SWE-X10: Simulating shallow water waves with lazy activation of patches using ActorX10

Alexander Pöppl and Michael Bader
Technical University of Munich
Boltzmannstraße 3, 85748 Garching bei München, Germany
Email: {poeppl, bader}@in.tum.de

Tobias Schwarzer and Michael Gläß
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Cauerstraße 11, 91058 Erlangen, Germany
Email: {tobias.schwarzer, michael.glass}@fau.de

Abstract—We present an efficient Finite Volume solver for the shallow water equations using an actor extension of the X10 programming language, ActorX10, as programming model. Each actor is assigned to a Cartesian patch of the computational grid. Using the actor’s finite state machine to control patch updates, we realize lazy activation of patches, only when a propagating wave enters the respective patch. Overlapping of communication and computation in the fully non-central actor-based control, as well as careful optimization (esp. vectorization) of kernels leads to high performance and parallel efficiency in shared and distributed memory. Benefits of lazy activation are demonstrated via reduced CPU hours for a benchmark scenario.

Index Terms—High performance computing, parallel programming, performance analysis, computer languages

I. INTRODUCTION

Assigning dynamic work load to heterogeneous and dynamically changing hardware is likely to evolve into a key challenge in scientific computing. Modern architectures evolve towards many-core designs, where accelerator techniques (consider GPGPUs or Xeon Phi co-processors), dynamic frequency scaling and temporary selective use (or shut-down) of parts of the CPU (“dark silicon”) characterize dynamic availability of resources. Similar, large-scale applications need to dynamically invest degrees of freedom where they lead to the largest gain in accuracy, leading to dynamic requirements of resources on the application side. Nevertheless, the predominant programming models in scientific computing primarily target a fixed set of resources – typically a given number of compute nodes – which is to be used as efficiently as possible.

In this paper, we present the proxy application SWE-X10 (earlier version shown in the extended abstract [13]), which follows an actor approach towards more dynamic use of resources. We show how finite state machines, as part of an actor-oriented parallel programming model, can be used to identify parts of the computational domain that are not updated: following a “lazy activation” paradigm, respective compute resources could remain idle until needed. In addition, the actor approach allows Cartesian grid patches to propagate in time without global control. The implementation is based on the APGAS language X10 and uses the actor extension ActorX10 [14]. We demonstrate the general feasibility of “lazy activation” and evaluate the parallel efficiency of the X10 implementation.

While the implementation is currently limited to uniform Cartesian grids, we envisage a block-adaptive version as the next step, where multilevel grids with different resolutions are used. Lazy activation would then be used for fine-grid levels. The actor approach also enables the use of automatic methods for design-time characterization, as known from embedded computing: Using fast analysis techniques or short sample simulations, such approaches not only allow for parameter optimization wrt. various parameters such as patch sizes, distributions, or load balancing, but also explore the different trade-offs between performance, the number of used components, energy consumption etc.

In the following, we first introduce the shallow water proxy application SWE-X10 (section II) and provide details on the actor model implemented in ActorX10 (section III). In section IV, we discuss the actor implementation and performance issues of SWE-X10. Section V evaluates parallel performance and potential gains from lazy activation.

II. MATHEMATICAL BACKGROUND

In this work, we focus on modeling based on hyperbolic partial differential equations (PDEs), which are characterized by their “wave-type” solutions that propagate signals at finite propagation speed. Initial disturbances in parts of the domain will not effect the rest of the domain (beyond an “event horizon”) immediately – an effect we will utilize later to avoid the computation of zero-updates. As an example for a particular hyperbolic PDE system, we focus on the shallow water equations.

