A UPC++ Actor Library and Its Evaluation On a Shallow Water Proxy Application

Alexander Pöppl
Department of Informatics
Technical University of Munich
Munich, Germany
poeppl@in.tum.de

Scott Baden
Computational Research Division
Lawrence Berkeley National Laboratory
Department of Computer Science and Engineering
University of California, San Diego
baden@ucsd.edu

Michael Bader
Department of Informatics
Technical University of Munich
Munich, Germany
bader@in.tum.de

***Abstract*—Programmability is one of the key challenges of Exascale Computing. Using the actor model for distributed computations may be one solution. The actor model separates computation from communication while still enabling their overlap. Each actor possesses specified communication endpoints to publish and receive information. Computations are undertaken based on the data available on these channels. We present a library that implements this programming model using UPC++, a PGAS library, and evaluate three different parallelization strategies, one based on rank-sequential execution, one based on multiple threads in a rank, and one based on OpenMP tasks. In an evaluation of our library using shallow water proxy applications, our solution compares favorably against an earlier implementation based on X10, and a BSP-based approach.**

***Index Terms*—Actor-based computation, tsunami simulation, programming models, PGAS**

I. Introduction

With this work, we demonstrate the performance and usability benefits of using the actor model for classical HPC. We will introduce an actor model based on the FunState [1] approach, and its implementation as a library in UPC++. There, we will explore and evaluate three different parallelization strategies for the actor library. We apply the actor model to a tsunami simulation proxy application, and compare its performance against our prior application SWE-X10 based on actorX10, an X10 implementation of our actor library, and SWE, the original tsunami application using MPI and OpenMP with the BSP approach for parallelization. We show that our solution demonstrates significantly higher performance in a weak scaling test, and also a significantly better performance with a lower per-core computational load compared to SWE-X10. We also demonstrate a clear performance benefit compared to SWE.

II. Motivation and Related Work

The imminent arrival of exascale computing introduced the debate on how to program these machines so that they can offer their expected performance. Currently, many applications still follow the *Bulk Synchronous Parallel (BSP)* model, with clearly defined phases for computation, communication and synchronization. The most widely used approach here is to use MPI for inter-node communication and parallelization, and OpenMP for the on-node parallelization. The BSP approach enables a clear separation of concerns, but the structure, especially with the synchronization step at the end may be too rigid to obtain the best performance. As the number of nodes increases, so will the difficulty of maintaining the pure BSP model, and therefore the burden to the application programmer.

A promising model is the *Partitioned Global Address Space (PGAS)* programming model [2]. This model assumes a global address space, but exposes the separate physical address domains. This may ease the burden on the application programmers, as they no longer need to think about in terms of message-passing, but can access data on remote ranks directly. Another promising model is the *task-based* programming model [3]. Here, the programmer specifies pieces of computation and communication as tasks, and also their dependencies. Afterwards, the resulting task graph is handed to a scheduling system that schedules them onto available computing resources. This model has been implemented in OpenMP [4] and also in runtime systems, for example in StarPU, which enables distributed task scheduling onto heterogeneous machines [5], or the AllScale project [6], which aims to separate the specification of parallelism from its low-level management on the target hardware. Task-based parallelism has been employed successfully in complex applications, for example in the Uintah application framework [7].

In the Invasive Computing project[1], we investigate novel approaches to use future, parallel and heterogeneous computers [8]. Most of the research is focused around the project's own hardware architecture, a cache-incoherent heterogeneous Multiprocessor System-on-Chip (MPSoC). This architecture features multiple smaller groups of CPU cores (called tiles) that share a cache hierarchy and a memory. The different tiles are connected using a Network-on-Chip. There are different types of tiles, such as tiles containing normal CPU

[1] http://www.invasic.de

cores, accelerators, and IO tiles. Applications are written in a modified version of X10, a language that implements the Asynchronous Partitioned Global Address Space (APGAS) model. We worked on applications from HPC with an invasive multigrid application [9] and an elastic MPI runtime [10]. Most target applications for this architecture, however, are from the domain of embedded systems. Here, it is important to have predictable application behavior and performance, especially with respect to hard real time scenarios. A popular approach to programming parallel application in this domain is the actor model.

The actor model relies on active objects, referred to as actors, to propel a computation. Actors execute their functionality concurrently, based on the data available to them. There is no direct data sharing between the different actors; they are connected through channels with FiFo semantics. An actor may perform a computation when the state of the channels changes, i.e., when an actor at the other end of a channel inserts or removes a token. The actor model was originally proposed by Hewitt [11] as a concept to facilitate programming for artificial intelligence. Agha [12] provided the first formal notation of the computational model. In the years since its inception, there have been a number of implementations of the model, with different characteristics emphasized.

In the Invasive Computing project, we implemented the actor model using the *FunState* model, which emphasizes clearly defined communication channels between the different actors [1]. The resulting library, *actorX10* [13], implements this model in a distributed memory environment (X10-based APGAS). In [13] and [14], we showed that this approach can be used for applications in HPC as well, by implementing a tsunami simulation proxy application, *SWE-X10*. SWE-X10 uses actors for communication across cores as well as shared memory domains and has been shown to work on the invasive hardware platform [15] as well as on HPC systems [14]. The PGAS features of X10 enable a straightforward implementation of the actor communication scheme, as the language enables an implicit transfer of arbitrary objects between ranks, and global pointers make it easy to track the relationships between the different actors.

On the other hand, using X10 comes with a number of drawbacks. Its high-level nature makes vectorization of code difficult unless one uses special annotations that enable manipulation of the intermediate C++ representation of the compiler. Legacy applications need to be rewritten in X10, which causes additional programming overhead. Furthermore, the use of third-party libraries such as netCDF [16] requires the creation of X10 wrapper classes to translate between X10 and C++ objects. Finally, a high performance communication backend is only available on clusters using the PAMI interconnect, otherwise an MPI translation layer is used [17].

ActorX10 is not the only implementation of the actor model: a popular implementation today is Akka [18], which has been incorporated into the Scala standard library in newer versions. However, it is not used for distributed HPC applications. The *C++ Actor Framework (CAF)* [19], [20] also supports distributed applications, but uses TCP sockets or OpenMPI as communication backend. CAF relies on a single mailbox and is able to handle incoming messages in arbitrary ordering. In contrast to that, we employ ports and channels as clearly defined communication points between actors. Therefore, communication between actors is more structured, and only specific actors can communicate with each other. Charm++ [21], [22] has a concept that is similar to actors. Objects (referred to as *chares*) send each other asynchronously executed remote procedure calls (signified by *entry methods*). Application programmers specify chares and entry methods using a custom notation (akin to CORBA interface files [23]). Similarly to the actor model, communication is structured through specific endpoints (entry methods), and chares need not be aware of the placement of communication partners. In essence, the model implemented in Charm++ resembles the CAF actor model more closely than ours.

As mentioned before, actorX10 shares the FunState actor formalism with the actor library that we propose in this paper, and exposes a similar interface for the application. However, we were able to identify bottlenecks and a number of opportunities for improvement, which enables us to scale an application to large-scale computations in the HPC context.

Therefore, we chose to implement an actor library that uses a modern communication infrastructure and an established programming language. The UPC++ library[2] fits these requirements well. It is implemented using modern C++ and utilizes C++ templates to provide a type-safe interface for asynchronous interaction within distributed memory environments. Like X10, it follows the APGAS model. However, it is implemented on a lower level so that communication operations and their overhead are made more explicit, allowing more finely grained control. All remote communication operations are performed asynchronously, and it is possible to send structured data as well as to execute code on remote ranks. Unlike with MPI, only one one-sided communication is supported, i.e., it is not necessary to have two ranks involved in the communication. Furthermore, the performance of UPC++ has been shown to match, or even surpass the one of competitive MPI implementations [24]. To implement the communication operations, the low-level communication library GASNet-EX uses network interconnection fabrics such as InfiniBand or Cray Aries directly [25]. Furthermore, using standard C++ enables interoperability with other libraries (e.g. netCDF I/O), debugging tools and performance analysis tools, and the integration of legacy components without wrappers. These features make UPC++ a perfect choice as the platform for the actor library.

III. The FunState Actor Model

In the following, we introduce the Actor Programming Model as the formal model for our library and the basic concepts used in our implementation. For our library, we follow a simplified version [13] of the *FunState* formalism [1].

