carrier-Greenspan	Uniform	MPI	995.7	0.7%	$1674.5 \cdot 10^9$	–
Carrier-Greenspan	Polynomial	Devastator	3212.4	0.5%	$3209.6 \cdot 10^9$	$113.3 \cdot 10^9$
Carrier-Greenspan	Polynomial	MPI	10732.0	0.1%	$18070.1 \cdot 10^9$	–

Table 3.3: Theoretical versus observed speed-ups.

lytic values of Λ and Δx. We compare this theoretical estimate against a speed-up based on the number of cells updated. Let W^{obs}_{MPI} and W^{obs}_{deva} be the number of cells updated during the simulation with respective runtimes, and define the work-based speed-up as S^{work} the ratio of observed updates, i.e.

$$S^{work} = \frac{W^{obs}_{MPI}}{W^{obs}_{deva}}.$$

Discrepancies in S^{work} and S^{th} arise due to the enforcing of the local ordering constraint as well as extra updates arising from a combination of timestep binning and variations in Λ.

Lastly, we introduce the observed speed-up, which is based on the ratio of run times,

$$S^{obs} = \frac{\bar{T}_{MPI}}{\bar{T}_{deva}},$$

where $\bar{T}_*$ corresponds to the mean observed execution time for either runtime. While the work based speed-up S^{work} represents an upper bound on achievable speed-up, the observed speed-up is able to take into consideration factors such as imbalance, rollbacks, and algorithmic overhead.

Table 3.3 presents the three speed-up values for each configuration. The discrepancies between S^{th} and S^{work} are less than 3% with the exception of the Carrier-Greenspan poly-

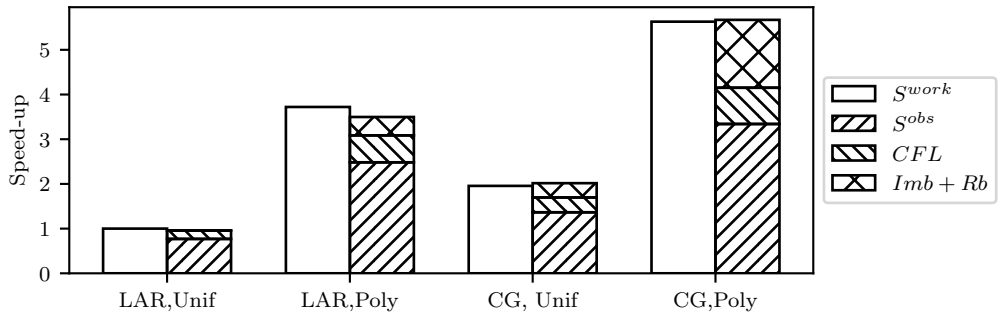


Figure 3.12: Comparison of speed-up due to work reduction S^{work} against observed speed-up S^{obs} and the impact of avoiding CFL computations (CFL) and fixing load imbalance and rollbacks ($Imb + Rb$) for the lake at rest (LAR) and Carrier-Greenspan (CG) problem on the uniform (Unif) and polynomial meshes (Poly).

nomial configuration where we over-predict the attainable speed-up by 17%. In particular, we suspect these discrepancies result due to difficulties in water draining out of dry regions akin to the updates seen in Figures 3.11b and 3.11d.

The observed speed-up ranges from 59% to 77% of the work-based theoretical speed-up, S^{work}. Since there is no theoretical benefit from using adaptive, local timestepping in the lake at rest on a uniform mesh configuration, this case highlights the overhead associated with using an adaptive, local timestepping scheme versus a static synchronous timestepping scheme. If we skip the CFL computation and use the analytic value of K^{int} (see the first row of Table 3.2), the observed speed-up is $S^{obs} = 0.96$.

To distinguish between algorithmic overhead of computing the CFL condition and other performance issues, we account for the cost of the CFL condition by multiplying the MPI execution times by the ratio of the execution times between the CFL and the no-CFL configurations of the lake at rest problem on the uniform mesh using the Devastator runtime. With these corrections, we obtain 82.9% of S^{work} for the lake at rest problem with the polynomial mesh, 86.7% of S^{work} of the Carrier-Greenspan problem with the uniform mesh, and 73.7% of S^{work} of the Carrier-Greenspan problem for the polynomial mesh. Furthermore, the impact of load imbalance and rollbacks is approximated by considering the average imbalance

during the simulation. The imbalance is computed as the ratio of cells updated (both rolled back and committed) on the most overworked rank divided by the average number of committed cell updates across all ranks. Over a sufficiently small time interval this imbalance ratio approximates the improvement obtained through perfect load balancing. We estimate the impact of load imbalance by considering average imbalance over 100 evenly sized time intervals.

The cumulative performance impacts of the CFL computation and load imbalance are shown in Figure 3.12. This graph compares the speed-up due to work reduction, S^{work} against the observed speed-up S^{obs} and illustrates how the CFL computation and load imbalance account for the discrepancy. It is important to note that a given part of the stacked bar graph multiplies factors below it, e.g. the top of the $Imb + Rb$ bar corresponds to the expected speed-up if the simulation were perfectly balanced *and* there was no cost associated with the CFL condition. This distorts the size of bars towards the top of the graph, but allows us to compare the magnitudes of the speed-ups across problem configurations. With these two factors, we are able to account for the discrepancy in expected S^{work} and observed S^{obs} speed-ups within 6.0%.

We also observed that rollback had a limited impact on performance for our experimental configurations. For the Carrier-Greenspan problem, rollback minimally impacted the imbalance of the simulation with the imbalance metric increasing by no more than 0.02 for either mesh. For the polynomial lake at rest problem, the imbalance of committed updates was only 1.02. However, once rollbacks are taken into consideration the imbalance went up to 1.13. Detailed discussion of rollback is presented in Section 3.6.4.

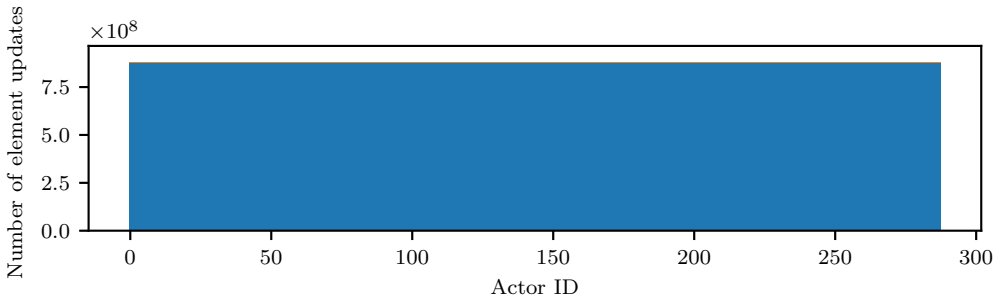
Another cause of the performance degradation due to imbalance may be limitations of static load balancing. Firstly, there exist discrepancies between the performance model outlined in Section 3.5.3 and the observed work done by each submesh, as evidenced by discrepancies between S^{th} and S^{work}. Since the load balancing is based on the work of

the theoretical model, this may lead to load imbalances, which we are not accounting for. Furthermore, the lower bound for the imbalance of the Carrier-Greenspan problem at the end of the Gurobi partitioning was 20%. Although this lower bound does not account for the discrepancy between the performance model and observed work as well as error due to the quadrature used to approximate the integral in (3.33), it suggests that there are underlying limits to how well the partitioner is able to statically load balance the problem. In that case, further reduction in load imbalance would benefit from dynamic load balancing techniques outlined in [16].

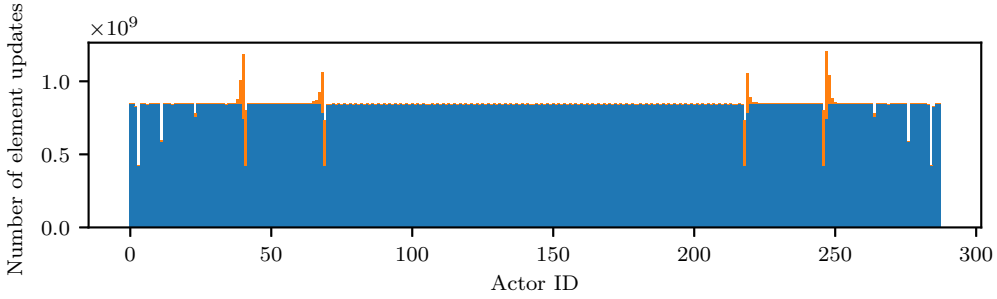
Lastly, we note that while the overhead associated with computing the CFL condition may seem high, it is worth noting that the MPI implementation is not provably stable. For the results presented here, we are able to take optimal timestep sizes, because we are able to analytically determine $(\Lambda/\delta x)_{\max}$. In practice, this value is unobtainable, and the selected Δt would either lead to sub-optimal timestepping, i.e. taking timesteps smaller than necessary or manifest instabilities. The local timestepping algorithm can set an arbitrarily small $\Delta t_{\min}$, and then take appropriately sized timesteps with step-size reductions taken as needed to guarantee stability.