A. The Shallow Water Equations

The shallow water equations are a nonlinear system of conservation laws for depth and momentum that is commonly used to simulate, for example, the propagation of tsunamis, as well as, with additional source terms, the inundation of coastal regions [11]. They can be derived from the three-
dimensional Navier-Stokes equations by averaging over the depth dimension. In its two-dimensional form,

\[
\begin{bmatrix}
    h \\
    hu \\
    hv
\end{bmatrix}_t + \begin{bmatrix}
    hu \\
    hu^2 + \frac{1}{2}gh^2 \\
    hv \\
    hvu \\
    hv^2 + \frac{1}{2}gh^2
\end{bmatrix}_x + \begin{bmatrix}
    hv \\
    hvv \\
    hu \\
    huu \\
    hv \\
    hvv
\end{bmatrix}_y = \begin{bmatrix}
    -ghB_x \\
    -ghB_y
\end{bmatrix},
\]

(1)

\(h(x, y, t)\) is the height of the water column at position \((x, y)\) and time \(t\); the velocities in the two spatial dimensions are described by \(u(x, y, t)\) and \(v(x, y, t)\), and \(g\) is the gravitational constant. Source terms on the right-hand side model additional effects, such as the influence of terrain above (during inundation) and below sea level. The latter, i.e., the elevation of the ocean floor, is often referred to as bathymetry data, here denoted as \(B(x, y)\).

B. The Finite Volume Model

The numerical model used for SWE-X10 is based on the SWE package, a hybrid OpenMP/MPI proxy application written in C++ [4]. Both packages follow a classical Finite Volume approach, as suggested by LeVeque et al. [11]. The simulation domain is discretized on an equidistant Cartesian grid. Each cell \(i, j\) stores a vector of cell-averaged quantities, \(Q_{i,j}^{(n)} = [h_{i,j}^{(n)}, \langle hu \rangle_{i,j}^{(n)}, \langle hv \rangle_{i,j}^{(n)}, b_{i,j}^{(n)}]\), at time step \(t_n\). Starting at initial time \(t_0\), the unknowns are successively updated according to the following update scheme:

\[
Q_{i,j}^{(n+1)} = Q_{i,j}^{(n)} - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-rac{1}{2},j}^{(n)} + A^- \Delta Q_{i+rac{1}{2},j}^{(n)} \right) - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-rac{1}{2}}^{(n)} + B^- \Delta Q_{i,j+rac{1}{2}}^{(n)} \right),
\]

(2)

where \(\Delta t\) is the time step size, and \(\Delta x\) and \(\Delta y\) denote the cell size in \(x\) and \(y\) direction.

\(A^\pm \Delta Q_{i \pm \frac{1}{2},j}^{(n)}\) and \(B^\pm \Delta Q_{i,j \pm \frac{1}{2}}^{(n)}\) are so-called net updates computed for the horizontal and vertical cell edges, respectively. They are obtained from the approximate solution of respective Riemann problems at the edges, and reflect the transport of mass and momentum between adjacent cells during a single time step. SWE-X10 supports two different approximate solvers: the \(f\)-Wave [3] solver is a relatively simple solver that produces good results for wave propagation across the open ocean. It is not, however, able to model inundation, and thus treats coast lines as a wall boundary. The HLLE solver [5, 8] handles inundation at the cost of a moderate increase in computational complexity. Both implementations have been tailored to allow auto-vectorization by the compiler.

The finite propagation speed of an initial perturbation is reflected in the update scheme (Equation 2), as it only considers the four direct neighbors of a cell. Propagation is therefore limited to one cell size per time step, which poses a restriction to the time step size \(\Delta t\) (the well-known CFL condition). In typical implementations, cells are updated regardless of a perturbation in their neighboring cells, which leads to a large number of “trivial” zero-updates that use computing resources unnecessarily.

### III. ACTOR MODEL

The actor model is a computational model first described by Hewitt [9] (in the context of artificial intelligence) and later formalized by Agha [1]. Essentially, an actor may be described as a thread of execution with a state attached to it. Actors execute their code independently and without explicit synchronization. Collaboration between actors may be attained by explicit messages through statically defined communication channels, or at the start of the program execution. This structure allows for a very low amount of coupling. As communication is handled by the actor framework, actors only need to specify their communication interfaces, and how to react to data being available to them. The concrete implementation of the actors is encapsulated from the other actors in the actor graph.