[2]http://upcxx.lbl.gov/

An *Actor Graph* is a directed multigraph $G_a = (A, C)$. The vertices of the graph are the *actors*, and edges are the *channels* that connect the actors. Channels $c_{n,t}$ are queues with a finite capacity n of tokens of a specific type t. An actor $a = (\text{ID}, r, I, O, F, R) \in A$ is a tuple containing a unique name ID, a placement (e.g. on a rank) r, the set of *InPorts* I, the set of *OutPorts* O, the set of functions F executed by the actor, and the finite state machine (FSM) R. The In- and OutPorts of an actor are each connected to a single Channel, and provide the sole means of data exchange between actors, using tokens of type t. There is no direct access to shared data structures, or any other means of communication that would bypass the channels. The functionality executed by an actor is encapsulated in functions $f \in F$. We distinguish between guard functions F_{guard} and actions F_{action}. A guard function test if tokens and capacity of connected channels are sufficient to execute an action, or determine which action will be performed. This is done without modifying the data in the channel. Actions are allowed to modify the actor's state and tokens available in the channels. They will typically consume tokens available on the InPorts in order to compute results subsequently made available on the OutPorts.

The behavior of an actor is determined by its FSM $R = (Q, q_0, T)$, a tuple containing the set of states Q, the initial state $q_0 \in Q$, and the set of state transitions T. The latter are again a tuple $t = (q, k, f, q') \in T$ with originating state q, the activation pattern k, actions f and resulting state q'. k may consist of guard functions and checks to make sure that there are enough tokens available on the InPorts as well as sufficient space available on the OutPorts. The state machine R of an actor a is activated whenever there is a change, i.e. a new token added or a token removed, in the channels associated with the actors' ports, $I \cup O$.

We will illustrate this using a shallow water equations proxy application as an example in section VI.

IV. Underlying Technology Stack

The actor library and application described in this paper use UPC++ 1.0 [24], [26]–[28] as the underlying framework for parallelization. UPC++ follows the *Asynchronous Partitioned Global Address Space (APGAS)* programming model. It provides high-level, type-safe abstractions for one-sided data movement, a communication style that has been embraced by most modern interconnection networks. The key features for this work are global pointers, distributed objects and remote procedure calls.

A. Library Features

UPC++ runs under a Single Program Multiple Data (SPMD) control model. Each rank executes an instance of the same program. The memory of each process is partitioned into a private memory segment and a shared segment that is accessible globally by all ranks. It is possible to hold pointers (referred to as global pointers) to the shared segment of other ranks, but a dereferencing is not possible. Instead, one may transfer data using remote memory accesses to transfer data to and from the remote process, or one may operate on the remote data directly using *remote procedure calls (RPCs)*. All remote operations are performed asynchronously. After issuing the request, control is returned directly to the application, without the request necessarily being carried out immediately. Instead, the application programmer may attach notification actions (referred to as *completion handlers*) to the completion of a specific communication-related operation (or operations), such as the end of the source-side part of an operation, or the overall operation completion. Notification actions may be a range of different completions, for example, one may receive a *future*, which is an object that encapsulates a value that is not yet available, issue *local procedure calls (LPCs)* to notify other threads on the same rank of the completion, or issue remote procedure calls to execute code on remote ranks.

RPCs enable the application programmer to execute code on a specific remote rank. They receive as parameters a lambda function or a function pointer and the parameters passed to the lambda or the function, and may return the result of the execution of the lambda function on the remote rank. As of the current framework version, it is not yet possible to transfer arbitrary object graphs, but only objects with types that fulfill certain criteria. The types of the parameters and the return value need to conform to the *DefinitelyTriviallySerializable* [3] type trait, or be within a set of supported types. This set contains, among others, `std::vectors`, distributed objects and views. A feature that would allow for arbitrary object graphs to be serialized is currently under development and will be included in a future version of the library.

Another important concept is the *distributed object*. It is created collectively, and provides a common name for objects across all participating ranks. There are convenience methods to copy the data from other ranks, or one can use RPCs to access data directly on another rank.

The final important feature for our actor library is the way multiple threads are handled in UPC++. UPC++ introduces an abstraction that encapsulates all the UPC++-internal state relevant for a thread, called a *thread persona*. Each operating system thread is assigned a persona, and further personas may be created by the application programmer. The persona that is created and assigned upon initialization of the library at application start is referred to as the *master persona*. A thread may assume any persona, but one needs to be careful not to use the same persona on multiple threads at the same time. This concept is important, as completions are signaled only to the persona that issued them. Furthermore, incoming RPCs are only handled by the master persona of the receiving rank. For communication between different personas of a single UPC++ rank, local procedure calls may be used. They work similarly to the remote procedure calls, but are issued within a rank,

[3]They are a "C++ type that is *DefinitelyTriviallySerializable*, or a type for which there is a user-supplied implementation of the visitor function *serialize*" [27]. A *DefinitelyTriviallySerializable* type is a "C++ type that is either *TriviallyCopyable* and has no user-supplied implementation of the visitor function, or for which the trait `is_definitely_trivially_serializable` is specialized to have a member value that is true." [27]

and performed on a specific persona.

B. Library Implementation

The UPC++ runtime does not employ background threads in order to process communication requests or to perform internal communications [24]. Instead, the cost of communication operations is made explicit through the use of computational resources during library calls. To this end, the specification describes for every function of the library API if and what kind of background operations may be performed alongside the specified functionality of the operation. The library developers distinguish two kinds of work that may be performed, *internal progress* or *user-level progress*. Internal progress involves operations that will not have an immediate effect on the application, but instead comprise processing of asynchronous operations, both by handing them to the GASNet-EX runtime, as well as handling the runtime's responses. This type of progress will typically be generated by operations that require progress themselves, i.e. communication operations such as RPCs. User-level progress has to be requested by the programmer through `upcxx::progress()`, by waiting on futures or when barriers are issued. Finding a balance between computations and giving sufficient amount of computation time to the library is left to the application programmer. Not querying for progress in a timely manner may cause inefficient use of resources, as results that may already be available are not regarded.

UPC++ has two different threading modes available: the sequential and the parallel thread mode. Depending on the mode that is chosen, a different library is linked. The *sequential* threading mode assumes that there is only ever a single thread in a rank that calls UPC++ library functions. This assumption allows the backend to mostly forgo synchronization and leads to less overhead for UPC++ library calls. The *parallel* threading mode makes UPC++ thread-safe at the cost of reduced efficiency [28]. UPC++ is built on top of GasNET-EX, which provides a low overhead, low level network-independent communication layer that can take advantage of offload support on a variety of communication networks [24].

V. Implementation of the Actor Library

In the following, we will discuss the implementation of the Actor Library in C++ using UPC++.

A. Actor Graph

The first component is the actor graph, which we model in the `ActorGraph` class. Actor graphs are created collectively across the entire application. Therefore each rank has its own local control object for the graph. The instances on the different ranks are linked together internally using a distributed object (`upcxx::dist_object<ActorGraph *>`). Furthermore, the instance holds a dictionary of actor names to global pointers to the actors. The information in this dictionary is used mainly to connect the different actors during initialization. Later, the actors hold the global pointers directly. For the actor graph sizes (≈ 65000 actors) we encountered in our scaling tests, replicating the entire hash map on each rank was sufficient. If the library is to be used in exascale scenarios, the locally replicated dictionary may be replaced with an indirect distributed hash table similar to the one implemented in [24] or [29]. New actors may be added using the `addActor()` method. Whenever this happens, a global reference is created, and broadcast to the actor graph instances on the other ranks. The connection of ports is performed through the actor graph's `connectPorts()` method. Here, only global pointers to the involved actors as well as the port names in question are needed, and it is not necessary for either of the involved actors to be in the local part of the actor graph. Instead, the local graph will issue RPCs to the involved ranks as needed to connect the actors.

Finally, the computation is started using the actor graph by all participating ranks through calling the `run()` method. This will cause the start of the actor-based computation, which will only terminate if all actors signal their termination using their `stop()` method and all messages that may have been issued are handled. Upon termination, the execution time of the computation for the given rank is returned by the act method.

B. Actors

The `Actor` class serves as an abstract base class for user-defined actors. It comprises a unique name, and dictionaries for its In- and OutPorts. Additionally, it implements the necessary book-keeping to keep track of RPCs and LPCs that may still be in the internal queues of UPC++. The state machine is provided by the application programmer through implementing the `act()` method. For each change in one of the ports, the finite state machine needs to be given the chance to perform a state transition. Therefore, actor instances need to keep track of how often the act method still needs to be called (i.e. how often the actor has been triggered). Whenever one of the ports has a token added or removed in the connected actor, the trigger count is incremented, and whenever `act()` has been called, the trigger count is decremented.