3.6.4 Description of Misspeculation

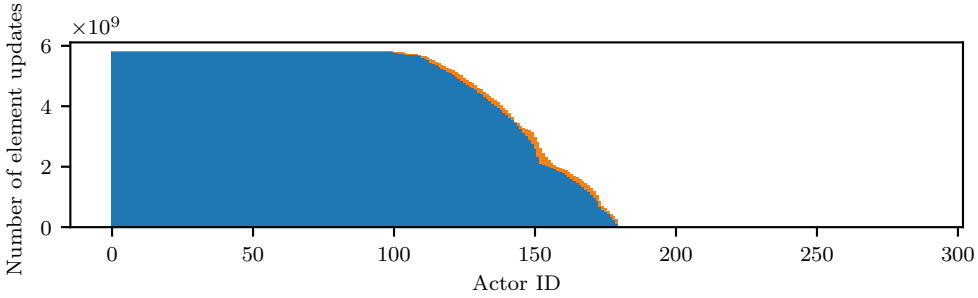
Since we are able to make updates arbitrarily expensive through the addition of cells to a submesh in order to hide task overheads, we only look at rollbacks of non-trivial updates. Events such as push fluxes or updates for which $\lceil t \rceil$ does not match the current simulation time, require little work and rolling them back incurs relatively little overhead. To help understand the behavior of rollback for the simulation, we plot the total number of elements updated and rolled-back per *actor* in Figure 3.13. Since all considered problems are one dimensional, the actors are ordered to reflect their positions in the mesh, i.e. actor 0 updates the leftmost submesh, and actor 1 corresponds to the submesh immediately to the right of



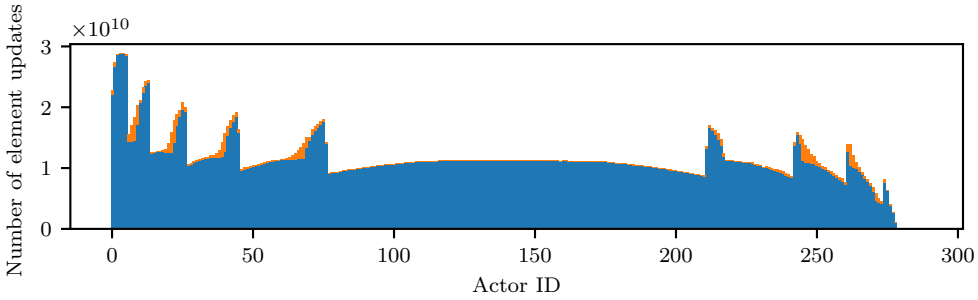
(a) Lake at Rest–Uniform



(b) Lake at Rest–Polynomial



(c) Carrier-Greenspan–Uniform



(d) Carrier-Greenspan–Polynomial

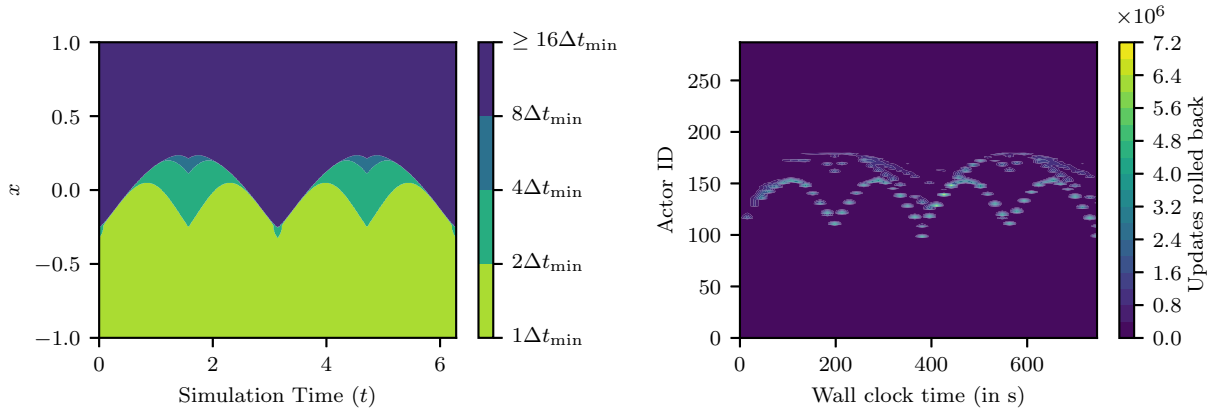
Figure 3.13: Cumulative number of element updates committed—shown in ■—and rolled back—shown in ■ updates for each actor.

the leftmost submesh, etc. We also plot the commits and rollbacks performed by each *rank* in Figure 3.15. For the lake at rest problem, ranks are assigned 6 contiguous actors. Hence, similar to the actors, the rank ID corresponds to the position relative to other ranks. However, for the Carrier-Greenspan problem, the integer programming problem does not consider submesh locality and any rank may be assigned submeshes from anywhere in the mesh.

For the lake at rest problem on either of the two meshes, shown in Figure 3.13a and 3.13b, we see that the partitioning strategy creates submeshes which require roughly the same amount of work to update. The partitioning into submeshes assumes no knowledge of the wave speed Λ and thus uses $|\Lambda| \equiv 1$. Since this corresponds exactly to the $|\Lambda|$ of the lake at rest problem, we achieve a good distribution of work between actors. In Figure 3.13b, we note that a few submeshes are assigned significantly less work than the mean amount of work. This arises due to the non-uniform increments in work added by a single cell. In the iterative submesh partitioning process, if a cell which is constraining the CFL condition for a submesh is removed from that submesh, the submesh will take a larger timestep, thus lowering the overall amount of work associated with that given submesh. Figure 3.13b shows that near these under worked regions is also where rollback occurs. As seen in Figure 3.15b, the ranks which contain these submeshes are assigned less work yet incur significant rollback. Ultimately this speculation causes these ranks to perform the largest number of element updates and determine the critical path of the simulation.

For the Lake at Rest-Uniform configuration shown in Figure 3.13a, the submesh detects stutter stepping after each update and waits. In this case, no updates are rolled back, and due to the balanced submesh partitioning, the imbalance is negligible.

For the Carrier-Greenspan problem, we observe that not all submeshes are assigned the same amount of total work since the flow field is not constant. For the uniform mesh shown in Figure 3.13c, we note that the number of updates per actor is relatively similar for the left



(a) Predicted Δt

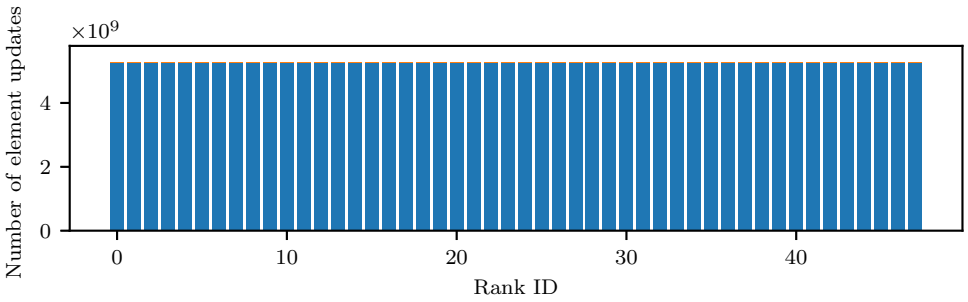
(b) Observed Rollback

Figure 3.14: Temporal occurrence of rollback compared to expected timestep size for the Carrier-Greenspan problem on the uniform mesh.

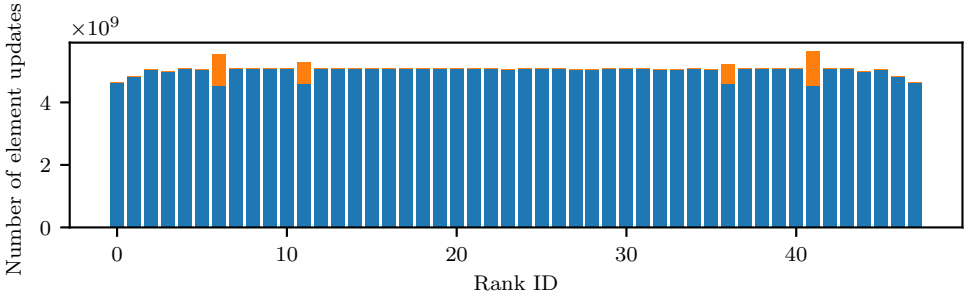
side of the mesh. In the center of the mesh, the cells are wetting and drying as the oscillatory wave passes over them. There is a small amount of rollback occurring, and updates per actor decrease as the water gets shallower. The dry right portion of the mesh does not execute any updates. The Carrier-Greenspan problem on the polynomial mesh heavily distorts the center of submesh. The sharply spiked pattern corresponds to timestep transition regions. The total rollback for both meshes remains low overall, with 1.3% of element updates being rolled back for the uniform mesh and 3.4% of element updates being rolled back for the polynomial mesh.

In Figure 3.14, we compare the theoretical timestep size based on the analytic solution against the time-resolved location of rollback for the Carrier-Greenspan problem on the uniform mesh. Rollback in this case tends to only occur near regions where the timestep is changing. At any given point, only a very small number of updates are rolled-back. However, the location of roll-back depends on the solution and therefore cannot be known a priori in general.

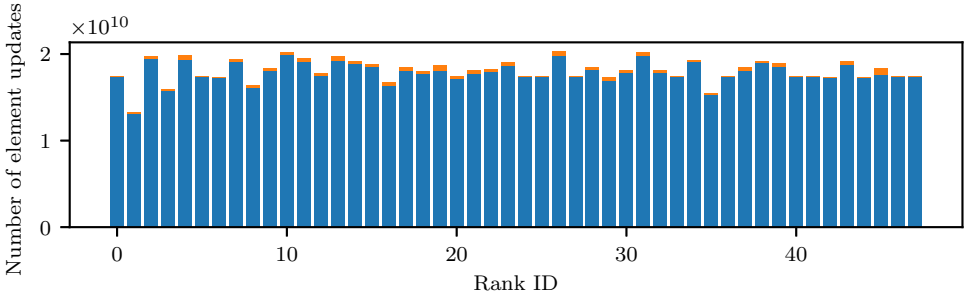
In Figure 3.15, we display the amount of work done by each rank throughout the simulation. The plots show that the load balancing approaches do a reasonable job of balancing



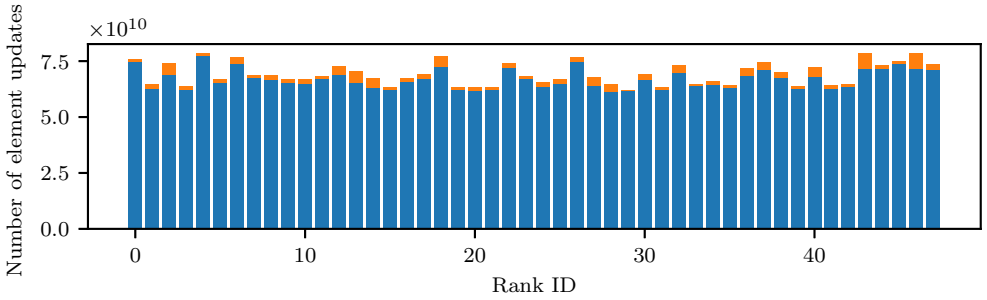
(a) Lake at Rest–Uniform



(b) Lake at Rest–Polynomial



(c) Carrier-Greenspan–Uniform



(d) Carrier-Greenspan–Polynomial

Figure 3.15: Cumulative number of element updates committed—shown in ■—and rolled back—shown in ■ updates for each rank.

committed updates across all ranks. To estimate the impact of rollback on the time to solution, we introduce the time averaged imbalance as

$$I_C = \frac{1}{T}\int_0^T \frac{\max_r u_c(r,t) - \overline{u_c}(t)}{\overline{u_c}(t)}\,\mathrm{d}t,$$

where $u_c(r,t)$ is the rate of updates committed on rank r at time t, and $\overline{u_c}(t)$ corresponds to the mean number of updates committed at time t. We also define the rollback based imbalance as

$$I_{C+RB} = \frac{1}{T}\int_0^T \frac{\max_r \left(u_c(r,t) + u_{rb}(r,t)\right) - \overline{u_c}(t)}{\overline{u_c}(t)}\,\mathrm{d}t,$$

where $u_{rb}(r,t)$ corresponds to the rate of updates rolled back on rank r. The aim of comparing the two metrics is to identify what percentage of the performance degradation is due to poor load balance versus misspeculation. The update and rollback densities are estimated by calculating how many updates are rolled back or committed every second. In a strictly performance oriented setting, we do not care about the absolute amount of rollback, but rather only about rollback incurred on the critical path, i.e. most overworked rank. In Table 3.4, we compare the two imbalances for the test cases. For the Carrier-Greenspan problem, we observe that due to the fact that the actors which are rolled back are distributed across many ranks, rollback does not accumulate on any single rank. We estimate that the time to solution would only be 2% faster if we didn't incur any rollback. On the other hand, for the polynomial lake at rest problem, the rollback is confined to only a few ranks. In turn, these ranks slow down the otherwise well balanced problem configuration. We predict that we would observe a 12% reduction in time to solution if this rollback were not present.