SWE-X10 uses the actor model where control is realized by means of Finite State Machines (FSMs) – resembling the FunState [15] Model-of-Computation – to implement a distributed and locally coordinated time stepping scheme. In the following section we will give a quick introduction of the actor model as implemented in the ActorX10 actor library [14], which was used to implement SWE-X10.

A. Components

The basic building blocks of the actor-based computational model are channels, ports, and actors. These components are arranged in the actor graph. In the following, we give a short definition and outline of each of these components’ functionalities.

Actors may be described abstractly by the tuple \(a = (I, O, F, R)\) containing the set of incoming ports \(I\) for receiving data, the set of outgoing ports \(O\) for sending data, and the set of functions \(F\) callable in the course of the firing of the actor’s finite state machine \(R\). The state machine \(R = (Q, q_0, T)\) consists of a finite set of states \(Q\), an initial state \(q_0 \in Q\) and a set of transitions \(T\) between these states. It is called whenever there is a change in one of the actor’s ports, which effectively leads to an inversion of control. The application developer only specifies how to react to the changes, but has no control over their timing or their sequence of their arrival. The order in which events are handled may be manipulated using states in the state machine. Each transition also defines guard functions that determine whether a transition is triggered, and the functions that will be executed when the transition is triggered. Finally, the number of data elements required to perform the transition and the necessary space needed to store the resulting output in the corresponding outgoing ports is specified.

Ports are the typed facades the actors utilize for their communications. Each port has a type describing the kind of data that may be sent or received through it, an actor it belongs to and a channel to which it is assigned dynamically at application run time. Incoming ports offer facilities to query for the availability of data or to read one or more elements from it. Outgoing ports offer to query the channel’s remaining...
public class SimulationActor extends Actor {
    public val leftIn = newInPort[Data]("left");
    public val rightIn = newInPort[Data]("right");
    public val leftOut = newOutPort[Data]("left");
    public val rightOut = newOutPort[Data]("right");
    public def this(name: String) {
        super(name);
    }
    public def act() {
        if (leftIn() && rightIn()) {
            && leftOut() && rightOut()) {
                applyLeft(leftIn.read());
                applyRight(rightIn.read());
                performTimestep();
                leftOut.write(getLeftUpdate());
                rightOut.write(getRightUpdate());
            }
        }
    }
    public static def main(Rail[String]) {
        val ag = new ActorGraph();
        val actor1 = new SimulationActor();
        val actor2 = new SimulationActor();
        val actor3 = new SimulationActor();
        ag.addActor(actor1);
        ag.addActor(actor2);
        ag.addActor(actor3);
        ag.connectPorts(actor1.leftOut, actor3.rightIn);
        ag.connectPorts(actor1.rightOut, actor2.leftIn);
        ag.connectPorts(actor2.rightOut, actor3.leftIn);
        ag.connectPorts(actor3.leftOut, actor2.rightIn);
        ag.moveActor(actor1, Place(1));
        ag.moveActor(actor1, Place(2));
        ag.start();
    }
}

Fig. 1. Sample implementation of an Actor class.

Hence, moveActor[GlobalRef[Actor], Place] is called to move them to the desired place.

C. Towards Performance Modeling

As briefly outlined, actor-based programming inherently facilitates a specific Model-of-Computation – in case of ActorX10, it is FunState [15]. Besides run-time advantages of actors (such as parallelism and the feasibility of self-scheduled systems), important design-time opportunities arise: Separation of control-flow and data-flow together with the explicit communication enable an – even automatic – creation of performance models. In the embedded computing community, aspects such as the boundedness of buffers for the exchange of data [17] or verification aspects such as deadlock-freeness [18] are established techniques. In our work, we intend to exploit the Model-of-Computation for a performance model that not only focuses on the data manipulation in the actions of the actor – a step well-understood in the HPC community – but especially on the aspect of self-scheduling, lazy activation and the communication of the actors. Since all data exchanges are explicitly captured in the actor model (also within the actors via the FSMs), a performance model may better capture overlapping of data transfer and calculation steps. In particular, such information allows to optimize actor distributions and load-balancing schemes, since interferences due to communication and enhanced parallelism due to lazy activation and decentralized scheduling now become tractable. Even automatic optimization approaches can be employed, well-known as design space exploration [6], [12] in the embedded computing domain.