C. Execution Strategies

Similar to MPI, UPC++ does not enforce a canonical way to parallelize an application. Therefore, we use the UPC++ library to communicate within and between ranks and implemented our own parallelization schemes. To realize parallel actor execution, we implemented three different execution strategies for the actor library, one relying on UPC++ ranks ("rank-based"), one based on C++ threads ("thread-based"), and one based on OpenMP tasks ("task-based"). In all cases, they implement the execution semantics of our actor model as specified in section III. This requires functionality that watches for changes in the channels and activates the corresponding actor when such a change happens, until the actor eventually signals its termination. For all execution strategies, this necessitates one or more *event loops* per rank. In the loop, the application needs to make tokens from other ranks available, and then execute the `act()` method on the local actors.

For the **rank-based execution strategy**, the event loop for each UPC++ rank is realized in the `ActorGraph` class. In the beginning of each iteration, there is a query for progress to the UPC++ runtime, causing waiting RPCs to be processed. Afterwards, any actor that has received any changes to its ports will get a chance to execute its `act()` method. This strategy is best suited for a small number of actors per UPC++ rank, as all actors within a rank are processed sequentially. Parallelism is achieved instead using multiple UPC++ ranks per physical node. In our case, we got the best results using one rank per logical core, and four to eight actors per rank. This method only uses one thread for the computation, and therefore avoids LPCs, and is able to use the faster, sequential UPC++ backend.

The **thread-based execution strategy** uses multiple C++ threads per UPC++ rank to parallelize actor execution. Each `Actor` instance is mapped onto its own thread, and the `ActorGraph` is mapped to the main thread of the rank. The individual actors and the actor graph each have their own event loop. The actor graph's loop continuously queries the UPC++ runtime for progress to process incoming RPCs, and therefore to exchange tokens and queue capacity updates with neighboring ranks. After an event is processed, an LPC is sent to the persona of the affected actor. In the actor's event loop, the first step is to query the runtime for progress to handle incoming LPCs, followed by a call to the act method if there was a change in one of the channels connected to the actor's ports. The actor graph queries for progress continuously in order to receive incoming RPCs from other ranks. For this method, one needs to be mindful of the amount of work done by the communication thread. If it is underutilized, there is a waste of resources, and if there is too much load on it, data exchange between actors is stalled. Furthermore, this model may cause additional overhead through spinning of the actors for progress. As there are multiple threads using functionality of the UPC++ runtime, its parallel backend is required.

Finally, the **task-based execution strategy** parallelizes the actor execution using OpenMP tasks. As with the rank-based execution strategy, there is only a single event loop per rank, in the `ActorGraph` instance. Similarly as with the other parallelization models, the runtime is queried for progress to process incoming RPCs. Then, we iterate through all local actors, and for those that have a positive trigger count, we schedule an OpenMP task that queries for progress and executes `act()`. The tasks are then executed asynchronously on worker threads (see Figure 1 for a possible execution trace) by the OpenMP runtime. We therefore needed to make sure that only a single worker thread is working on each actor at the same time. OpenMP enables the specification of dependencies between different tasks through memory addresses, in our case the address of an actor. This makes sure that tasks from different actors are executed concurrently, while preventing two tasks from the same actor to be executed at the same time. The advantage of this approach compared to the thread-based execution strategy is the possibility to match the number of threads employed by a node to its computational resources. Instead of a one-to-one mapping of actors to operating system

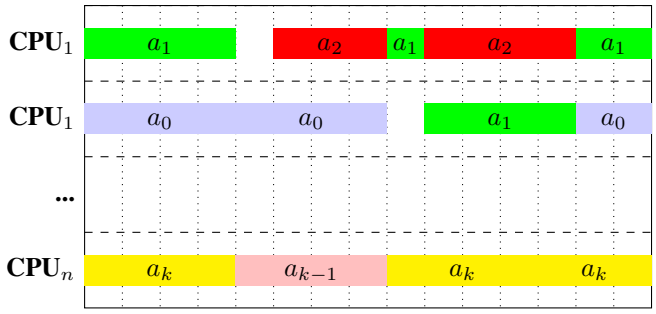


Fig. 1. OpenMP tasking for Actor FSM Execution. k actors are scheduled onto n CPU cores

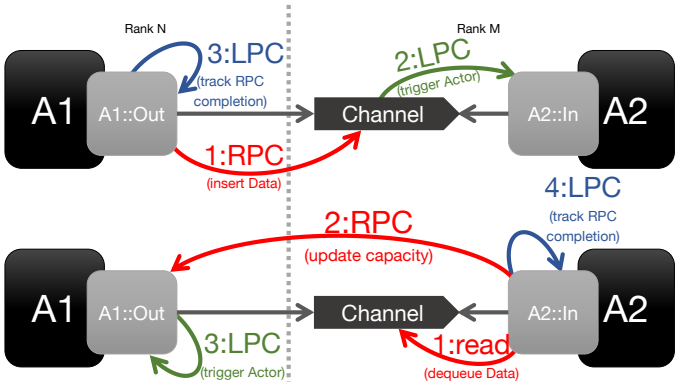


Fig. 2. Communication using Ports. The upper half depicts a `write()` issued by actor "A1" on the port "Out", and the lower half depicts a `read()` issued by actor "A2" on the port "In". RPCs are issued if the communication occurs across rank-boundaries. The LPCs depicted in blue are issued when the task-based execution strategy is used, while the green ones are needed when the thread-based execution-model is used.

threads, work is distributed onto a number of threads specified at runtime. When an actor is not triggered, it will not create tasks to be scheduled, and it therefore does not use up any CPU resources. Compared to the rank-based execution strategy, we are able to distribute a larger amount of actors onto a single rank without decreasing the performance significantly. This reduces the number of RPCs that need to be issued for communication across rank boundaries. As with the thread-based execution strategy, the use of multiple worker threads necessitates the parallel UPC++ backend.

D. Ports and Communication

Ports are implemented through the class templates `InPort<T,c>` and `OutPort<T,c>` with a fixed token type `T` and capacity `c`. They are used as the communication interface to other actors and handle the insertion (`write()`) and extraction (`read()`) of data from the channels. Channels are also implemented as class templates `Channel<T,c>` with the same template parameters. Internally, they use a ring buffer to store data until it is read. We chose to have the channels always on the same rank as the InPort so that the receiving actor does not need to buffer messages. The communication scheme is depicted in Figure 2. When an actor ("A1") *writes* to its port ("Out"), the port will insert the token into the channel.

If the channel is on another rank, the port will issue an RPC to that rank, otherwise, the channel is accessed directly. Next, the receiving actor ("A2") needs to be notified of the change. To that end, depending on the choice of execution strategy, LPCs may need to be issued. When using the thread-based execution strategy, an LPC is sent to the persona of "A2", to trigger a state machine execution. Finally, it is necessary to keep track of the number of currently active RPCs. For the rank-based and the thread-based execution strategies, that book-keeping can be performed locally using a lambda-continuation, and for the task-based execution strategy, an LPC-continuation is attached to the actor.

A *read* operation starts with the InPort removing the data from the channel. Afterwards, the corresponding port on the other side of the channel needs to be notified. The notification comprises an update to the counter local to the OutPort, followed by triggering of the sending actor. As with writing to the channel, if necessary, the notification will be issued using an RPC. The notification and book-keeping are performed akin to the ones of the read operation, with LPCs being issued as necessary.

VI. Pond, an Actor-Based Tsunami Application

Tsunamis are typically simulated using the shallow water equations, which may be derived from the more general Navier-Stokes equations by averaging over the unknown quantities along the water column [30]. The equations are given as

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = S(t,x,y)$$

Here, x and y refer to the two spatial dimensions, t refers to time, h is the height of the water column, hu and hv are the momenta in the two spatial dimensions, g is the gravitational constant, and $S(x,y,t)$ is used to model source terms. Depending on the source terms used, one may model, e.g., tsunamis, or flooding.

As a sample application to demonstrate the use of our UPC++ actor library, we implemented *Pond*, a shallow water proxy application. The application is based on two prior applications, SWE and SWE-X10. In the following, we will first discuss the two prior applications and highlight their influence on Pond, beginning with SWE. Then we will introduce Pond, with a focus on the actor graph and implementation choices that are different here.