3.6.5 Conservative Parallel Discrete Event Simulation

An alternative way to parallelize the algorithm would be through conservative parallel discrete event simulation. Whereas the timewarp based algorithm has a mechanism to correct

Mesh Type	Lake at Rest I_C	Lake at Rest I_{C+RB}	Carrier-Greenspan I_C	Carrier-Greenspan I_{C+RB}
Uniform	1.00	1.00	1.12	1.14
Polynomial	1.01	1.13	1.16	1.18

Table 3.4: Time-averaged imbalance with and without rollback

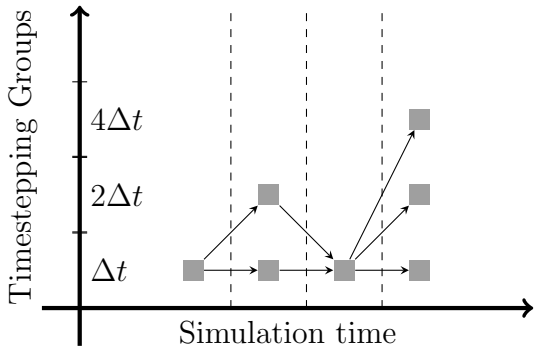


Figure 3.16: Illustration of conservative parallel discrete event simulation with no look ahead.

causality violations. Conservative parallel discrete event simulation requires the program to guarantee that no causality violation can occur. Perhaps the most simple version of a conservative parallel discrete event simulation would be to maintain a global priority queue and execute events with the lowest timestamp. A schematic of elements updating in various timestepping groups is shown in Figure 3.16. This implementation suffers from insufficient available parallelism and significant synchronization overhead. For the hurricane storm surge problem considered in [33], a mesh with $\mathcal{O}(10^6)$ elements only has $\mathcal{O}(10^4)$ elements stepping in the smallest timestepping group. Amdahl's law limits the scalability of any such approach.

Conservative parallel discrete event simulations introduce more parallelism into the system through the introduction of *look ahead*. If for a given global timestamp, we can ensure that no causality violations can occur over some time interval, all events scheduled within that interval may be safely executed. The use of a global priority queue corresponds to a look ahead of zero. To derive look ahead estimates for systems of conservation laws, we must prove that no push flux—the result of a neighbor updating—will be processed within the

look ahead interval.

To achieve a viable implementation using a conservative parallel discrete event simulation, two major problems need to be resolved.

1. *Energy-based look ahead estimates*: The wavespeed $|\Lambda|$ can be bounded above and below by the energy of the system with a constant for Burgers' equation and the shallow water equations. This intuition provides a useful glimpse into the evolution of Λ. We know from the Burgers' equation example used in the motivation of this chapter that any look ahead estimate must consider the discrete domain of dependence of the problem. This requires global knowledge since pathological spikes of energy could exist anywhere. However, a hope would be that once initial global communication has occurred, we may be able to derive estimates for local energy growth, and thus bound changes in wavespeed $|\Lambda|$. By determining the growth of Λ, look ahead estimates can then be determined by deriving lower bounds on when neighboring actors will update.

2. *Loosening of TVD properties*: The aim of this thesis is to derive a total variation diminishing timestepping scheme. The analysis used in this chapter requires that the event trace be locally ordered. From a parallel discrete event standpoint, this is problematic as it may generate event cascades, i.e. events scheduled on neighboring actors with the same timestamp. This complicates the derivation of look ahead estimates, because in addition to bounding the growth of $|\Lambda|$, we also need to ensure that no locally ordering violations can occur. Lastly, the binning of timesteps to powers of two of $\Delta t_{\min}$ also presents issues. Given an actor at timestep 10 with a largest allowable timestep of 15, the next update would occur at 24 (i.e. the largest multiple of 8). If however, at time 10, a message is processed that causes the wavespeed to decrease and the allowable timestep increased to 16, the timestep would occur sooner—rather than later—at time 16 (the largest multiple of 16). Thus, there are also cases where lowering

the wavespeed causes the proposed timestep to *decrease*. We conjecture that the need for locally ordered event traces can be eliminated by relaxing the requirement of total variation diminishing to total variation bounded. Total variation bounded solutions correspond to a larger, but still stable solution class. The relaxation to total variation bounded solutions may be inevitable as we move to higher order timestepping schemes, e.g. [30]. Apart from allowing for easier derivation of look ahead estimates, eliminating the locally ordered event trace requirement would also eliminate the need to bin timesteps, reducing the total amount of work performed.

Given the difficulties required to obtain meaningful look ahead estimates, we have primarily focused on optimistic speculation in this thesis. Rather than focus on deriving look ahead estimates, we have found it easier to derive tests which identify when speculation is guaranteed to be rolled-back based on local information and then simply wait for the message that causes the rollback of any subsequent events to be processed. Furthermore, given the near optimal performance of the optimistic parallel discrete event simulator for this small problem, we need to identify use cases to motivate development of conservative parallel discrete event simulation.

3.7 Conclusion

This work presented an adaptive local timestepping algorithm for conservation laws. Loop invariants were used to derive a proof of formal correctness, ensuring that the algorithm is total variation stable even when considering dynamic changes in local wave speeds. Furthermore, the algorithm was parallelized using a speculative parallel discrete event simulator. Results indicate that the parallelization recovers a majority of the expected speed-up, when accounting for the cost of the CFL condition.

As part of our future work, we will examine higher order timestepping strategies. The

forward Euler step forms the basis of higher order strong stability preserving methods. Future work will include the development of higher order adaptive timestepping methods in the spirit of [30]. Furthermore, we aim to apply this method to physically relevant coastal modeling problems to better quantify benefit of adaptive local timestepping to communities of interest.

At a more general level this work makes two important contributions to the development of asynchronous algorithms. Firstly, the nonlinearity of this problem makes knowing the task graph a priori impossible. While there has been an emergence of task-based runtimes in recent years to address architectural heterogeneity, this work calls into question the efficacy of a task-dependency graph as the sole means of representing scientific computing work flows within these modern runtime systems. Close interdisciplinary work is required with runtime system engineers and applied mathematicians to ensure that avenues for efficient implementations of adaptive algorithms remain accessible.

Secondly, the application of loop invariant analysis in order to assess the correctness of application significantly simplifies the development. Formal correctness techniques are expected to play an increasingly important role in the era of extreme heterogeneity as exhaustively replicating the states of a system in a highly concurrent setting will become increasingly difficult [125]. The results here showcase how to use the invariant analysis proposed in [10]. Due to the formal correctness proof, we ran into significantly fewer bugs during application development. Nevertheless, we found the derivation of the proof to be tedious. Automating this proof technique would allow us to better manage complexity and support the derivation of increasingly complicated algorithms.

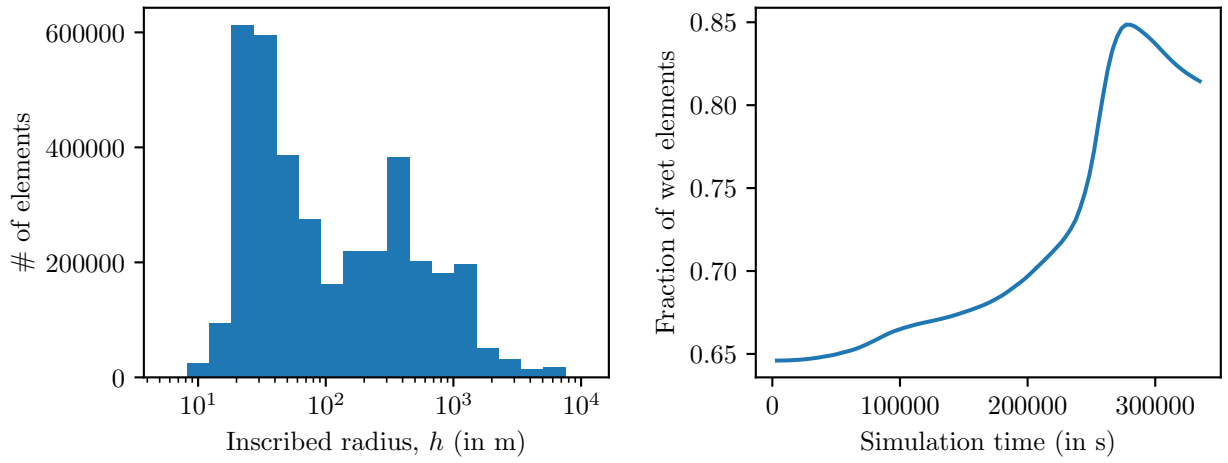
Chapter 4

Conclusion

While load balancing and local timestepping certainly display great promise for the acceleration of hurricane storm surge, their implementation in a storm surge code is left as future work. In this chapter we conclude with an analysis of the impact of this work on hurricane storm surge, examine local timestepping through a Tao of parallelism analysis [106], and conclude with thoughts on storm surge simulations in post-Moore's era computing.