IV. IMPLEMENTATION

Current computer architectures rely heavily on parallelism on multiple layers (SIMD registers, multiple cores and nodes) to achieve peak performance. To fully utilize these architectures, SWE-X10 reflects these layers following the APGAS...
principle of the X10 programming language. To exploit coarse-grained parallelism that comes with having multiple nodes with several multi-core CPUs, we partition the Cartesian discretization grid into equally sized, rectangular patches. The finite volume scheme outlined in subsection II-B updates cells by calculating net-updates of the conserved quantities for all cell edges. The necessary data – the outermost cell of the neighboring patches – are provided through ghost layers. In SWE-X10, the task of updating ghost layers, and therefore the coordination of the simulation is handled by actors. For the coordination, there is a one-to-one mapping from actors to patches, with every actor coordinating the sequential computation of updates within its patch. Each actor is connected to the actors managing the patches adjacent to the actor’s own patch.

In Figure 3, this behavior is illustrated via an actor graph with $3 \times 3$ actors. The connection between two actors is signified by a single edge. This connection is realized by four channels in the actual implementation, one pair of channels (incoming and outgoing) to exchange simulation data and another pair to exchange control messages, such as termination notices. In Figure 4, the same actor graph is shown with all channels and their capacities visible. The capacity of the channel needs to be at least two to avoid deadlocks due to non-consumed tokens. It does not have to be larger either due to the circular dependencies between the individual actors.

### A. Coordination via an actor-based run loop

SWE-X10 follows the execution model imposed by the ActorX10 library. For regular time steps, we follow the model introduced in subsection III-B. As shown above, the actor is connected using two channels, one for incoming and one for outgoing data. Coordination is also similar: Every incoming border triggers the act method, but the update will only be triggered once all necessary updates are available and there is enough space in the channels to which the updates are written. When no updates are needed, for example if none of an actor’s neighbors is active, the method will trigger itself, until one of the neighbors becomes active.

As already explained, the simulated solution will remain in its initial steady state until the propagating wave initiated by an initial disturbance will arrive. We avoid superfluous computations on respective patches by assigning each actor an attribute that stores its activity status. Initially, the status of each actor is set based on the scenario’s initial condition. In many cases, the initial perturbation of the steady state will be restricted to a few patches, while the rest of the domain remains at rest.

Figure 6 gives an example for the FSM-based coordination. As a first step, all actors in the simulation send their activity status to their neighbors using the control channels (sendActivation()). Actors that contain part of this initial perturbation are set to the state propagating-wave, while the rest is set to lake-at-rest. Then, active actors perform the simulation steps (computeStep()) once they receive all necessary updates.
necessary updates (recvActive()). During computation of the update, the patch determines for each of its copy layers whether the update actually changes any values. If changes occurred or if the neighboring actor is already active, the updated layer will be sent. If the neighbor is still inactive, the actor will also send a control message stating that updates are now available (sendActivation()). Upon receiving the message (recvActivation()), the neighbor in the lake-at-rest state will set itself to propagating-wave and send its new activity status to all neighbors (sendStatus()), so that other neighbors that are already active can also start sending updates to the newly active actor. Finally, once an actor has reached the termination condition, it will send a termination signal (sendTerm()) to other actors, which will be propagated until no more actors are active (recvTerm()).