SWE[4] [31] implements a Finite Volume discretization on a Cartesian grid with explicit Euler time stepping, following the approach of LeVeque et al [30]. This yields the following update scheme:

$$\begin{aligned} Q_{i,j}^{(\mathrm{n}+1)} = Q_{i,j}^{(\mathrm{n})} &- \frac{\Delta t}{\Delta x}\left(\mathcal{A}^{+}\Delta Q_{i-\frac{1}{2},j}^{(\mathrm{n})} + \mathcal{A}^{-}\Delta Q_{i+\frac{1}{2},j}^{(\mathrm{n})}\right) \\ &- \frac{\Delta t}{\Delta y}\left(\mathcal{B}^{+}\Delta Q_{i,j-\frac{1}{2}}^{(\mathrm{n})} + \mathcal{B}^{-}\Delta Q_{i,j+\frac{1}{2}}^{(\mathrm{n})}\right), \end{aligned} \quad (1)$$

[4]https://www5.in.tum.de/wiki/index.php/SWE

For each grid cell i,j, the unknown quantities $Q_{i,j}^{(n)} = [h_{i,j}^{(n)}, (hu)_{i,j}^{(n)}, (hv)_{i,j}^{(n)}, b_{i,j}^{(n)}]$ are updated iteratively based on an initial state t_0 with a timestep Δt. $\mathcal{A}^{\pm}\Delta Q_{i\pm\frac{1}{1},j}^{(n)}$ and $\mathcal{B}^{\pm}\Delta Q_{i,j\pm\frac{1}{1}}^{(n)}$ denote the solutions of the Riemann problems at the left and right, and top and bottom boundaries, respectively.

The SWE framework offers multiple approximate Riemann solvers for the aforementioned Riemann problems, with different characteristics in terms of precision, applicability (e.g. capability of handling inundation of coastal areas) and computational effort. For our solution, we chose to utilize the HLLE Riemann solver [32], which is able to model inundation of dry areas of the domain.

The implementation uses the abstraction of the `SWE_Block` class for a Cartesian grid patch of the simulation domain. The class serves as an encapsulation of the computational parts of the simulation. In SWE, the computation of a new time step is organized as follows: *1. Set boundary conditions.* In the beginning of the iterations, the cells at the boundary of the block's domain need to be set according to specified boundary conditions, according to the scenario for the edges of the simulation domain or based on data from other blocks for edges adjacent to other blocks (ghost layers). *2. Compute Fluxes.* Next, for all cell boundaries, the fluxes are computed by iterating over the block-local domain. In SWE, this process may be parallelized using OpenMP-parallel for loops. The individual flux computations are data-parallel and may be vectorized [33] using OpenMP pragmas[5] with a suitable compiler. *3. Updating cell values.* Thereafter, the cell quantities are updated according to the scheme above, based on the timestep size Δt. Finally, if desired, output data may be produced using the netCDF I/O library.

SWE follows the BSP programming model for parallelization, with MPI as the inter-rank communication interface. Each node is assigned an MPI rank, and a single block, relying on OpenMP for intra-node parallelism. In the beginning of each timestep, ghost layers are exchanged using `MPI_SendRecv` calls. After the flux computation, the largest globally safe time step is determined using an `MPI_Allreduce` operation.

SWE-X10 is an actor-based tsunami simulation proxy application, written in X10, using the actorX10 library [13], [14]. It shares the numerical approach (see Equation 1) of SWE, and also implements an `SWEBlock` class that offers the same operations also offered by its counterpart in SWE. Its system design is very similar to Pond, and it may be seen as its direct precursor. In SWE-X10, we partition the simulation domain into quadratic patches (each managed by a `SWEBlock` instance) and assign each patch to an actor. Instead of one large patch per rank as in SWE that is parallelized using OpenMP, here there will be multiple sequentially executed actors that run concurrently. The patches need to communicate the cell values on their boundaries to adjacent patches, therefore we connect actors with adjacent patches using channels of

[5]Using `#pragma omp simd` with a reduction on the loops and by requiring the compiler to generate vectorized versions of the Riemann solver functions using `#pragma omp declare simd`.

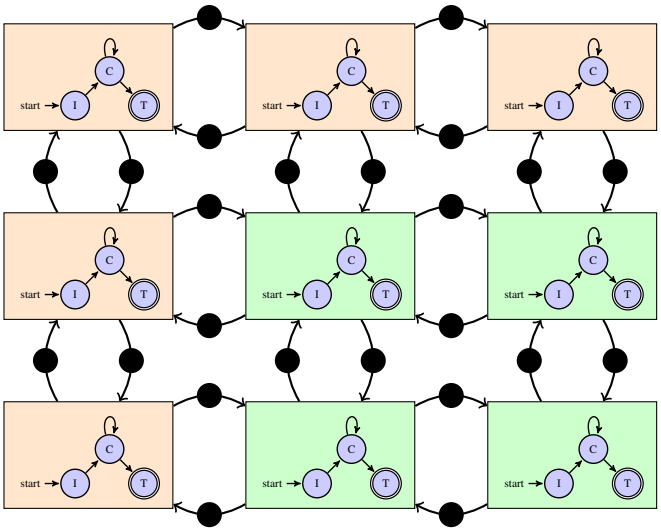


Fig. 3. Sample actor graph with a 3×3 grid. Actors depicted in orange are on one node, and actors depicted in green on another.

`CommunicationData` objects that hold the row/column of the cell data that is to be transferred. The basic setup is depicted in Figure 3. In contrast to SWE, there is no global dependency here, the coordination is completely local, as each actor only depends on its direct neighbors. This removes the global waiting condition that leads to all ranks always waiting for the slowest one, and enables actors to proceed with computing time steps without always having to wait for slower counterparts in parts of the simulation domain not directly adjacent to theirs. However, the timestep is fixed at the start of the simulation. This setup does not yet exploit the advantages of the actor model.

One way to profit from the actor model is the use of *lazy activation*, as shown in [14]. Here, an actor only starts computing and sending updates to its neighbors when the tsunami actually reaches the edges of the actor's domain. Actors that are in parts of the simulation domain that are at rest initially do not perform any computations at all. In a test scenario, we were able to significantly reduce the number of CPU hours spent on the computations of the solution of a scenario. Another feature that is easier to implement using the actor model is *patch-adaptive local time stepping (LTS)*. With LTS, the actors set their individual timestep as a defined fraction of the original timestep (i.e. $\ldots, \frac{1}{4}, \frac{1}{2}, 1, 2, \ldots$) for the patch they manage. The individual actors send their updates out as they are computed, and actors receiving them may either use the data if the time step fits, interpolate in case the neighbor has a bigger timestep, or throw out unnecessary updates if the neighbor has a smaller timestep. This can be especially interesting in parts of the domain where inundation occurs. These areas can have a significantly smaller time step than areas that only contain water, or areas that the wave already passed through. In comparison to the global time stepping scheme of SWE, one only needs to compute the small timesteps where they are actually required, instead of in the entire simulation domain. We currently have a prototypic implementation of this in SWE-X10.

Pond may be viewed as an amalgamation of SWE and SWE-X10. As both SWE and Pond are written in C++, we reuse the `SWE_Block` class with OpenMP disabled, but otherwise unchanged. We also reuse other parts of SWE, such as the File I/O, and the HLLE solver. On top of that, we added the actor-based coordination as implemented in SWE-X10. As the focus on this paper is more on the actor library and its different execution strategies, we stuck to a basic implementation, as originally shown in [13]. Implementation of Lazy Activation and LTS should be possible without any modifications of the actor library.

The actor graph for Pond may therefore be written as

$$
\begin{aligned}
G_a^{\text{Pond}} &= (A_{\text{Pond}}, C_{\text{Pond}}) \qquad (2)\\
A_{\text{Pond}} &= \{a_{i,j} |\ 0 \leqslant i < n_x \wedge 0 \leqslant j < n_y\}\\
C_{\text{Pond}} &= \big\{c_{a_{i,j},a_{i',j'}} |\ \ (i = i' \wedge |j - j'| = 1)\\
&\qquad\qquad \vee (|i - i'| = 1 \wedge j = j')\big\},
\end{aligned}
$$

similarly to the SWE-X10's actor graph [13]. Figure 3 depicts an example actor graph for a simulation domain distributed onto 3×3 patches.