4.1 Implications for hurricane storm surge

In order to make the DG method viable for hurricane storm surge, we require a reduction in required compute resources to still complete a storm surge simulation within a two hour time window. While the proposed local timestepping method represents a significant building block towards using adaptive local timestepping for the simulation of hurricane storm surge, the method must be extended in several important ways. Firstly, the method must be extended to higher-order timestepping schemes. Both partitioned Runge-Kutta methods [30] and the extrapolation based methods in [59] offer potential means of achieving this. The high order timestepping schemes extend straightforwardly to the generalized event trace setting. However, mapping these methods onto a parallel discrete event engine requires more work. Firstly, the extension to multiple dimensions is non-trivial. In the proof of correctness, we showed that a submesh would only send a single message to its neighbor

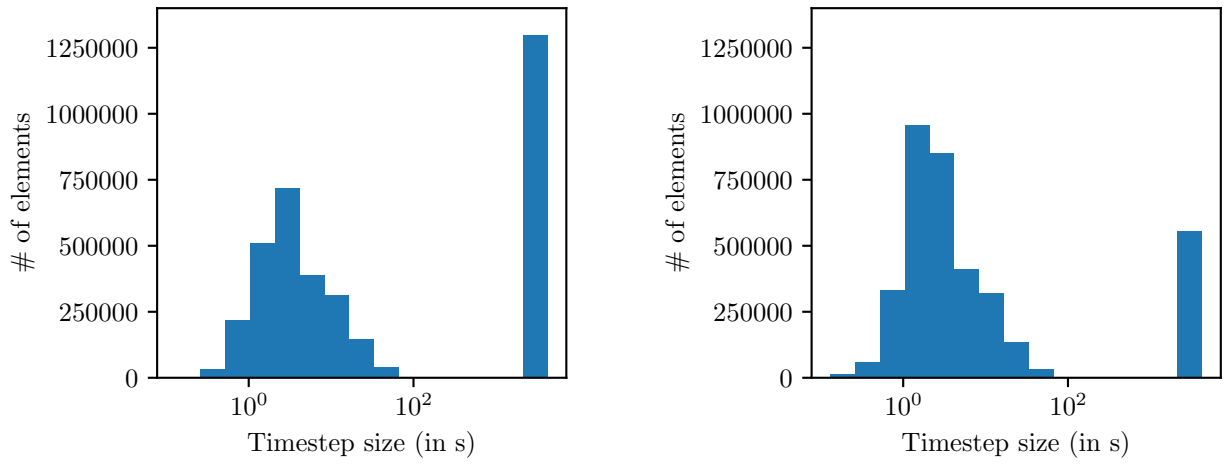


(a) Element sizes

(b) Wet fraction

Figure 4.1: Storm36 metadata

per timestep. This was a result of assuming that the minimum diameter of the submeshes was at least two cells. In a multi-dimensional setting, it is possible that a submesh updates, sends unforced push fluxes to a neighbor, and then receives a neighboring push flux that forces the other neighbors to update even though they have already received a push flux. This requires additional control flow as well as incorporation of these states into the proof of correctness. The second issue arises due to the presence of multiple Runge-Kutta stages per timestep as well as multiple halo-exchanges within a given Runge-Kutta stage. For a given timestepping method these exchanges are statically determined. These message exchanges can be thought of as a static task-graph (determined at compile time) embedded inside a dynamic task-graph (determined based on the state of the cell). When executing the tasks inside a static task-subgraph, we know precisely what messages need to be processed before progress can be made. During these portions of the algorithm, any speculation will be rolled back, and insofar we should always wait on these messages. Better abstractions for embedding subgraphs inside a speculative execution context would simplify algorithm development.



(a) $t = 2\,\mathrm{hr}$ (b) $t = 77\,\mathrm{hr}$

Figure 4.2: Timestep distribution during Storm36

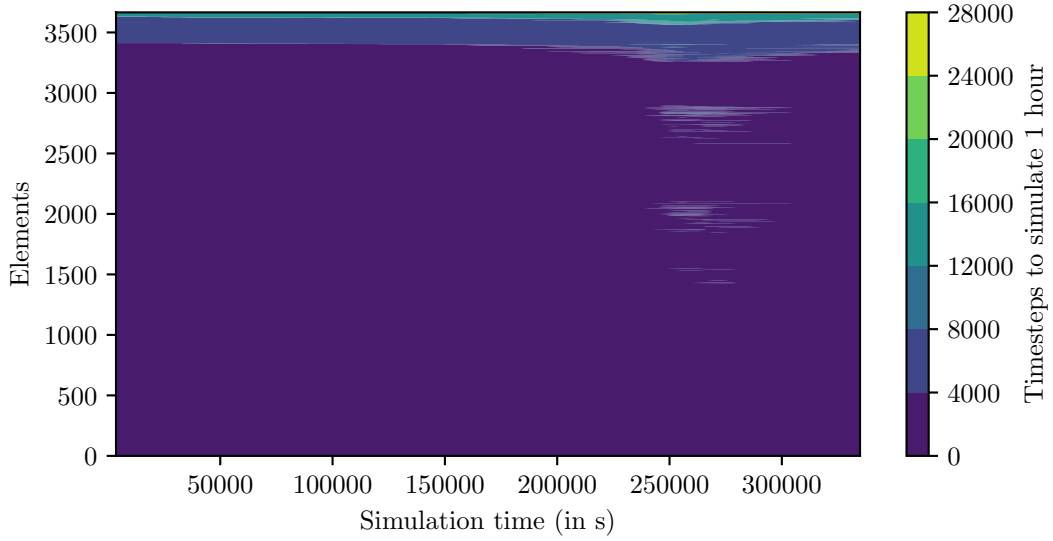


Figure 4.3: Work distribution during Storm36

Although high order adaptive local timestepping is a topic of current research, we can use the performance model developed in Chapter 3 to approximate the speed-up for a hurricane storm surge simulation. Here we use an ADCIRC solution to the Storm36 problem considered in Chapter 2 to compute the wavespeed $|\Lambda|$ at each element. The simulation is run for 3.75 days and the output is recorded hourly. Figure 4.1 shows both the wet fraction as well as the distribution of length scales h, the inscribed radius of a given element. The mesh contains 3.6 million elements with h ranging from 5.5 m to 8.5 km. We use the output to compute $|\Lambda|$ and determine the theoretical speed-up S^{th}. Based on the minimum inscribed radius and assuming a CFL number of 1/3, we determine the largest stable timestep to be 0.13 s and bin the remaining timesteps relative to this $\Delta t_{\min}$. Figure 4.2 shows the distribution of timesteps for both before the storm makes landfall and at peak inundation. In Figure 4.2a, only 0.1% of elements are required to take a timestep of $\Delta t_{\min}$, and the median timestep size is 4.16 s ($32\Delta t_{\min}$). As the storm makes landfall in Figure 4.2b, the number of elements in the finest timestepping group remains low at 0.3%. However significant inundation has occurred. Dry cells are binned into a timestep of 1 hour. The number of cells in this timestepping group has decreased from 35.4% to 15.1% and the median timestep size has also decreased to 2.08 s ($16\Delta t_{\min}$). To highlight the small number of cells at the finest timestepping level, we present the work w_j at every thousandth element as a function of simulation time in Figure 4.3. We have intentionally left the colorbar scale linear to highlight that a very small percentage of the elements requires stepping at the finest level. With this information, we can compute the theoretical speed-up S^{th} to be 15.1. Importantly, this number will be smaller in practice since the actual timestep size will also depend on other elements in the given submesh. The discrepancy in expected speed-up computed from [33] to the number presented here is due to the fact that in [33] the number of timestepping groups was limited to 4 and they used a different mesh. The stability results associated with our adaptive timestepping scheme make this algorithm a good candidate for hurricane storm surge. With proper limiting

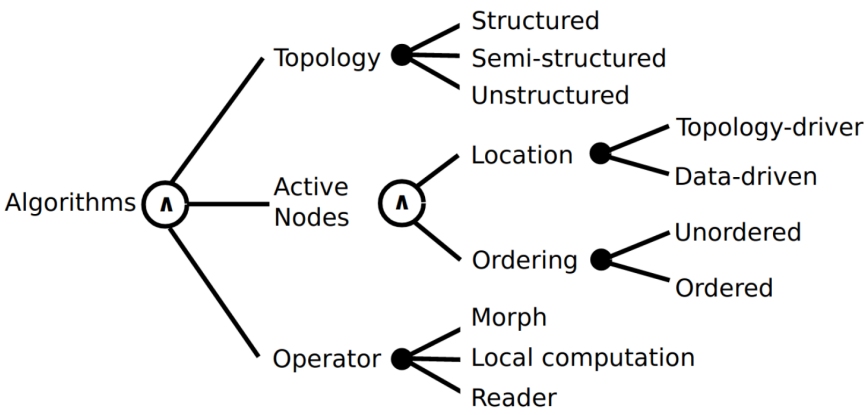


Figure 4.4: Tao classification of parallel algorithms [106]

of bad speculation, we can run the simulation in an almost task-graph manner. Causality violations are relatively rare, yet continuously occurring, making semi-static re-assignment of timestepping groups non-viable. One important question that remains is what is the available parallelism in the simulation? Does our speculative approach bypass the limitations of the adaptive timestepping method, which requires a synchronization after each Δt_{min}? This is certainly a lower bound to the amount of available parallelism. On the upper end, we are constrained by the parallelism available in the event trace, which is only known a posteriori. These performance analyses are the subject of future work.

4.2 Tao analysis of adaptive timestepping

The tao of parallelism categorizes algorithms based on salient features to guide parallel implementations [106]. Programs are thought of as a graph of abstract data types (ADT) and algorithms. In the case of DG finite element methods, the abstract data types correspond to the submeshes, and the edges of the graph are used to reason about dependencies, i.e. which flux buffers need to have been processed before proceeding. The algorithm corresponds to the timestepping scheme, which exchanges flux buffers and updates elements. The tao analysis then examines three main features shown in Figure 4.4.

