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

Maximilian H Bremer
University of Texas, Austin, TX
Lawrence Berkeley National Laboratory, Berkeley, CA
max@ices.utexas.edu

John D Bachan
Lawrence Berkeley National Laboratory, Berkeley, CA
jdbachan@lbl.gov

Cy P Chan
Lawrence Berkeley National Laboratory, Berkeley, CA
cychan@lbl.gov

ABSTRACT
The performance of hurricane storm surge simulations is critical to forecast and mitigate the deadly effects of hurricane landfall. Supercomputers play a key role to run these simulations quickly; however, disruptive changes in future computer architectures will require adapting simulators to maintain high performance, such as increasing asynchrony and improving load balance. We introduce two new multi-constraint, fully asynchronous load balancers and a new discrete-event simulator (DGSim) that is able to natively model the execution of task-based hurricane simulations based on efficient one-sided, active message-based communication protocols. We calibrate and validate DGSim, use it to compare the algorithms’ load balancing capabilities and task migration costs under many parameterizations, saving of over 5,000x core-hours compared to running the application code directly. Our load balancing algorithms achieve a performance improvement of up to 56 percent over the original static balancer and up to 97 percent of the optimal speed-up.

ACM Reference format:
DOI: 10.1145/nnnnnn.nnnnnnn

1 INTRODUCTION
Since 1980, seven out of the ten most costly US climate disasters were hurricanes, with Hurricane Katrina being the most expensive[?]. Hurricanes Harvey, Maria, and Irma, which occurred in 2017, are expected to be 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[?]. As a result, asynchronous task-based programming has been of great interest due to its potential to express increased software parallelism, introduce more flexible load balancing capabilities, and hide the cost of communication through task over-decomposition[?]. These programming models decouple the specification of the algorithm from the task scheduling mechanism, which determines where and when each task may execute and orchesstrates 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. However, transitioning scientific applications from their current synchronous implementations to asynchronous, task-based programming models requires a very high software engineering effort. Additionally, load balancing the execution of the resulting task graph remains a challenge due to the underlying computational hardness of optimal task scheduling.

This paper 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
The main contributions of this paper 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 repartitioning 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 paper is as follows. Section 2 presents related work. In Section 3, 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 6 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 7, we present our model validation and experimental results, demonstrating the viability of these load balancing approaches.

2 RELATED WORK

In adaptive mesh refinement (AMR) codes and $hp$-adaptive finite elements, dynamic load balancing typically relies one of three approaches: (1) graph partitioning using space filling curves, known as geometric partitioning [1]; (2) graph partitioning algorithms, such as those provided in the METIS and SCOTCH libraries [2]; or (3) diffusion or refinement based approaches [3]. 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 [4]. 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 [5]. 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 [6], hierarchical partitioning [7], and gossip protocols [8].

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. [9]). For example, the SST-macro simulator [10] 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 [11]. Evaluation of various load balancing strategies using discrete event simulation has been conducted in [12], and the use of particular asynchronous load balancing algorithms has been discussed in [13]. However, [14] examines only a simple greedy algorithm that ignores communication costs, and [15]...
computations entirely locally, and \( S \) corresponds to the area kernel, which requires computation at the element interface. The system of differential equations is then temporally discretized using a Runge-Kutta (RK) method.

The above numerical model is parallelized by distributing elements across a set of concurrent processes (referred to as ranks) that cooperate by message passing. In (1), both the update kernel and the area kernel can be computed entirely locally (independent of other elements). The edge 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, this results in a task dependency graph that exposes a large amount of task parallelism and asynchrony.

One of the key aspects of a storm surge code is its ability to simulate inundation. Due to numerical artifacts, 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 limiter is applied after the update kernel. We use the limiter proposed in [7], 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.

4 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
As there is a limitation associated with the amount of fine-grain parallelism we can expose due to scheduling overhead, developed a model to estimate the time to execute each task. Since DGSim utilizes a skeletonized form of the kernel, we explicitly enter or exit a load balancing phase. Thus we have no need to concurrently with task execution. Therefore, the scheduling logic to handle 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) [7]. 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 visits, 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.

5 PERFORMANCE MODEL CALIBRATION

5.1 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 \( T \), that consists of the elements \( \{ \Omega_e \} \), we approximate the execution time to advance \( T \) by one RK stage, \( t_T \) by \( t_T = \sum_{\Omega_e} \omega_e \tau_e \) where \( \omega_e \) is 1 if the element is wet and 0 if it is dry, and \( \tau_e \) 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 \( \tau_e \) 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 paper. 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 computes a 10th of the total timesteps. This conservatively approximates the available work to hide the load balancing costs. The change in wet/dry fraction is scaled in a consistent manner to ensure that the entire hurricane is simulated. 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.