As we execute Pond on distributed memory systems, the actor graph needs to be distributed onto multiple ranks. For unstructured grid codes, a known technique is the use of graph partitioners such as *Metis* [34]. Metis uses a multi-level graph partitioning approach to provide a balanced graph partitioning with minimal edge cut. This yields a partitioning of the actor graph that balances load balancing against communication overhead. We configured the graph partitioning library to produce contiguous partitions with a maximal load imbalance of $\pm 10\%$.

The actor-based modeling lead to a layered software architecture in Pond. At the top layer, the actor graph implicitly coordinates the computation of updates based on the available data from the neighbors. The simulation actors control the calculations on the patch-level based on the available data. This behavior is encoded in the simulation actors' FSMs. The FSM's actions call functionality from the bottom layer `SWE_Block` class that is responsible for the implementation of the iterations over the patch domain. In terms of the FunState model, we obtain

$$
\begin{aligned}
a_{i,j} &= \big(\text{ID}, r, I_{a_{i,j}}, O_{a_{i,j}}, F_a, R_a\big) \qquad (3)\\
I_{a_{i,j}} &= \Big\{ip_{i',j'} | c_{a_{i',j'},a_{i,j}} \in C_{\text{Pond}}\Big\}\\
O_{a_{i,j}} &= \Big\{op_{i',j'} | c_{a_{i,j},a_{i',j'}} \in C_{\text{Pond}}\Big\}.
\end{aligned}
$$

The finite state machine R_a is depicted in Figure 4. The set of functions consists of the guards: *mayRead()* returns true if there is at least one data item available in each channel, and *mayWrite()* returns true if there is sufficient space available in all channels to write at least one data item to each channel. The actions sendData() and receiveData() write data from the copy layer to the channels or read from the channels and write to the

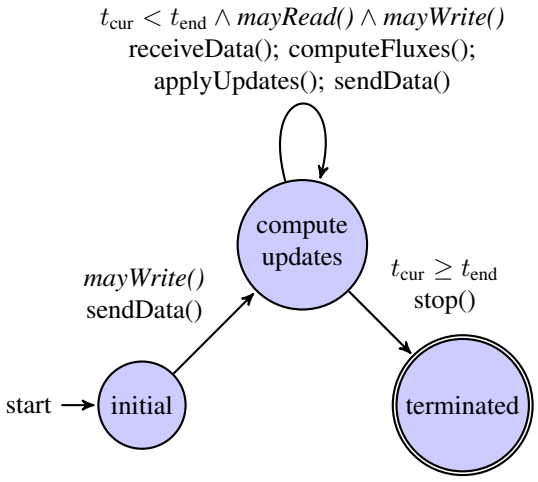


Fig. 4. Finite state machine for the simulation actor. Italicized functions are guard functions, the rest are actions.

ghost layers, respectively. Once data is read, computeFluxes() and applyUpdates() are called to perform the computation of fluxes and the updating of the cells' unknown values.

VII. Results

We tested the implementation on the Knights Landing partition of the National Energy Research Scientific Computing Center's Cori Cluster [35]. The partition employs 9688 nodes with a single-socket Intel Xeon Phi and a combined theoretical peak performance of 29.5 PFlop. Each Xeon Phi is equipped with 68 cores clocked at 1.4GHz, yielding a theoretical peak performance of 6 TFlop (SP) per node. The peak memory bandwidth is 102 GB/s for the off-chip DDR memory, and around 460 GB/s for the on-chip MCDRAM. For our tests, we used the default configuration of the KNL nodes, and the Intel Programming Environment. Optimizations were enabled (`O3`), and the iteration over the patch was therefore automatically vectorized using AVX-512.

We compare Pond using the three different execution strategies (as described in subsection V-C) to the two applications described in the previous section, the X10 application SWE-X10, and the BSP-based SWE. As described above, all applications use the same numerical scheme, and the same vectorized HLLE solver. In all tests, we used a radial dam break scenario, which is easily scaled to any size. We measure the time from the start of the actor graph execution until all actors are terminated for Pond and SWE-X10, and the time from the beginning of the first iteration to the end of the last one for SWE. File I/O was disabled in all cases. Finally, we are able to compute the number of Flop/s based on the number of iterations in SWE, and the number of patch updates in SWE-X10 and Pond. The configurations are also described in Table I. We chose them based on the best results obtained in empirical pre-tests.

The weak scaling test was performed with a per-node workload of 4096×4096 grid cells per node. We performed tests starting on a single node up to 128 nodes. The base workload per core was set at 256×256 grid cells per logical thread. Results are shown in Figure 5a.

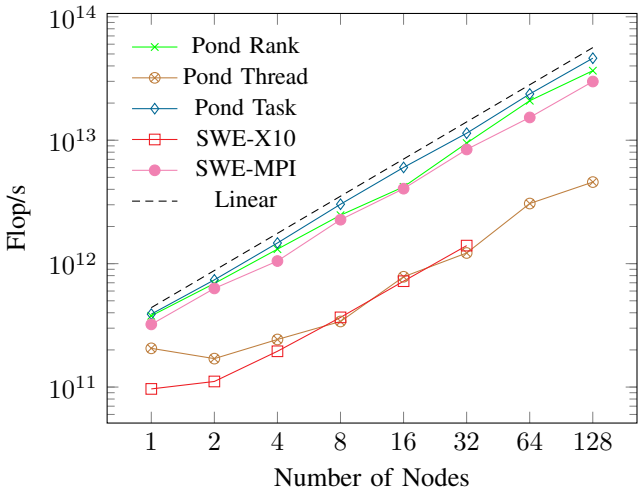
SWE-MPI scaled about linearly with the number of nodes in the computation. SWE-X10 exhibited the lowest performance, with roughly an order of magnitude lower performance compared to SWE-MPI or the fast configurations of Pond. At this lower level, scaling is linear from two nodes up to 32 nodes. For larger configurations, the job failed to complete, with all of the allocated computation time spent on the distribution of actors. The architecture of the actorX10 library unfortunately requires this to be performed sequentially. While overhead is small for configurations with a small number of nodes, the high number of actors for runs with more than 32 nodes, the number of actors (4096), and channels(≈ 16000) that need to be migrated proved infeasible. For Pond, there are significant differences between the different actor execution strategies. *Pond Thread* performs worst, at a similar level as SWE-X10. *Pond Rank* performs on a level competitive with SWE-MPI, managing to yield a roughly 20% performance benefit over SWE-MPI for the largest run with 128 nodes. *Pond Task* outperforms the other implementations in this test, and is on average 38% faster than SWE-MPI, with over 50% higher performance for the run with 128 nodes.

In the weak scaling test, *Pond Rank*, *Pond Task* and SWE-MPI exhibited comparable performance and scaled linearly with the number of nodes. Therefore, we performed strong scaling tests to explore the scalability limits of the respective applications. In the first test, we set the size of the simulation domain to 16384×16384 grid cells. For a single-node configuration, this led to a patch size of 512×512 grid cells, down to a patch size of 64×64 grid cells with 128 nodes. In the second test, we set the size of the simulation domain to 8192×8192 cells, yielding configurations with 256×256 cells per patch for single-node runs down to 32×32 for runs with 128 nodes. For Pond and SWE-X10, the patch size for an individual actor is determined so that there is at least one actor per logical core of the node. If it is not possible to evenly divide the patch size in this case, the patch size is halved, and the smaller patches are distributed. In the case of SWE-MPI, the OpenMP parallel for-loop handles the distribution of the node-local data to the cores. Nevertheless, the per-node working set size remains the same between all configurations.