- **T**opology: The topology refers to the ADT graph, for example, if a task is embarrassingly parallel, this might be represented as a graph with an empty edge set. Graphs are classified by the amount of information required to describe them, ranging from structured to semi-structured to unstructured. For our work, we consider the submesh graph generated by the partitioning of a dual graph of a planar mesh. The most important invariant of the graph is that it remains static throughout the simulation. Since we are able to utilize this information to guarantee once an event can be safely processed, we refer to the graph as semi-structured.
- **A**ctive nodes: At any given point in the simulation, active nodes are nodes which may perform compute. In the case of the task-based synchronous timestepping, a submesh may only update once it has received the updated boundary state from each of the submesh's neighbors, i.e. once all of its dependencies have been satisfied. The topic of active nodes is split into two further subcategories, *location* and *ordering*. Location emphasizes whether the active nodes are purely determined by the graph (referred to as topology-driven) or if the state also influences whether or not the algorithm is able to advance (referred to as data-driven). For both synchronous and adaptive local timestepping algorithms, the methods are data-driven. Based on whether or not a submesh has processed a message determines whether or not it can advance to the next timestep. The second subcategory is *ordering*. Timestepping methods are inherently ordered. It is nonsensical to have a submesh update in a non-sequential order.
- **O**perator: Operators are applied to the ADTs. If the operator changes the topology of the graph it is classified as *morph*. A *local computation* updates the state of the ADT, and a *reader* changes neither the state nor the topology. All of our timestepping methods are local computations. Updating the state does not introduce new vertices into the ADT graph.

All three timestepping schemes for hurricane storms surge—asynchronous timestepping with

a fixed timestep, asynchronous local timestepping, and synchronous timestepping with an adaptive timestep—rely one the same underlying ADT, with an irregular static topology, consist of operators that perform local computation, and have data-driven ordered active nodes. To distinguish between the types of parallelism between each of the timestepping scheme, we rely on the further classification of ordered algorithms by Hassaan [56, 57]. Hassaan introduces the notion of a *Kinetic Dependence Graph* (KDG) to classify ordered algorithms. The KDG is defined by a triple (G, P, U), where:

- G is the task graph,
- P is a *safe source test*,
- U is an update rule.

The task graph looks similar to the graph associated with the ADTs. In this case directed edges correspond to the next pending flux buffer exchange. The update rule corresponds to the method for updating the ADT, i.e. stepping the mesh forward in time. The key distinction between the three timestepping methods comes from the safe source test. Given an active node of the graph, i.e. a submesh that can be updated, this function examines the graph, and determines whether or not that submesh can locally update. If the safe source test is trivial, any active node can update in parallel and the method is called *source stable*.

Coming back to the timestepping schemes, the asynchronous fixed timestepping algorithm is an example of a source stable KDG. Once all messages have been processed at a timestamp, we can safely—here in reference to the algorithm, not the numerical stability—advance to the next timestep. This importantly is contrasted with the other two methods in which case a non-local condition (the CFL condition) needs to be satisfied in order to ensure that the submesh can be safely updated, i.e. the currently proposed timestep will not need to be shortened. Note that the need to check the safety of an update arises due to the fact the shallow water equations are non-linear. Mathematically, the safe-source test requires examining the state of the full domain of dependence. For linear systems of equations, this

problem is statically determined, and so timesteps can be proposed (even local timesteps) such that the source test can be guaranteed to be satisfied by construction.

For the synchronous adaptive timestep, the safe source test simply checks all cells, ensuring that the domain of dependence at the proposed timestep is inside the region checked by the safe source test. On the other hand, for the speculative adaptive local timestepping algorithm, we assume that generally the source test will evaluate to true and then rollback in the case that that assumption was incorrect. The performance optimizations outlined in Section 3.5.2 then provide two important limitations to bad speculation. The *Reducing unnecessary speculation* fix can be thought of as tighter policing of which nodes are active. This fix acknowledges outstanding push fluxes from neighbors as dependencies, and so the node forgoes the executing of further updates until these waited upon messages arrive, and the node reactivates. The *Avoiding small timesteps due to binning* fix can be thought of as the introduction of a non-trivial source test. By examining the state of an active node, we are able to defer execution until a message from the neighbor arrives. However, rather than wait for the safe source test to be resolved, we introduce dependencies by forcing selected neighbors to synchronize. Areas of future work include coming up with better source stable tests. In particular, considering the lake at rest problem on the polynomial mesh, even though the set of equations is non-linear, the solution is source stable by construction, and insofar it should be possible to eliminate all rollback. This could be thought of as being equivalent to enforcing the dependencies of a locally linearized local timestepping problems, and then only speculating when non-linearities dominate the evolution.

Amorphous data-parallelism arises out of tao-analysis to quantify optimal parallel performance. *Amorphous data parallelism* describes the amount of parallelism found in active nodes throughout the simulation. Parallelism profiles inform what performance benefits we may expect to see. Comparing the fixed timestep asynchronous timestepping scheme with the synchronous adaptive timestepping method we see little difference in available paral-

lelism, the difference being the cost of the synchronization associated with updating the global timestep. The direct impact may be unclear since taking a global adaptive timestep may reduce the overall work. As an example, the fixed timestep sizes of year long simulations of the Australian coast are limited by a brief period during the rainy season where inland estuaries flood. Since the timestep is fixed, it will be unnecessarily stringent for the remainder of the year. Navigating this trade-off is problem specific. The deciding factors require comparing the cost of the all-reduces to update the adaptive timestep versus the suboptimal fixed size timestepping. For the locally adaptive method, we can expect to see any critical path shortening also found for the adaptive synchronous timestepping algorithm. We may experience further shortening of the critical path through good speculation, but we have not found a use case where this plays a dominant role. Primarily, the benefit of the local timestepping can be attributed to work reduction. The amount of parallelism decreases since there are fewer active nodes. But since we'd like to simulate the storm with fewer nodes the more important question is given the available parallelism how well can we strong scale. By enabling speculation, the aim is to keep more nodes active allowing more work to be done. The synchronizations, which in this case are managed through GVT are hidden behind useful work, and furthermore, distributed performance studies done for traffic simulations suggest that Devastator should be able to hide the cost of GVT over core counts relevant for storm surge simulation [25].

4.3 Impact of the end of Moore's Law

We conclude this thesis with a look towards the future. Moore's Law, the techno-economic factors which have led to the exponential growth in computer performance over the past 50 years is ending as physical limitations in lithography are attained [113]. Rather than producing faster general purpose hardware, solutions at the computer architecture level will

have to use transistors more efficiently to provide continued performance gains in computing. These architectures will become increasingly specialized for specific problems, resulting in computing systems with a swiss army knife of accelerators [125]. These changes are already underway within the machine learning community with start-ups such as Tenstorrent, Cerebaras, and SambaNova as well as initiatives under industry giants such as Microsoft's Project Catapult, Facebook's Big Sur nodes, and Google's tensor processing units. While the economic drivers for industry primarily revolve around accelerating machine learning workflows, these innovations will translate to scientific computing with mixed success.

The technical difficulty associated with effectively using these new computers threatens to split the scientific computing community into two camps: those whose applications are able to efficiently run on these new architectures, and those who see limited benefit from new architectures either due to hardware constraints and/or the high opportunity cost of extensive code refactoring. Originally, the road map to exascale had proposed a two path solution, one based on manycore processors and the other based on GPUs. The difficulties in excising performance from the second generation Intel Xeon Phi (KNL) architecture and subsequently weak demand for the product, lead to the shuttering of the CPU path as Intel canceled the Xeon Phi product line. The Exascale Computing Initiative has since fully committed to the GPU path towards an exascale machine. With two exascale GPU-based clusters—Oak Ridge Leadership Computing Facility (OLCF)'s Frontier and Argonne Leadership Computing Facility's Aurora—scheduled for delivery in 2021. The interests of the winners and losers of these post-CPU architectures are apparent in the contrasting recent acquisitions made by OLCF and the Texas Advanced Computing Center (TACC). The most recent acquisition by TACC was Frontera, a predominantly Intel Xeon cluster. These server class x86 CPU nodes are substantially less power efficient than GPUs. At a peak performance ($R_{\max}$) of 23.5 PFLOPS, Frontera consumes 5.9 MW of power (4.0 PFLOPS/MW)[1]. In comparison,

[1]Performance results for Frontera is taken from November 2019 TOP500 table, and the power consumption

OLCF's GPU-based Summit, which was acquired one year before Frontera, boasts a peak performance ($R_{\max}$) of 149 PFLOPS at 10 MW of of consumption (14.9 PFLOPS/MW)[2]. The trade-off is that the Intel Xeon-based architecture requires very little modification to existing code bases, greatly improving immediate usability. Whereas tremendous effort has been poured into making applications performant on GPUs with substantial funding from the US Department of Energy (DOE). This divergence in hardware and application readiness will lead to a schism in scientific computing where access to the most performant machines is restricted to a few highly optimized applications, while the remainder of the scientific computing community will continue to use machines like Frontera to achieve their research goals. Through improvements in programming models and code portability, these post-CPU architectures may slowly achieve more widespread adoption.

At the same time, the field of computational science is not in stasis. Different numerical methods will become more commonly used, reflecting funding priorities as well as competitive advantages of software frameworks. Regardless of the adoption of post-Moore architectures, the fundamental design constraints are incontrovertible. Algorithms which can effectively leverage the increasing FLOPS to bandwidth ratios will ultimately win out over codes that do not. The algorithms currently used to simulate hurricane storm surge, low order stencil codes, are expected to be among the losers. We propose three research directions which may lead to improved performance of storm surge simulation on future architectures:

1. *Stable high order discretizations*: The impact of high order finite element methods has already been extensively studied [17, 44, 45, 102, 118, 123] and specifically examined in the context of the shallow water equations in [13, 18, 19, 47, 85, 128]. For smooth solutions, not only do we achieve higher accuracy per degree of freedom and are able to coarsen the mesh, which decreases the number of required timesteps, the larger mass

estimate was taken from `https://www.tacc.utexas.edu/diy-how-to-build-a-supercomputer`, accessed May 11, 2020.

[2]Both performance and power results are based on the November 2019 TOP500 entry for Summit.

matrices associated with each element require a higher arithmetic intensity to update. For memory bound applications, this additional computational work generates no performance penalty as memory traffic remains the performance bottleneck. However, the emphasis on stable is not to be underestimated. In the case of the shallow water equations, specifically stable treatment of the wet-dry interface remains a thorny issue. Promising approaches include the use of WENO like algorithms to accurately compute derivatives near the wet-dry interface [13] and projection methods that map troubled high-order cells to refined low-order cells [107].