5.2 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.

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 [7]. Table 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). Since DGSim provides a dedicated communication thread per socket, the QuickPath and Aries endpoints are shared by only one or two threads, and thus would see minimal contention.

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 bandwidth on Edison is 103 GiB/s [10], 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 [10]. Since no worker threads may send or receive messages, the dedicated communications thread can use the full unidirectional 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 [10]. 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 7.1 presents a validation of our performance model, comparing our modeled execution time versus empirically observed execution time of a skeletonized DGSWEM implementation.

### 6 BALANCERS

#### 6.1 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 5, 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 (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 [11].

Each tile 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 introduce the assignment variables, $\chi_{ik}$, where

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

and $\chi_{ik}$ is subject to the following constraints:

$$\sum_{k=1}^{n_{\text{max}}} \chi_{ik} = 1 \quad \forall i = 1, \ldots, n_{\text{tiles}}.$$  \hspace{2cm} (2)

Ideally, we would solve for time-dependent values of $\chi_{ik}$ 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

$$T = \max_k \sum_{i=1}^{n_{\text{tiles}}} t_i \chi_{ik}.$$  \hspace{2cm} (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:

$$\sum_{i=1}^{n_{\text{tiles}}} m_i \chi_{ik} \leq M_k \quad \forall k = 1, \ldots, n_{\text{ranks}}.$$  \hspace{2cm} (4)

Our optimization problem is defined by (2), (3), and (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 (4) and the execution time (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.

6.1.1 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.

6.1.2 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 and rank models).

The nested nature of these models is represented in Figure 1. These models provide representations of the simulation upon which the load balancers make relocation decisions.

---

**Fig. 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.

**Listing 1**: Asynchronous Diffusion Tile Model

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

### 6.2 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 [? ]. 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 [? ]. Note that balancing the memory and load constraints and balancing the wet and dry elements are equivalent formulations of the same load balancing problem.

### 6.3 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 procedure used in [? ]. Specifically, our asynchronous diffusion approach includes no world model, but we include tile models and rank models outlined in Listings 1 and 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 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 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 that stealing heuristic may lead to chatter, which is when this stealing heuristic may lead to chatter, which is when the ranks have been assigned new IDs. This approach can have a significant impact on the number of tiles migrated: for

### 6.4 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 [?]. 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 re locates, 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 [?]. 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 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.

7 NUMERICAL EXPERIMENTS

7.1 Empirical Validation of DGSim

To ensure that our simulator is accurate, we compared the DGsim execution times for the Storm36 simulation to a skeletonized version of DGSWEM run on NERSC’s Edison. The skeletonized DGSWEM implements the programming model outlined in Section 4. Inter-process communication is achieved through one-sided asynchronous remote procedure calls using UPC++ [7], 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 messaging and threading overhead models as described in Section 5.1 are reasonable. The parallel efficiencies for the two simulations are shown in Figure 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. [7].

7.2 Load Balance Comparison

In order to compare the load balancing algorithms outlined in Section 6, we compare performance for the hurricane used in the previous subsection using a fixed machine and algorithmic configuration outlined in Table 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 metrics: 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 $I = \frac{\text{max}_{r} T_{r}}{\bar{T}}$, where $\max T$ is maximum load on a given rank and $\bar{T}$ is the average load across the system. The combined performance results are shown in Figure 3 and the elapsed times and speed-ups in Table 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 3 has a high imbalance. This is reflected in Figure 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. 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 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 4d and 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.

<table>
<thead>
<tr>
<th>Machine</th>
<th>DGSWEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMA domains/node</td>
<td>2</td>
</tr>
<tr>
<td>Threads/NUMA domain</td>
<td>12</td>
</tr>
<tr>
<td>CPU clock-speed</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>Asynchronous Diffusion</td>
<td>40 ms</td>
</tr>
<tr>
<td>Rebalance Frequency</td>
<td>5 s</td>
</tr>
</tbody>
</table>

Table 2: DGSIm parameters for load balance comparison.
Fig. 3: Load imbalance for Storm 36 using 1200 cores with the configuration outlined in Table 2. The following load balancing strategies were evaluated: static (ST), multi-constraint static (MCS), asynchronous diffusion (AD), and semi-static (SS).