In future work, we plan to implement block adaptivity and, as related functionality, a (multi-rate) local time-stepping scheme. Both will make use of the FSM semantics provided by the actor model. For example, when an actor is communicating with another actor running with only half of the time step size (due to higher resolution, e.g.), then it may use two separate states to deal with ghost layer data: one state for steps where a communication is needed and another state when the previously received data needs to be interpolated. Block adaptivity will especially make load balancing more complicated. Actors with a highly refined grid will have a significantly higher computational load. Hence, we will either need to allow splitting of actors, and allow actors to have more than one neighbor in each direction. Or we could make the assignment of actors to patches and cores more flexible and, for example, assign several cores (maybe even on accelerators) to the computation of refined patches.

B. Performance considerations

The translation of X10 to machine code is a two-step process. First, X10 code is compiled down to GNU C++. Then, the generated C++ code is translated down to machine instructions. This two-step process and the research character of X10 lead to a number of interesting performance pitfalls. Some of them, such as a lack of an available expression analysis, can be traced back to the two-step compilation process [10]. In the X10 compiler, the implementers focus on high-level optimizations specific to language features of X10, and leave the more standard optimizations to be done by the C++ compiler on the generated code. However, the generated C++ code may be too complicated for the compiler to recognize optimizable patterns. For example, the X10 array access `val b = a(3)` translates to:

```
1: x10_double b = __extension__ {{
  2:   x10_double ret6171;
  3:   ret6171 = (a->FMGL(raw))->apply({(x10_int)3});
  4:   ret6171;
  5: }}
```

All of these function calls are inlined by the C++ compiler at a later stage of the compilation, however, optimization steps executed prior to inlining will disregard the array access due to the complicated structure of the access¹ [7], [10]. This is especially problematic in an HPC context. Auto-vectorization, for example, relies on the compiler to find code sections where loops may be transformed to exploit SIMD instructions. Compilers rely on certain dependency patterns to infer that a vectorization of the code is possible. In our tests, we found that the hot spots of our application, the solution of the edge-local Riemann problems on the unknown arrays was not vectorized by the compiler, though previous work showed successful auto-vectorization using C++ [2].

Another, related problem are heap allocations in hot spots of the simulation. In C/C++, one can allocate arrays on the stack (e.g. `float a[4]`) to store intermediate values in more extensive computations. In X10, the most primitive array type is the class Rail, a one-dimensional, zero-based, fixed-size storage for instances of a type. As class instances, they are typically heap-allocated. A naive port of the Riemann solver to X10 with Rail objects causes the overall runtime of the application to be dominated by the heap allocations made during the execution of the solver. There is an annotation in X10 that guides the compiler to allocate the object on the stack instead, but we found that this did not improve the execution time by much. We suspect secondary allocations to be responsible for that behavior. One solution proposed before is to split small arrays of fixed size up into single variables to eliminate the stack allocations. While this technique reduced readability of the code, performance was greatly improved.

In SW-E-X10, we decided to integrate native C++ code for the computational hot-spots of the simulation. Using the `@NativeRep` annotation, it is possible to skip the code generation process for an X10 class and to provide another implementation that matches the generated interface of the original X10 class instead. That alternative implementation uses the

¹The optimization of function calls is complicated in C++, as the code of the function may not be available to the compiler, and it is difficult to rule out side effects.
dimensions provided by the \texttt{xl0.regionarray\textunderscore Array} object to iterate over the raw memory with a conventional loop nest and storing the unknowns of the simulation directly. We assisted the compiler to generate vectorized code via \#pragma simd annotations. Using state-of-the-art compilers (we used Intel C++ compiler 16.0 and Intel MPI), this is sufficient to obtain a more-than-4× speed-up over the non-vectorized, pure X10 version (see section V).

Another potential performance pitfall in the current implementation of X10 is the implicit capture of objects in \texttt{at 
<PLACE>\textunderscore}>)-statements. Whenever a place-shift occurs, all objects referenced in the body of the statement are serialized and copied to the other place. A common mistake that leads to performance degradation in this context is the implicit capture of \texttt{this}. This happens when a place-shift is performed in the body of an instance method and an instance member or another instance method of the object is accessed. In both cases, the whole object (maybe an entire patch) including the complete object graph that is reachable from it will be copied, serialized and sent to the other place [7].