2. *Lower floating point precision*: Another way to reduce the bandwidth requirements is to make the solution smaller. By moving from double precision to lower precision solutions, the amount of data loaded from memory can be significantly reduced. Work by Düben and Palmer has examined the impact of reduced precision on weather simulation [39]. Findings suggest that the size of floating point numbers could be reduced to 15 bits with no visible degradation of the solution. Extension of these results to hurricane storm surge must be done cautiously. Careful analysis must be done to demonstrate that numerical accuracy remains higher than the uncertainty associated with the inputs (storm track, Manning's n, coastal bathymetry). This could introduce significant speed-ups with minimally invasive modifications to the code.

3. *Exploiting data parallelism of ensemble runs*: Ultimately, we do not care about the performance of a single run, but rather the overall performance of an ensemble of runs. These runs are entirely independent of one another, but execute the same code on the same mesh. Assuming that we have a natural size, e.g. a cache line width or warp size, we propose solving that many solutions concurrently thereby exposing data parallelism. Thus, we would ensure that all data in the cache line is being used, lowering the amount of data required to be loaded from memory. This would particularly help with low order

methods, where the sparse matrix-vector products associated with exchanging flux information at the boundary cause significant numbers of cache misses. The impact of this approach on the time to solution remains unclear. While we certainly would induce fewer cache misses, we would have to scale out the single run which executes multiple simulations further than the multiple separate runs. However, overall compute resource utilization would increase. For problems for which time to solution is less important and numerous runs are required, e.g. updating flood insurance maps or uncertainty quantification [50], we would expect to see significant savings in core-hours used.

Bibliography

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

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[3] Cevdet Aykanat, B. Barla Cambazoglu, Ferit Findik, and Tahsin Kurc. Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids. *Journal of Parallel Distributed Computing*, 67(1):77 – 99, 2007.

[4] John Bachan, Dan Bonachea, Paul H Hargrove, Steve Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, and Scott B Baden. The UPC++ PGAS library for exascale computing. In *Proceedings of the Second Annual PGAS Applications Workshop*, PAW '17, 2017.

[5] Richard F. Barrett, X. S. Hu, Sudip S. Dosanjh, S. Parker, Michael A. Heroux, and J. Shalf. Toward codesign in high performance computing systems. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '12, 2012.

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[7] John B Bell, Clint N Dawson, and Gregory R Shubin. An unsplit, higher order godunov method for scalar conservation laws in multiple dimensions. *Journal of Computational Physics*, 74(1):1 – 24, 1988.

[8] Martin Berzins, Justin Luitjens, Qingyu Meng, Todd Harman, Charles A. Wight, and Joseph R. Peterson. Uintah: A scalable framework for hazard analysis. In *Proceedings of the 2010 TeraGrid Conference*, TG '10, 2010.

[9] A. Bhatele, S. Fourestier, H. Menon, L. V. Kale, and F. Pellegrini. Applying graph partitioning methods in measurement-based dynamic load balancing. Technical report, Lawrence Livermore National Laboratory, 2011.

[10] Paolo Bientinesi and Robert A. van de Geijn. Goal-oriented and modular stability analysis. *SIAM Journal on Matrix Analysis and Applications*, 32(1):286–308, 2011.

[11] Sébastien Blaise and Amik St-Cyr. A dynamic *hp*-adaptive discontinuous Galerkin method for shallow-water flows on the sphere with application to a global tsunami simulation. *Monthly Weather Review*, 140(3):978–996, 2012.

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

[13] Boris Bonev, Jan S. Hesthaven, Francis X. Giraldo, and Michal A. Kopera. Discontinuous Galerkin scheme for the spherical shallow water equations with applications to tsunami modeling and prediction. *Journal of Computational Physics*, 362:425 – 448, 2018.

[14] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, Nov 2013.

[15] Maximilian Bremer, Kazbek Kazhyken, Hartmut Kaiser, Craig Michoski, and Clint Dawson. Performance comparison of HPX versus traditional parallelization strategies for the discontinuous Galerkin method. *Journal of Scientific Computing*, 80(2):878–902, 2019.

[16] Maximilian H. Bremer, John D. Bachan, and Cy P. Chan. 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, pages 44–56, 2018.

[17] Alexander Breuer, Alexander Heinecke, and Michael Bader. Petascale local time stepping for the ADER-DG finite element method. In *International Parallel and Distributed Processing Symposium*, IPDPS '16, pages 854–863, May 2016.

[18] S.R. Brus, D. Wirasaet, E.J. Kubatko, J.J. Westerink, and C. Dawson. High-order discontinuous Galerkin methods for coastal hydrodynamics applications. *Computer Methods in Applied Mechanics and Engineering*, 355:860 – 899, 2019.

[19] Steven R. Brus. *Efficiency Improvements for Modeling Coastal Hydrodynamics through the Application of High-Order discontinuous Galerkin Solutions to the Shallow Water Equations*. PhD thesis, University of Notre Dame, 2017.

[20] S. Bunya, J.C. Dietrich, J.J. Westerink, B.A. Ebersole, J.M. Smith, J.H. Atkinson, R. Jensen, D.T. Resio, R.A. Luettich, C. Dawson, V.J. Cardone, A.T. Cox, M.D. Powell, H.J. Westerink, and H.J. Roberts. A high resolution coupled riverine flow, tide, wind, wind wave and storm surge model for Southern Louisiana and Mississippi: Part I - Model development and validation. *Monthly Weather Review*, 138:345–377, 2010.

[21] Shintaro Bunya, Ethan J Kubatko, Joannes J Westerink, and Clint Dawson. A wetting and drying treatment for the Runge–Kutta discontinuous Galerkin solution to the shallow water equations. *Computer Methods in Applied Mechanics and Engineering*, 198(17-20):1548–1562, 2009.

[22] Carsten Burstedde, Lucas C Wilcox, and Omar Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[23] G.F. Carrier and H.P. Greenspan. Water waves of finite amplitude on a sloping beach. *Journal of Fluid Mechanics*, 4(1):97–109, 1958.

[24] Vincenzo Casulli. A high-resolution wetting and drying algorithm for free-surface hydrodynamics. *International Journal for Numerical Methods in Fluids*, 60(4):391–408, 2009.

[25] C. Chan, B. Wang, J. Bachan, and J. Macfarlane. Mobiliti: Scalable transportation simulation using high-performance parallel computing. In *Proceedings of the International Conference on Intelligent Transportation Systems*, ITSC '18, pages 634–641, Nov 2018.

[26] Cy P. Chan, John D. Bachan, Joseph P. Kenny, Jeremiah J. Wilke, Vincent E. Beckner, Ann S. Almgren, and John B. Bell. Topology-aware performance optimization and modeling of adaptive mesh refinement codes for exascale. In *Proceedings of the First Workshop on Optimization of Communication in HPC*, COM-HPC '16, pages 17–28, 2016.

[27] Bernardo Cockburn, San-Yih Lin, and Chi-Wang Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: One-dimensional systems. *Journal of Computational Physics*, 84(1):90 – 113, 1989.

[28] Bernardo Cockburn and Chi-Wang Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws II: General framework. *Mathematics of Computation*, 52(186):411–435, 1989.

[29] Bernardo Cockburn and Chi-Wang Shu. Runge–Kutta discontinuous Galerkin methods for convection-dominated problems. *Journal of Scientific Computing*, 16(3):173–261, 2001.

[30] Emil M. Constantinescu and Adrian Sandu. Multirate timestepping methods for hyperbolic conservation laws. *Journal of Scientific Computing*, 33(3):239–278, December 2007.

[31] Clint Dawson and Robert Kirby. High resolution schemes for conservation laws with locally varying time steps. *SIAM Journal on Scientific Computing*, 22(6):2256–2281, 2001.

[32] Clint Dawson, Ethan J. Kubatko, Joannes J. Westerink, Corey Trahan, Christopher Mirabito, Craig Michoski, and Nishant Panda. Discontinuous Galerkin methods for modeling hurricane storm surge. *Advances in Water Resources*, 34(9):1165–1176, 2011.

[33] Clint Dawson, Corey Jason Trahan, Ethan J. Kubatko, and Joannes J. Westerink. A parallel local timestepping Runge–Kutta discontinuous Galerkin method with applications to coastal ocean modeling. *Computer Methods in Applied Mechanics and Engineering*, 259:154 – 165, 2013.

[34] M. de la Asunción, M.J. Castro, J.M. Mantas, and S. Ortega. Numerical simulation of tsunamis generated by landslides on multiple GPUs. *Advances in Engineering Software*, 99:59 – 72, 2016.

[35] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2-3):133–152, 2005.

[36] J. C. Dietrich, S. Bunya, J. J. Westerink, B. A. Ebersole, J. M. Smith, J. H. Atkinson, R. Jensen, D. T. Resio, R. A. Luettich, C. Dawson, V. J. Cardone, A. T. Cox, M. D. Powell, H. J. Westerink, and H. J. Roberts. A high resolution coupled riverine flow, tide, wind, wind wave and storm surge model for southern Louisiana and Mississippi: Part ii - synoptic description and analyses of Hurricanes Katrina and Rita. *Monthly Weather Review*, 138:378–404, 2010.

[37] J.C. Dietrich, J.J. Westerink, A.B. Kennedy, J.M. Smith, R. E. Jensen, M. Zijlema, L.H. Holthuijsen, C. Dawson, R.A. Luettich, M.D. Powell, V.J. Cardone, A.T. Cox, G.W. Stone, H. Pourtaheri, M.E. Hope, S. Tanaka, L.G. Westerink, H. J. Westerink, and Z. Cobell. Hurricane Gustav (2008) waves and storm surge: Hindcast, synoptic analysis and validation in southern Louisiana. *Monthly Weather Review*, 139:2488–2522, 2011.

[38] J.C. Dietrich, M. Zijlema, J.J. Westerink, L.H. Holtjuijsen, C. Dawson, Jr. R.A. Luettich, R. Jensen, J.M. Smith, G.S. Stelling, and G.W. Stone. Modeling hurricane wave and storm surge using integrally-coupled, scalable computations. *Coastal Engineering*, 58:45–65, 2011.

[39] Peter D. Düben and T. N. Palmer. Benchmark tests for numerical weather forecasts on inexact hardware. *Monthly Weather Review*, 142(10):3809–3829, 2014.

[40] Michael Dumbser and Raphaël Loubère. A simple robust and accurate a posteriori sub-cell finite volume limiter for the discontinuous Galerkin method on unstructured meshes. *Journal of Computational Physics*, 319:163 – 199, 2016.