<table>
<thead>
<tr>
<th>Cores</th>
<th>( T_{ST} )</th>
<th>( T_{MC} )</th>
<th>( S_{ST} )</th>
<th>( T_{AD} )</th>
<th>( T_{MC} )</th>
<th>( S_{ST} )</th>
<th>( S_{MC} )</th>
<th>( T_{SS} )</th>
<th>( T_{MC} )</th>
<th>( S_{ST} )</th>
<th>( S_{MC} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>2.613</td>
<td>2.048</td>
<td>1.28</td>
<td>1.686</td>
<td>3.654</td>
<td>1.55</td>
<td>1.21</td>
<td>1.675</td>
<td>723.109</td>
<td>2.325</td>
<td>1.56</td>
</tr>
<tr>
<td>2400</td>
<td>1.310</td>
<td>1.058</td>
<td>1.24</td>
<td>0.849</td>
<td>9.532</td>
<td>1.89</td>
<td>1.54</td>
<td>1.25</td>
<td>856.694</td>
<td>557.000</td>
<td>1.53</td>
</tr>
<tr>
<td>3600</td>
<td>0.876</td>
<td>0.698</td>
<td>1.26</td>
<td>0.571</td>
<td>16.614</td>
<td>1.39</td>
<td>1.53</td>
<td>1.22</td>
<td>585.491</td>
<td>568.000</td>
<td>1.50</td>
</tr>
<tr>
<td>4800</td>
<td>0.659</td>
<td>0.553</td>
<td>1.19</td>
<td>0.431</td>
<td>24.402</td>
<td>1.39</td>
<td>1.53</td>
<td>1.28</td>
<td>452.593</td>
<td>3.932</td>
<td>1.46</td>
</tr>
<tr>
<td>6000</td>
<td>0.528</td>
<td>0.444</td>
<td>1.19</td>
<td>0.348</td>
<td>36.955</td>
<td>1.52</td>
<td>1.28</td>
<td>373.574</td>
<td>1.255</td>
<td>1.41</td>
<td>1.19</td>
</tr>
</tbody>
</table>

Table 3: Performance of load balancers for Storm36. 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 \( \sigma_{TM} \) for a sample size of 5.

Fig. 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.

7.3 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, the parallel efficiencies for the four partitioning strategies are shown in Figure 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; \( E_{MC} / E_{ST} = 92.1\% \). As the number of cores increases each
The asynchronous diffusion load balancing scales the best with \( E_{\text{6000}} / E_{\text{1200}} = 96.9\% \). 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.

**8 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 the most well, 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.

**ACKNOWLEDGMENTS**

Maximilian Bremer was funded by the U.S. Department of Energy’s Computational Science Graduate Fellowship, Grant DE-FG02-97ER25308. This manuscript has been authored by authors at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. Furthermore, this research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.
A ARTIFACT DESCRIPTION:
SEMI-STATIC AND DYNAMIC LOAD BALANCING FOR ASYNCHRONOUS HURRICANE STORM SURGE SIMULATIONS

A.1 Abstract
This artifact contains descriptions of the discrete event simulator used to produce all of the results shown in the paper. Additionally described are the input data sets as well as the post-processing steps.

A.2 Description
A.2.1 Check-list (artifact meta information).
- Algorithm: Discrete Event Simulation
- Program: C++, MATLAB
- Compilation: GNU C++11 compiler
- Data set: Formatted text files
- Run-time environment: Shared partition on NERSC’s Edison
- Execution: Serial
- Output: Formatted text files
- Experiment workflow: Modify parameters; compile; submit
- Publicly available?: No

A.2.2 How software can be obtained (if available).
Software is not yet open-source.

A.2.3 Hardware dependencies.

A.2.4 Software dependencies. In order to compile the code, we require a gnu C++ compiler that supports the C++11 standard. Versions that are known to work are:
- g++ (Ubuntu 4.8.4-2ubuntu1 14.04.3) 4.8.4
- g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-11)
- g++ (GCC) 6.1.0 20160427 (Cray Inc.)

For post-processing, we require MATLAB. In this paper, version 9.0.0.341360 (R2016a) was used.

A.2.5 Datasets. There are two main input data files: an ADCIRC-formatted finite element mesh, and a custom generated file that stores the simulations wet/dry status.

A.3 Installation
N/A

A.4 Experiment workflow
The machine and problem parameters are a combination of hard coded constants and environment variables. Compile time parameters should be set before compile time. Additionally, compiler directives are used to select load balancing strategies. For example to run the semi-static strategy, we would compile
flags=-DSEMISTATIC make exe/dgswem.o3

A detailed overview of compiler directives and environment variables can be found in the repository’s README.md.

A.5 Evaluation and expected result
The basic meta-data such as simulation time and numbers of tiles moved is written to the standard output. Compute profiling data such as how much time was spent, computing, sending, and idling is written to the timer/ directory. This data is processed via matlab, by executing the st36_proc.m matlab script located in scripts/visualization.

Load imbalance is stored in the diagnostics folder diag/. This data is processed via the MATLAB script, diag_proc.m.

Due to the self-timing procedure for determining scheduling overhead, the simulation results are non-deterministic. Therefore, it is important to run several each simulation several times and present measurement variation.