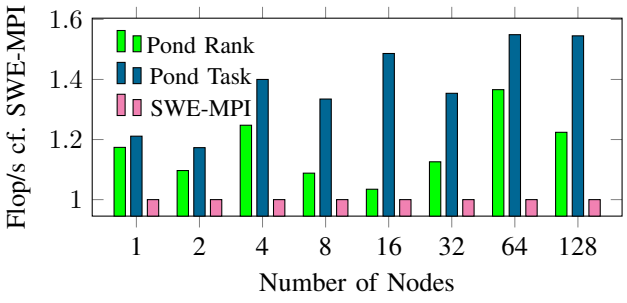
The results of both tests, shown in Figure 6a and Figure 6c suggest similar conclusions. The performance of SWE-MPI degenerates gradually as the size each core's working set shrinks. For the actor-based solutions, the more constrained patch size leads to a more sudden drops in performance, as smaller patch sizes–necessary to evenly distribute the grid–lead to more actors and therefore more coordination overhead. SWE-X10 only yields acceptable performance using patch sizes of at least 512×512, otherwise performance is dominated by coordination overhead. The application performs significantly better with large patch sizes. For patch sizes smaller than 256×256, there are no more performance benefits to increasing the number of ranks in the computation, and the additional

TABLE I
CONFIGURATIONS USED FOR THE SCALING TESTS

Execution Type	Symbol	Description
Pond Rank	×	Pond using the rank-based execution strategy. One Rank per logical core, and four to eight actors per rank
Pond Thread	⊗	Pond using the thread-based execution strategy. Two ranks per node, and one to two actors per logical core
Pond Task	◇	Pond using the task-based execution strategy. Sixteen ranks per node, and roughly four to eight actors per logical core
SWE-X10	□	SWE-X10 using actorX10. Two Places per node, and one to two actors per logical core
SWE-MPI	●	SWE using MPI and OpenMP. One rank per node, and 272 OpenMP threads per rank
Linear	- - -	Ideal scaling based on fastest single node configuration of Pond Task, one rank, no RPCs.

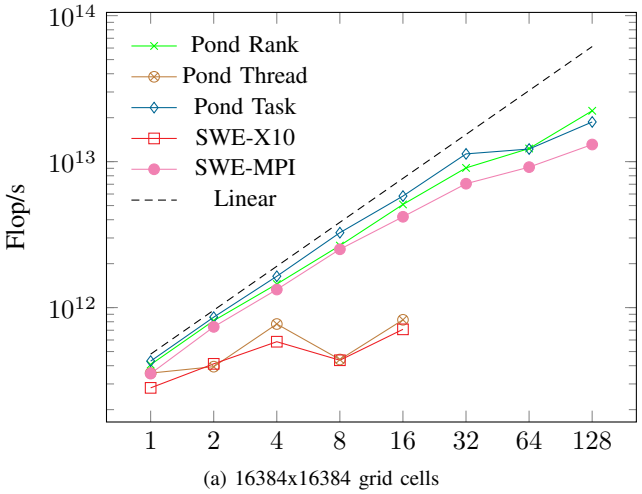


(a) Weak Scaling

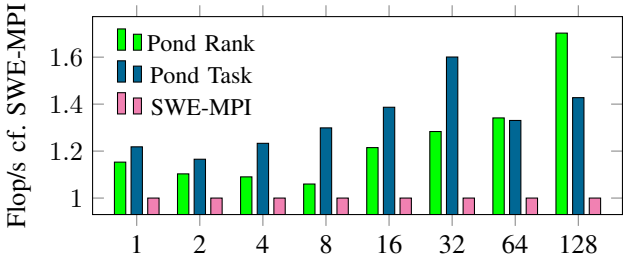


(b) Detailed comparison

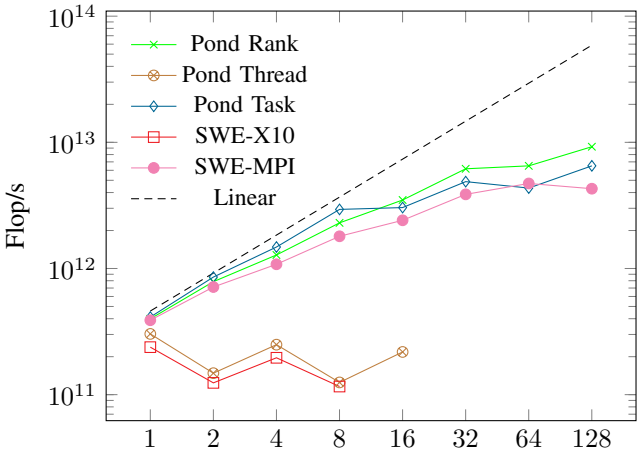
Fig. 5. Weak Scaling test (4096 × 4096 cells per node). The largest two configurations of SWE-X10 did not run to completion.



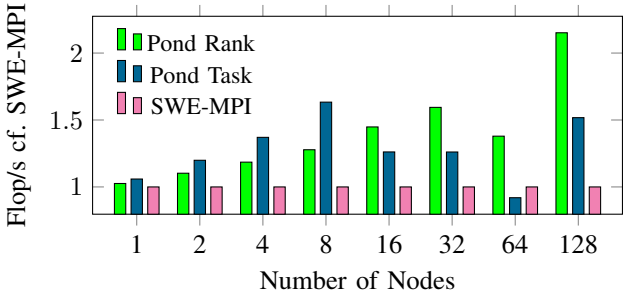
(a) 16384x16384 grid cells



(b) Detailed comparison: Strong Scaling 16384 × 16384



(c) 8192x8192 grid cells



(d) Detailed comparison: Strong Scaling 8192 × 8192

Fig. 6. Strong Scaling test. See Table I for the configurations. In Figure 6b and Figure 6d, the relative performance compared to the run of SWE-MPI with the corresponding number of nodes is displayed

compute capacity is used up completely on the additional overhead. A similar behavior may be observed with *Pond Thread*. This may be explained by their similar execution strategy. In both cases, each actor is executed on its own thread. There is overhead for the communication between the threads of a rank, and for *Pond Thread*, the communication is handled by a single communication thread. As there are typically more actors than cores, the operating system may not be able to schedule the threads at the same time. In SWE-X10, actors without work are sent to sleep, which will cause the overhead of an OS context change. In Pond Thread, an actor without work will continue polling for progress, again causing overhead, as the compute resource might have been used for the computation of updates by another actor. Both approaches have the same effect: they are only feasible if the workload per thread is sufficiently large, a property that prevents more fine-grained parallelization. This is handled differently with the other execution strategies. With *Pond Rank*, each rank only has one thread, leading to one UPC++ rank per logical core. Each rank is responsible for its own progress, and, as there is only one thread per UPC++ rank, the sequential UPC++ backend may be used. In contrast to *Pond Thread*, there is no need for synchronization and locking, which leads to better performance. In case of *Pond Task*, work is distributed onto the worker threads through the OpenMP runtime, and actors only get scheduled if they are actually triggered. The OpenMP master thread is responsible for communication and keeps polling the UPC++ runtime for updates. This may be a bottleneck if there are too many communication requests to be performed, leading to the worker threads being idle while the master thread is busy evaluating incoming RPCs.

We evaluated this effect in a separate strong scaling test, again with 16384×16384 grid cells by comparing the performance of different amounts of UPC++ ranks per node (see Figure 7). We found that for the Xeon Phi, configurations using eight or sixteen ranks per node outperformed configurations with one, two or four ranks per node, especially in situations with smaller workloads per actor. For runs with a smaller number of nodes, and therefore a larger computational cost for each task, a low number of ranks per node is a better choice, as more cores may be utilized for computation, whereas for runs with more nodes, the individual tasks carry less load, while the number of communication requests remains the same. This value is difficult to fine-tune automatically by the library, as the incoming communication requests are always handled by the single thread holding the master persona. We anticipate that this value needs to be fine-tuned to respective target architectures and the anticipated workload.

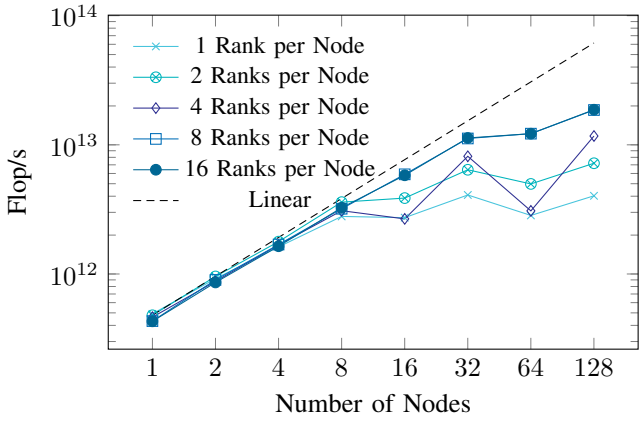


Fig. 7. Strong scaling test evaluating different configurations of the task-based execution strategy of Pond. We compare different numbers of Ranks per physical node.