[41] Denys Dutykh and Didier Clamond. Modified shallow water equations for significantly varying seabeds. *Applied Mathematical Modelling*, 40(23):9767 – 9787, 2016.

[42] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. Cray cascade: A scalable hpc system based on a dragonfly network. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 1–9, Washington, DC, USA, 2012. IEEE Computer Society.

[43] Chaulio R. Ferreira and Michael Bader. Load balancing and patch-based parallel adaptive mesh refinement for tsunami simulation on heterogeneous platforms using Xeon Phi coprocessors. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '17, pages 12:1–12:12, 2017.

[44] Paul Fischer, Misun Min, Thilina Rathnayake, Som Dutta, Tzanio Kolev, Veselin Dobrev, Jean-Sylvain Camier, Martin Kronbichler, Tim Warburton, Kasia Swirydowicz, and Jed Brown. Scalability of high-performance PDE solvers, 2020. arXiv:2004.06722 [cs.PF].

[45] Lucas Friedrich, Gero Schnücke, Andrew R. Winters, David C. Del Rey Fernández, Gregor J. Gassner, and Mark H. Carpenter. Entropy stable space–time discontinuous Galerkin schemes with summation-by-parts property for hyperbolic conservation laws. *Journal of Scientific Computing*, 80:175–222, 2019.

[46] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[47] Rajesh Gandham, David Medina, and Timothy Warburton. GPU accelerated discontinuous Galerkin methods for shallow water equations. *Communications in Computational Physics*, 18(1):37–64, 2015.

[48] Nickolay Y. Gnedin, Vadim A. Semenov, and Andrey V. Kravtsov. Enforcing the courant–friedrichs–lewy condition in explicitly conservative local time stepping schemes. *Journal of Computational Physics*, 359:93 – 105, 2018.

[49] Sigal Gottlieb, Chi-Wang Shu, and Eitan Tadmor. Strong stability-preserving high-order time discretization methods. *SIAM Review*, 43(1):89–112, 2001.

[50] Lindley Graham, Troy Butler, Scott Walsh, Clint Dawson, and Joannes J. Westerink. A measure-theoretic algorithm for estimating bottom friction in a coastal inlet: Case study of Bay St. Louis during Hurricane Gustav (2008). *Monthly Weather Review*, 145(3):929–954, 2017.

[51] Jean-Luc Guermond, Richard Pasquetti, and Bojan Popov. Entropy viscosity method for nonlinear conservation laws. *Journal of Computational Physics*, 230(11):4248 – 4267, 2011. Special issue High Order Methods for CFD Problems.

[52] Shubhangi Gupta, Barbara Wohlmuth, and Rainer Helmig. Multi-rate time stepping schemes for hydro-geomechanical model for subsurface methane hydrate reservoirs. *Advances in Water Resources*, 91:78 – 87, 2016.

[53] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020.

[54] Ernst Hairer. Order conditions for numerical methods for partitioned ordinary differential equations. *Numerische Mathematik*, 36(4):431–445, Dec 1981.

[55] Ami Harten. High resolution schemes for hyperbolic conservation laws. *Journal of Computational Physics*, 49(3):357–393, 1983.

[56] Muhammad Amber Hassaan. *Parallelization of Ordered Irregular Algorithms*. PhD thesis, University of Texas at Austin, 2016.

[57] Muhammad Amber Hassaan, Donald D. Nguyen, and Keshav K. Pingali. Kinetic dependence graphs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 457–471, New York, NY, USA, 2015. ACM.

[58] Jan S Hesthaven and Tim Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Springer Science & Business Media, 2007.

[59] Thi-Thao-Phuong Hoang, Wei Leng, Lili Ju, Zhu Wang, and Konstantin Pieper. Conservative explicit local time-stepping schemes for the shallow water equations. *Journal of Computational Physics*, 382:152 – 176, 2019.

[60] M.E. Hope, J.J. Westerink, A.B. Kennedy, P.C. Kerr, J.C. Dietrich, C. Dawson, C.J. Bender, J.M. Smith, R.E. Jensen, M. Zijlema, L.H. Holthuijsen, Jr. Luettich, R.A., M.D. Powell, V.J. Cardone, A.T. Cox, H. Pourtaheri, H.J. Roberts, J.H. Atkinson, S. Tanaka, H.J. Westerink, and L.G. Westerink. Hindcast and validation of Hurricane Ike (2008) waves, forerunner, and storm surge. *Journal of Geophysical Research Oceans*, 118:4424–4460, 2013.

[61] M.E. Hope, J.J. Westerink, A.B. Kennedy, J.M. Smith, H.J. Westerink, A. Cox, S. Nong, K.J. Roberts, D.T. Resio, and Totht A.P. Hurricane Sandy (2012) wind, waves and storm surge in New York Bight. I: Model validation. *Journal of Waterway, Port, Coastal and Ocean Engineering*, 2016.

[62] Laleh Rostami Hosoori and Amir Masoud Rahmani. An adaptive load balancing algorithm with use of cellular automata for computational grid systems. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 419–430, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[63] Intel. An introduction to the intel quickpath interconnect. `http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html`, 2009. Accessed: 2018-07-31.

[64] Nikhil Jain, Abhinav Bhatele, Sam White, Todd Gamblin, and Laxmikant V. Kale. Evaluating hpc networks via simulation of parallel workloads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 14:1–14:12, Piscataway, NJ, USA, 2016. IEEE Press.

[65] Curtis L. Janssen, Helgi Adalsteinsson, Scott Cranford, Joseph P. Kenny, Ali Pinar, David A. Evensky, and Jackson Mayo. A simulator for large-scale parallel computer architectures. *International Journal of Distributed Systems and Technologies*, 1(2):57–73, 2010.

[66] Curtis L. Janssen, Helgi Adalsteinsson, and Joseph P. Kenny. Using simulation to design extremescale applications and architectures: Programming model exploration. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):4–8, March 2011.

[67] David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[68] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[69] Homa Karimabadi, Jonathan Driscoll, Jagrut Dave, Yuri Omelchenko, Kalyan Perumalla, Richard Fujimoto, and Nick Omidi. Parallel discrete event simulations of grid-based models: Asynchronous electromagnetic hybrid code. In Jack Dongarra, Kaj Madsen, and Jerzy Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 573–582, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[70] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[71] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–13. IEEE Computer Society, 1998.

[72] Robert Kirby. On the convergence of high resolution methods with multiple time scales for hyperbolic conservation laws. *Mathematics of Computation*, 72(243):1239–1250, 2003.

[73] Peter M. Kogge and John Shalf. Exascale computing trends: Adjusting to the "new normal" for computer architecture. *Computing in Science & Engineering*, 15(6):16–26, 2013.

[74] Lilia Krivodonova. An efficient local time-stepping scheme for solution of nonlinear conservation laws. *Journal of Computational Physics*, 229(22):8537 – 8551, 2010.

[75] E.J. Kubatko, S. Bunya, C. Dawson, and J.J. Westerink. Dynamic p-adaptive Runge-Kutta discontinuous Galerkin methods for the shallow water equations. *Computer Methods in Applied Mechanics and Engineering*, 198:1766–1774, 2009.

[76] E.J. Kubatko, J.J. Westerink, and C. Dawson. *hp* Discontinuous Galerkin methods for advection dominated problems in shallow water flow. *Computer Methods in Applied Mechanics and Engineering*, 196:437–451, 2006.

[77] Ethan J. Kubatko, Shintaro Bunya, Clint Dawson, Joannes J. Westerink, and Chris Mirabito. A performance comparison of continuous and discontinuous finite element shallow water models. *Journal of Scientific Computing*, 40(1):315–339, Jul 2009.

[78] Ethan J Kubatko, Joannes J Westerink, and Clint Dawson. Semi discrete discontinuous Galerkin methods and stage-exceeding-order, strong-stability-preserving Runge–Kutta time discretizations. *Journal of Computational Physics*, 222(2):832–848, 2007.

[79] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI)*, 42(6):211–222, 2007.

[80] Dmitri Kuzmin. A vertex-based hierarchical slope limiter for p-adaptive discontinuous Galerkin methods. *Journal of Computational and Applied Mathematics*, 233(12):3077 – 3085, 2010.

[81] Randall J. LeVeque. *Numerical methods for conservation laws*. Birkäuser Verlag, 1992.

[82] F. Lörcher, G. Gassner, and C.-D. Munz. A discontinuous Galerkin scheme based on a space–time expansion. I. Inviscid compressible flow in one space dimension. *Journal of Scientific Computing*, 32(2):175–199, Aug 2007.

[83] R.A. Luettich, J.J. Westerink, et al. ADCIRC: A parallel advanced circulation model for oceanic, coastal and estuarine waters, 2017. Users manual available at www.adcirc.org.

[84] Y. Maday, S. Kaber, and E. Tadmor. Legendre pseudospectral viscosity method for nonlinear conservation laws. *SIAM Journal on Numerical Analysis*, 30(2):321–342, 1993.

[85] Simone Marras, Michal A. Kopera, Emil M. Constantinescu, Jenny Suckale, and Francis X. Giraldo. A residual-based shock capturing scheme for the continuous/discontinuous spectral element solution of the 2D shallow water equations. *Advances in Water Resources*, 114:45 – 63, 2018.

[86] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, Min Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. The open community runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference*, HPEC, pages 1–7, Sept 2016.

[87] A. Meister, S. Ortleb, and Th. Sonar. Application of spectral filtering to discontinuous Galerkin methods on triangulations. *Numerical Methods for Partial Differential Equations*, 28(6):1840–1868, 2011.

[88] Harshitha Menon and Laxmikant Kalé. A distributed dynamic load balancer for iterative applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 1–11, 2013.

[89] Dumbser Michael, Käser Martin, and Eleuterio F. Toro. 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):695–717, 2007.

[90] C. Michoski, A. Alexanderian, C. Paillet, E.J. Kubatko, and C. Dawson. Stability of nonlinear convection-diffusion-reaction systems in discontinuous Galerkin methods. *Journal of Scientific Computing*, 70:516–550, 2017.

[91] C. Michoski, C. Dawson, E.J. Kubatko, D. Wirasaet, S. Brus, and J.J. Westerink. A comparison of artificial viscosity, limiters, and filter, for high order discontinuous Galerkin solution in nonlinear settings. *Journal of Scientific Computing*, 2015.