ActorX10 may be used to avoid some of these pitfalls. Applications that comply with the ActorX10 execution model typically use the APGAS features of X10 for the application setup. During the computation, however, different parts of the application only communicate via the channels defined initially, and no direct access to other actors’ data is permitted. Instead, all data transfers are performed through explicit library calls. A capture of \texttt{this} is not possible, as, in contrast to place-shifts, the member access expression is evaluated prior to the evaluation of the method call. As long as the model is not violated, no implicit captures of \texttt{this} will happen during the execution of the actor-based computation [14].

V. RESULTS

We performed a battery of tests to evaluate the performance of SWE-X10. Initially, we focused on the single-core performance with and without the native, vectorized solver. Thereafter, we evaluated the differences in time-to-solution when actors are only activated gradually. Finally, we evaluate the scaling behavior of our application in strong- and weak-scaling tests.

All tests were performed on the MAC Cluster (http://www.mac.tum.de/wiki/index.php/MAC\_Cluster), a small cluster with (among others) 28 nodes equipped with two Intel Xeon E5-2670 CPUs (Sandy Bridge architecture) with a peak performance of 332.8 GFlop/s (single precision). The measured STREAM triad performance is 108.9 GB/s per node. Nodes are connected through InfiniBand QDR. We used the X10 Compiler in version 2.5.1 (MPI backend) and generated C++ code by Intel compiler 16.0.

A. Benefits of vectorization

The first test compares the floating point performance of pure X10 code against X10 code with native C++ code for the iteration over the unknowns of the domain, annotated with \#pragma simd. In both cases, we use the HLLE solver. We performed tests on a single CPU core, with four actors and varying grid sizes. The results are given in Figure 7. As outlined in the following sections, the SWE-X10 implementation is on par with the C++-performance of the original SWE code.

B. Single-node performance

In this section, we summarize the scaling behavior of SWE-X10 in terms of shared-memory parallelization, as it was previously reported in an extended abstract [13]. We performed a weak scaling test on a single node, and compared the results to SWE [2], [4], which is our corresponding reference implementation (using MPI+OpenMP and C++). The setup for the benchmark is a radial dam break. Each CPU core holds a region of 1024 × 1024 cells, which is distributed onto 4 actors with patches of size 512 × 512. We chose this configuration as it led to highest performance.

The performance for SWE-X10 and SWE for this test are shown in Figure 8. For SWE-X10, a single core reaches a performance of about 25 GFlop/s. The performance plateaus at about 75 GFlop/s from 10 cores, indicating a saturation of the memory bandwidth. SWE ends up performing at the same level, however, it only saturates at a configuration with 14 cores. In both cases, the codes manage to reach 23% of the node’s peak performance.
C. Multi-node performance

To evaluate the multi-node performance of SWE-X10, we performed another weak-scaling test (shown previously in the extended abstract [13]). In this test, each CPU (with eight cores) acted as one X10 place. Again, 4 actors with $512 \times 512$ cells were assigned to each core, resulting in a total of 32 actors per CPU. We executed the test for configurations ranging from one CPU up to sixteen nodes. The achieved performance is shown in Figure 9.

From one to eight nodes ($2^3$ to $2^7$ cores), the performance scales linearly, but at sixteen nodes, a sublinear result was observed. As the same performance degradation occurs with SWE, this may indicate a hardware issue.

In general, SWE-X10 performs better than SWE, which can be explained by the differing communication patterns used in the two software packages. In SWE-X10, we implemented the actor-oriented approach, which is inherently concurrent and enables overlapping of computation with communication. An actor may update its patch while another actor communicates and is waiting for data, e.g. In contrast, SWE uses a simple SPMD pattern, where computation and communication happen in phases, without overlap.

We conclude that the actor implementation in SWE-X10 allows favorable speed-up in shared and distributed memory, competitive to MPI+OpenMP implementations.