VIII. Outlook and Conclusion

In this work, we introduced an actor library built on top of the UPC++ PGAS library. The library is based on the FunState model and uses explicit communication channels and finite state machines in order to expose the control and data flow of an application explicitly. We implemented three execution strategies: one using threads for each actor, one using UPC++ ranks for parallelization, and one relying on OpenMP tasks. To demonstrate the viability of using actor-based programming in HPC, we implemented Pond, a tsunami simulation proxy application using the UPC++ actor library. UPC++ facilitates the use of one-sided communication operations, enabling us to overlap communication and communication using operations that mirror the capabilities of the underlying hardware. We compared it to SWE-X10, a prior actor-based application written in X10, and SWE, a tsunami application using MPI and OpenMP on the Intel Xeon Phi partition of Cori. Pond outperforms SWE-X10 by an order of magnitude in a weak scaling test. Due to the lower overhead of our library, we are able to use significantly smaller patch sizes without penalizing performance. Pond also outperforms SWE, a straightforward BSP-type implementation, with the added benefit that the application programmer does not need to implement OpenMP and MPI parallelization, but only needs to specify the actors and the communication with their neighborhood. For our actor library, we found that the OpenMP-based execution strategy which maps actor state machine executions to OpenMP tasks performs best, except at the scalability limit of the strong scaling test, where the rank-based execution strategy outperforms it. In further work, we want to implement the actor-based approach using other emerging runtime systems, such as HPX [36]. Furthermore, we would like to extend Pond to enable block-adaptive local time stepping. Finally, we would like to explore a use of the actor model for other problem domains. Both our actor library and Pond are available under the GPL at https://bitbucket.org/apoeppl/actor-upcxx.

Acknowledgments

This research was funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) - Project number 14671743 - TRR 89 Invasive Computing. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the National

Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Scott Baden was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich, "FunState – an internal design representation for codesign," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 4, pp. 524–544, Aug. 2001.

[2] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, ser. PASCO '07. New York, NY, USA: ACM, 2007, pp. 24–32. [Online]. Available: http://doi.acm.org.eaccess.ub.tum.de/10.1145/1278177.1278183

[3] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1422–1434, Apr 2018. [Online]. Available: https://doi.org/10.1007/s11227-018-2238-4

[4] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, March 2009.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1631

[6] A. Hendricks, T. Heller, H. Jordan, P. Thoman, T. Fahringer, and D. Fey, "The AllScale runtime interface: Theoretical foundation and concept," in *Proceedings of the 9th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers*, ser. MTAGS '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 13–19. [Online]. Available: https://doi-org.eaccess.ub.tum.de/10.1109/MTAGS.2016.4

[7] Q. Meng, A. Humphrey, and M. Berzins, "The Uintah framework: A unified heterogeneous task scheduling and runtime system," in *Digital Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 2441–2448, sC12 2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC 2012. [Online]. Available: http://www.sci.utah.edu/publications/Men2012b/uintah-wolfhpc12.pdf

[8] S. Wildermann, M. Bader, L. Bauer, M. Damschen, D. Gabriel, M. Gerndt, M. Glaäs, J. Henkel, J. Paul, A. Pöppl, S. Roloff, T. Schwarzer, G. Snelting, W. Stechele, J. Teich, A. Weichselgartner, and A. Zwinkau, "Invasive computing for timing-predictable stream processing on MPSoCs," *it - Information Technology*, vol. 58, no. 6, pp. 267–280, Sep. 2016.

[9] H.-J. Bungartz, C. Riesinger, M. Schreiber, G. Snelting, and A. Zwinkau, "Invasive computing in HPC with X10," in *Proceedings of the Third ACM SIGPLAN X10 Workshop*, ser. X10 '13. New York, NY, USA: ACM, 2013, pp. 12–19. [Online]. Available: http://doi.acm.org/10.1145/2481268.2481274

[10] A. Mo-Hellenbrand, I. Comprés, O. Meister, H.-J. Bungartz, M. Gerndt, and M. Bader, "A large-scale malleable tsunami simulation realized on an elastic MPI infrastructure," in *Proceedings of the Computing Frontiers Conference*, ser. CF'17. New York, NY, USA: ACM, 2017, pp. 271–274. [Online]. Available: http://doi.acm.org.eaccess.ub.tum.de/10.1145/3075564.3075585

[11] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[12] G. A. Agha, "ACTORS: A model of concurrent computation in distributed systems," MIT Artificial Intelligence Laboratory, Tech. Rep. AITR-844, Jun. 1985.

[13] S. Roloff, A. Pöppl, T. Schwarzer, S. Wildermann, M. Bader, M. Glaß, F. Hannig, and J. Teich, "ActorX10: An actor library for X10," in *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, ser. X10 2016. New York, NY, USA: ACM, 2016, pp. 24–29. [Online]. Available: http://doi.acm.org/10.1145/2931028.2931033

[14] A. Pöppl, M. Bader, T. Schwarzer, and M. GlaSS, "SWE-X10: Simulating shallow water waves with lazy activation of patches using ActorX10," in *2016 Second International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, Nov 2016, pp. 32–39.

[15] A. Pöppl, M. Damschen, F. Schmaus, A. Fried, M. Mohr, M. Blankertz, L. Bauer, J. Henkel, W. Schröder-Preikschat, and M. Bader, "Shallow water waves on a deep technology stack: Accelerating a finite volume tsunami model using reconfigurable hardware in invasive computing," in *Euro-Par 2017: Parallel Processing Workshops*, D. B. Heras, L. Bougé, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, J. G. Barbosa, L. Ricci, S. L. Scott, S. Lankes, and J. Weidendorfer, Eds. Cham: Springer International Publishing, 2018, pp. 676–687.

[16] R. Rew and G. Davis, "NetCDF: an interface for scientific data access," *IEEE Computer Graphics and Applications*, vol. 10, no. 4, pp. 76–82, July 1990.

[17] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri, "X10 and APGAS at petascale," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 53–66. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555245

[18] P. Nordwall, J. Andrén, J. Rudolph, A. Engelen, C. Batay, and H. Edelson, "The Akka actor library documentation," 2011. [Online]. Available: https://doc.akka.io/docs/akka/current/

[19] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, "Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments," in *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2013, pp. 87–96.

[20] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting Actor Programming in C++," *Computer Languages, Systems & Structures*, vol. 45, pp. 105–131, April 2016. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2016.01.002

[21] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy, "The CHARM Parallel Programming Language and System: Part I – Description of Language Features," *Parallel Programming Laboratory Technical Report #95-02*, 1994.

[22] L. V. Kalé and G. Zheng, "The Charm++ programming model," in *Parallel Science and Engineering Applications: The Charm++ Approach*. CRC Press, 2016, pp. 1–16.

[23] O. Portal, "Object management group: common object request broker," *OMG. org. Retrieved from omg. org August*, vol. 22, p. 2009, 2009.

[24] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed, "UPC++: A High-Performance Communication Framework for Asynchronous Computation," in *Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS. IEEE, 2019. [Online]. Available: https://escholarship.org/uc/item/1gd059hj

[25] D. Bonachea and P. H. Hargrove, "GASNet-EX: A High-Performance, Portable Communication Library for Exascale," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-2001174, October 2018, to appear: Languages and Compilers for Parallel Computing (LCPC'18). [Online]. Available: https://escholarship.org/uc/item/0xg7b704

[26] J. Bachan, D. Bonachea, P. H. Hargrove, S. Hofmeyr, M. Jacquelin, A. Kamil, B. van Straalen, and S. B. Baden, "The UPC++ PGAS library for exascale computing," in *Proceedings of the Second Annual PGAS Applications Workshop*, ser. PAW17. New York, NY, USA: ACM, 2017, pp. 7:1–7:4. [Online]. Available: http://doi.acm.org/10.1145/3144779.3169108

[27] "UPC++ Specification, v1.0 Draft 10," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-2001192, March 2019. [Online]. Available: https://escholarship.org/uc/item/25m555p9

[28] J. Bachan, S. B. Baden, D. Bonachea, P. H. Hargrove, S. Hofmeyr, M. Jacquelin, A. Kamil, and B. van Straalen, "UPC++ Programmer's Guide, v1.0-2019.3.0," Lawrence Berkeley National Laboratory,

Tech. Rep. LBNL-2001191, March 2019. [Online]. Available: https://escholarship.org/uc/item/9vf0h34w

[29] L. Monnerat and C. L. Amorim, "An effective single-hop distributed hash table with high lookup performance and low traffic overhead," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 7, pp. 1767–1788, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3342

[30] R. J. LeVeque, D. L. George, and M. J. Berger, "Tsunami modelling with adaptively refined finite volume methods," *Acta Numerica*, vol. 20, pp. 211–289, 2011.