[92] C. Michoski, C. Dawson, C. Mirabito, E.J. Kubatko, D. Wirasaet, and J.J. Westerink. Fully coupled methods for multiphase morphodynamics. *Advances in Water Resources*, 59:95–110, 2013.

[93] C Michoski, Chris Mirabito, Clint Dawson, D Wirasaet, Ethan J Kubatko, and Joannes J Westerink. Adaptive hierarchic transformations for dynamically p-enriched slope-limiting over discontinuous Galerkin systems of generalized equations. *Journal of Computational Physics*, 230(22):8028–8056, 2011.

[94] Misbah Mubarak, Christopher D. Carothers, Robert B. Ross, and Philip Carns. A case study in using massively parallel simulation for extreme-scale torus network codesign. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '14, pages 27–38, New York, NY, USA, 2014. ACM.

[95] Margaret Myers and Robert van de Geijn. LAFF-ON Programming for Correctness, 2018. URL: http://www.cs.utexas.edu/users/rvdg/pubs/LAFFPfC.pdf. Last visited on 2020/02/25.

[96] NERSC. Nersc edison configuration. http://www.nersc.gov/users/computational-systems/edison/configuration/, 2018. Accessed: 2018-07-31.

[97] NOAA National Centers for Environmental Information (NCEI). U.S. billion-dollar weather and climate disasters. https://www.ncdc.noaa.gov/billions/, 2018. Accessed: 2018-05-23.

[98] Y. A. Omelchenko and H. Karimabadi. Parallel asynchronous hybrid simulations of strongly inhomogeneous plasmas. In *Proceedings of the Winter Simulation Conference 2014*, pages 3435–3446, Dec 2014.

[99] Y.A. Omelchenko and H. Karimabadi. Event-driven, hybrid particle-in-cell simulation: A new paradigm for multi-scale plasma modeling. *Journal of Computational Physics*, 216(1):153 – 178, 2006.

[100] Y.A. Omelchenko and H. Karimabadi. HYPERS: A unidimensional asynchronous framework for multiscale hybrid simulations. *Journal of Computational Physics*, 231(4):1766 – 1780, 2012.

[101] Stanley Osher and Richard Sanders. Numerical approximations to nonlinear conservation laws with locally varying time and space grids. *Mathematics of Computation*, pages 321–336, 1983.

[102] Will Pazner and Per-Olof Persson. Approximate tensor-product preconditioners for very high order discontinuous Galerkin methods. *Journal of Computational Physics*, 354:344 – 369, 2018.

[103] Olga Pearce, Todd Gamblin, Bronis R. de Supinski, Martin Schulz, and Nancy M. Amato. Mpmd framework for offloading load balance computation. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*, IPDPS, pages 943–952, 2016.

[104] François Pellegrini. *Scotch and libScotch 5.1 user's guide*. INRIA Bordeaux Sud-Ouest.

[105] Per-Olof Persson and Jaime Peraire. Sub-cell shock capturing for discontinuous Galerkin methods. In *Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit*, page 13. IEEE Computer Society Press, 2006.

[106] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI '11, page 12–25, New York, NY, USA, 2011. Association for Computing Machinery.

[107] Leonhard Rannabauer, Michael Dumbser, and Michael Bader. ADER-DG with a-posteriori finite-volume limiting to simulate tsunamis in a parallel adaptive mesh refinement framework. *Computers & Fluids*, 2018.

[108] William H Reed and TR Hill. Triangular mesh methods for the neutron transport equation. Technical report, Los Alamos Scientific Lab., N. Mex.(USA), 1973.

[109] M. Rietmann, D. Peter, O. Schenk, B. Uçar, and M. Grote. Load-balanced local time stepping for large-scale wave propagation. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '15, pages 925–935, 2015.

[110] Martin Schlegel, Oswald Knoth, Martin Arnold, and Ralf Wolke. Multirate Runge–Kutta schemes for advection equations. *Journal of Computational and Applied Mathematics*, 226(2):345 – 357, 2009. Special Issue: Large scale scientific computations.

[111] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes. Technical Report 97-014, UMN, 1997.

[112] Bruno Seny, Jonathan Lambrechts, Thomas Toulorge, Vincent Legat, and Jean-François Remacle. An efficient parallel implementation of explicit multirate Runge-Kutta schemes for discontinuous Galerkin computations. *Journal of Computational Physics*, 256(1):135–160, 2014.

[113] John Shalf. The future of computing beyond Moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, 2020.

[114] Q. Shao, S.K. Matthäi, and L. Gross. Efficient modelling of solute transport in heterogeneous media with discrete event simulation. *Journal of Computational Physics*, 384:134 – 150, 2019.

[115] Luka Stanisic, Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Arnaud Legrand, Florent Lopez, and Brice Videau. Fast and accurate simulation of multithreaded sparse linear algebra solvers. In *Proceedings of the 21st International Conference on Parallel and Distributed Systems*, ICPADS '15, pages 481–490, 2015.

[116] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. *Concurrency and Computation: Practice and Experience*, 27(16):4075–4090, 2015.

[117] D. Stone, S. Geiger, and G.J. Lord. Asynchronous discrete event schemes for PDEs. *Journal of Computational Physics*, 342:161 – 176, 2017.

[118] Kasia Świrydowicz, Noel Chalmers, Ali Karakus, and Tim Warburton. Acceleration of tensor-product operations for high-order finite element methods. *The International Journal of High Performance Computing Applications*, 33(4):735–757, 2019.

[119] S. Tanaka, S. Bunya, J.J. Westerink, C. Dawson, and R.A. Luettich. Scalability of an unstructured grid continuous Galerkin based hurricane storm surge model. *Journal of Scientific Computing*, 46:329–358, 2011.

[120] Arne Taube, Michael Dumbser, Claus-Dieter Munz, and Rudolf Schneider. A high-order discontinuous Galerkin method with time-accurate local time stepping for the Maxwell equations. *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, 22(1):77–103, 2009.

[121] Corey J. Trahan and Clint Dawson. Local time-stepping in Runge-Kutta discontinuous Galerkin finite element methods applied to the shallow-water equations. *Computer Methods in Applied Mechanics and Engineering*, 217-220:139 – 152, 2012.

[122] Thomas Unfer, Jean-Pierre Boeuf, François Rogier, and Frédéric Thivet. An asynchronous scheme with local time stepping for multi-scale transport problems: Application to gas discharges. *Journal of Computational Physics*, 227(2):898 – 918, 2007.

[123] Carsten Uphoff, Sebastian Rettenberger, Michael Bader, Elizabeth H. Madden, Thomas Ulrich, Stephanie Wollherr, and Alice-Agnes Gabriel. Extreme scale multi-physics simulations of the tsunamigenic 2004 Sumatra megathrust earthquake. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 21:1–21:16, New York, NY, USA, 2017. ACM.

[124] Stefan Vater, Nicole Beisiegel, and Jörn Behrens. A limiter-based well-balanced discontinuous Galerkin method for shallow-water flows with wetting and drying: One-dimensional case. *Advances in Water Resources*, 85:1 – 13, 2015.

[125] Jeffrey S. Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, Brian Van Essen, Shinjae Yoo, Alex Aiken, David Bernholdt, Suren Byna, Kirk Cameron, Frank Cappello, Barbara Chapman, Andrew Chien, Mary Hall, Rebecca Hartman-Baker, Zhiling Lan, Michael Lang, John Leidel, Sherry Li, Robert Lucas, John Mellor-Crummey, Paul Peltz Jr., Thomas Peterka, Michelle Strout, and Jeremiah Wilke. Extreme heterogeneity 2018 - Productive computational science in the era of extreme heterogeneity: Report for DOE ASCR workshop on extreme heterogeneity. Technical report, US Department of Energy Office of Science (SC), 2018.

[126] J. J. Westerink, R. A. Luettich, J. C. Feyen, J. H. Atkinson, C. N. Dawson, H. J. Roberts, M. D. Powell, J. P. Dunion, E. J. Kubatko, and H. Pourtaheri. A basin to channel scale unstructured grid hurricane storm surge model applied to southern Louisiana. *Monthly Weather Review*, 136:833–864, 2008.

[127] Jeremiah J Wilke, David S Hollman, Nicole Lemaster Slattengren, Hemanth Kolla, Francesco Rizzi, Keita Teranishi, Janine Camille Bennett, and Robert L. Clay. The DARMA approach to asynchronous many-task (amt) programming. In *JOWOG Programming Models and Co-Design Meeting (Presentation)*, February 2016.

[128] Niklas Wintermeyer, Andrew R. Winters, Gregor J. Gassner, and Timothy Warburton. An entropy stable discontinuous Galerkin method for the shallow water equations on curvilinear meshes with wet/dry fronts accelerated by GPUs. *Journal of Computational Physics*, 375:447 – 480, 2018.

[129] D. Wirasaet, S. Brus, C.E. Michoski, E.J. Kubatko, and J.J. Westerink. Artificial boundary layers in discontinuous Galerkin solutions to shallow water equations in channels. *Journal of Computational Physics*, 299:597–612, 2015.

[130] D Wirasaet, EJ Kubatko, CE Michoski, S Tanaka, JJ Westerink, and C Dawson. Discontinuous Galerkin methods with nodal and hybrid modal/nodal triangular, quadrilateral, and polygonal elements for nonlinear shallow water flow. *Computer Methods in Applied Mechanics and Engineering*, 270:113–149, 2014.

[131] Yulong Xing and Xiangxiong Zhang. Positivity-preserving well-balanced discontinuous Galerkin methods for the shallow water equations on unstructured triangular meshes. *Journal of Scientific Computing*, 57(1):19–41, October 2013.

[132] Deli Zhang, Jeremiah Wilke, Gilbert Hendry, and Damian Dechev. Validating the simulation of large-scale parallel applications using statistical characteristics. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 1(1):3:1–3:22, February 2016.

[133] Gengbin Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[134] Gengbin Zheng, Michael S. Breitenfeld, Hari Govind, Philippe Geubelle, and Laxmikant V. Kale. Automatic dynamic load balancing for a crack propagation application. Technical Report 06-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, June 2006.

[135] Gengbin Zheng, Esteban Meneses, Abhinav Bhatelé, and Laxmikant V. Kalé. Hierarchical load balancing for Charm++ applications on large supercomputers. In *Proceedings of the 39th International Conference on Parallel Processing Workshops*, ICPPW, pages 436–444, 2010.