D. Lazy activation of actors

In the following, we evaluate the benefit of lazy activation of patches using our actor approach. We cannot illustrate this benefit via reduced time-to-solution, yet, as this would require either load balancing among actors or migration of actors (incl. their patches) during run-time. At the moment, actors are created and distributed to CPU cores at program-start. Migrating them to different cores or nodes is not possible yet. An alternative would be to migrate patches and assign them to actors on different cores or nodes. While this would reflect standard practice in HPC, it would substantially complicate the actor model: in particular, communication structures (i.e., channels between actors) would need to change dynamically. Such functionality is not yet implemented in ActorX10, and it will be part of our future research to evaluate the feasibility of such an approach.

Hence, instead of time-to-solution, we measured the benefit of lazy evaluation by determining the CPU time ("CPU hours"), i.e. by summing up the time spent computing updates on each CPU. This metric assumes that CPUs of inactive actors can remain (or be set to) an idle state, and allows a comparison of the amount of work (roughly reflecting the required energy) that is performed during program execution. In a resource-aware environment, it may be possible to activate CPU cores only when calculations actually commence.

As a test setup, we again chose a radial dam break scenario with the dam break in the lower left corner of the scenario. Boundary conditions are set to wall type, and the simulation time is set to 90 simulated seconds. We chose a grid resolution of $8192 \times 8192$ cells, distributed onto actors with patches of $512 \times 512$ cells each. The actors were distributed across eight CPUs on four nodes of the MAC cluster, according to the scheme displayed in Figure 10.

We compared the CPU time with and without lazy activation. For the run without lazy activation of patches, we obtained an overall execution time of 1,433 seconds. As all actors compute solutions from the start, and therefore all CPUs are used throughout the whole simulation, this sums up to 12,264 CPU seconds, or 3.41 CPU hours.

In comparison, the version with lazy activation takes 1,203 seconds to complete. To obtain the aggregate CPU time, we logged the initial activation of the first actor for each place, from which we obtained the activation times displayed in Figure V-D. Summing over all active times yields an overall CPU time of 6,741 CPU seconds, or 1.87 CPU hours. Hence, in terms of total CPU time spent, we obtained a significantly lower use of computing resources.

Note that the concrete gain of lazy activation of course depends heavily on the initial scenario. In cases where the initial disturbance is in the center of the domain, gains will
be significantly smaller. Nevertheless, typical tsunami simulation setups for example, will feature comparably local initial disturbances and (esp. for simulation of far-field tsunamis at ocean-scale) large “inactive” domains during the first phase of a simulation.

VI. CONCLUSION AND OUTLOOK

In this paper, we presented first results of an actor-based solver for the shallow water equations, SWE-X10. We demonstrated high node-level performance (due to auto-vectorization of “X10-native” loops) and solid scalability that benefits from overlapping of computation and communication, exploiting the actor model. Currently, SWE-X10 only makes limited use of the flexibility provided by the actor model; we thus aim to expand SWE-X10 in two key directions: (1) an expansion towards block-adaptive mesh refinement together with local time stepping, and (2) the characterization of performance and respective improved scheduling in heterogeneous environments.

When using block-adaptive meshes with local time stepping, as in [11], multiple layers of meshes with different resolutions are used. Here, lazy activation may be enforced on the fine-level meshes, and dynamic deactivation needs to be offered by the actors’ finite state machines. Any activation or deactivation of an actor then needs to trigger a load-balancing step: based on a performance characterization of actor activities (as outlined in subsection III-C), actors may be migrated to other X10 places. Also, dynamic splitting of heavy-weight actors or merging of several light-weight actors should be considered to improve the granularity of parallelism, similar to [16]. Finally, an important goal is to implement (local) time stepping schemes (incl. actor-based load balancing) without a global coordination scheme, which are known to easily cause scalability bottlenecks.

REFERENCES


ACKNOWLEDGEMENTS

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89).