[31] A. Breuer and M. Bader, "Teaching parallel programming models on a shallow-water code," in *Proceedings of the 2012 11th International Symposium on Parallel and Distributed Computing*, ser. ISPDC '12. IEEE Computer Society, 2012, pp. 301–308.

[32] A. Harten, P. D. Lax, and B. van Leer, "On upstream differencing and Godunov-type schemes for hyperbolic conservation laws," in *Upwind and High-Resolution Schemes*, M. Y. Hussaini, B. van Leer, and J. Van Rosendale, Eds. Springer, 1997, pp. 53–79. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-60543-7_4

[33] M. Bader, A. Breuer, W. Hölzl, and S. Rettenberger, "Vectorization of an augmented Riemann solver for the shallow water equations," in *Proceedings of the 2014 International Conference on High Performance Computing and Simulation (HPCS 2014)*. IEEE, 2014, pp. 193–201.

[34] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: https://doi.org/10.1137/S1064827595287997

[35] Y. He, B. Cook, J. Deslippe, B. Friesen, R. Gerber, R. Hartman-Baker, A. Koniges, T. Kurth, S. Leak, W.-S. Yang, Z. Zhao, E. Baron, and P. Hauschildt, "Preparing NERSC users for Cori, a Cray XC40 system with Intel many integrated cores," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 1, p. e4291, 2018, e4291 CPE-17-0254. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4291

[36] H. Kaiser, B. A. L. aka wash, T. Heller, A. Bergé, M. Simberg, J. Biddiscombe, A. Bikineev, G. Mercer, A. Schäfer, A. Serio, and et al., "Stellar-group/hpx: Hpx v1.3.0: The c++ standards library for parallelism and concurrency," May 2019.

APPENDIX: ARTIFACT DESCRIPTION

Summary of the experiments reported

We ran strong scaling and weak scaling tests with node configurations of 1, 2, 4, 8, 16, 32, 64 and 128 nodes with the applications SWE, SWE-X10 and Pond. For Pond, we tested the three different execution strategies (rank-based execution strategy, thread-based execution strategy and task-based execution strategy). We built the applications using the following commands:

Application	Compile Command
SWE	`scons buildVariablesFile = build/options/SWE_cray_mpi_vectorized.py`
SWE-X10	`make clean; make NATIVE=1 NERSC=1 LOG=info RT=mpi`
Pond Rank	`CXX="env UPCXX_THREADMODE=seq upcxx -O3" cmake -DBUILD_RELEASE=ON -DBUILD_USING_UPCXX_WRAPPER=ON -DENABLE_FILE_OUTPUT=OFF -DENABLE_MEMORY_SANITATION=OFF -DENABLE_LOGGING=OFF -DENABLE_O3_UPCXX_BACKEND=ON -DENABLE_PARALLEL_UPCXX_BACKEND=OFF -DACTORLIB_USE_OPENMP_TASKS=OFF -DIS_CROSS_COMPILING=ON -build pond <PATH>; make`
Pond Thread	`CXX="env UPCXX_THREADMODE=par upcxx -O3" cmake -DBUILD_RELEASE=ON -DBUILD_USING_UPCXX_WRAPPER=ON -DENABLE_FILE_OUTPUT=OFF -DENABLE_MEMORY_SANITATION=OFF -DENABLE_LOGGING=OFF -DENABLE_O3_UPCXX_BACKEND=ON -DENABLE_PARALLEL_UPCXX_BACKEND=ON -DACTORLIB_USE_OPENMP_TASKS=OFF -DIS_CROSS_COMPILING=ON -build pond <PATH>; make`
Pond Task	`CXX="env UPCXX_THREADMODE=par upcxx -O3" cmake -DBUILD_RELEASE=ON -DBUILD_USING_UPCXX_WRAPPER=ON -DENABLE_FILE_OUTPUT=OFF -DENABLE_MEMORY_SANITATION=OFF -DENABLE_LOGGING=OFF -DENABLE_O3_UPCXX_BACKEND=ON -DENABLE_PARALLEL_UPCXX_BACKEND=ON -DACTORLIB_USE_OPENMP_TASKS=ON -DIS_CROSS_COMPILING=ON -build pond <PATH>; make`

Depending on the cluster configuration, the path to Metis and NetCDF needs to be provided manually. A generator for the SLURM scripts that were used for our experiments may be found in the actor-upcxx GIT repository (folder *jobscript-gen*).

Artifact Availability

a) Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

b) Hardware Artifact Availability: There are no author-created hardware artifacts.

c) Data Artifact Availability: There are no author-created data artifacts.

d) Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

e) List of URLs and/or DOIs where artifacts are available:

- **SWE:** https://github.com/TUM-I5/SWE
- **actorX10:** Not currently publicly available, contact Alexander Pöppl for access
- **SWE-X10:** Not currently publicly available, contact Alexander Pöppl for access
- **Pond and Actor Library:** https://bitbucket.org/apoeppl/actor-upcxx
- **X10 2.3.1:** http://x10-lang.org/releases/x10-release-231.html
- **UPC++:** https://upcxx.lbl.gov

Consideration for SCC

No.

Baseline experimental setup, and modifications made for the paper

f) Relevant hardware details: KNL Compute Nodes: Each node is a single-socket Intelő Xeon Phi Processor 7250 ("Knights Landing") processor with 68 cores per node @ 1.4 GHz

g) Operating systems and versions: SUSE Linux Enterprise Server 15

h) Compilers and versions:

- *Intel C++ Compiler:* icpc version 18.0.1.163 (gcc version 7.3.0 compatibility)
- *IBM X10 Compiler:* X10 2.3.1

i) Applications and versions:

j) Libraries and versions:

- UPC++ 2019.3.0
- Metis 5.1.0

k) Key algorithms: Actor Library, HLLE Solver

l) Input datasets and versions: N.A.

m) Paper Modifications: For the default version of SWE, the OpenMP parallelization was not functional, we re-enabled it in the build system and modified the code where necessary. We also added the HLLE solver (not part of the main repository. It works as a drop-in replacement to the other Riemann solvers. Finally, we had to slightly modify the build system to make it work on Cori. A GIT patch containing the modifications, and the HLLE solver are available at: https://bitbucket.org/apoeppl/actor-upcxx/downloads/

Changes necessary to run SWE-X10 on Cori are pushed to the SWE-X10 Git repository in the branch *fix_cori-compilation*. The C++ code the X10 compiler generates seems to trigger a bug in the Intel C++ Compiler, version 19. Furthermore, the newest Java version capable of running the X10 Compiler is Java 7, therefore, the *JAVA_HOME* variable needs to be pointed to such an installation.

The version of Pond and the actor library that has the functionality that was used for the test is marked in the actor-upcxx repository with the tag *pond-paper-submission-commit*. In some cases, the commits may not match the logs, this is due to changes in the job script generator that was used for the generation of the SLURM scripts for the tests.

n) Output from scripts that gathers execution environment information: Contains the default modules on Cori plus changes to run Pond on the KNL partition. Due to a system upgrade shortly before the submission, the previous environment could not be completely replicated, a number of previously available packages were removed.

1) modules/3.2.11.1
2) nsg/1.2.0
3) intel/19.0.3.199
4) craype-network-aries
5) craype/2.5.18
6) cray-libsci/19.02.1
7) udreg/2.3.2-7.0.0.1_4.23__g8175d3d.ari
8) ugni/6.0.14.0-7.0.0.1_7.25__ge78e5b0.ari
9) pmi/5.0.14
10) dmapp/7.1.1-7.0.0.1_5.15__g25e5077.ari
11) gni-headers/5.0.12.0-7.0.0.1_7.30__g3b1768f.ari
12) xpmem/2.2.17-7.0.0.1_3.20__g7acee3a.ari
13) job/2.2.4-7.0.0.1_3.26__g36b56f4.ari
14) dvs/2.11_2.2.131-7.0.0.1_7.3__gd2a05f7e
15) alps/6.6.50-7.0.0.1_3.30__g962f7108.ari
16) rca/2.2.20-7.0.0.1_4.29__g8e3fb5b.ari
17) atp/2.1.3
18) PrgEnv-intel/6.0.5
19) craype-mic-knl
20) cray-mpich/7.7.6
21) craype-hugepages2M
22) altd/2.0
23) darshan/3.1.7
24) gcc/7.3.0
25) cmake/3.14.4
26) cray-netcdf-hdf5parallel/4.6.1.3
27) upcxx/2019.3.2