2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM) | 978-1-6654-5410-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/PAW-ATM56565.2022.00009

Asynchronous Workload Balancing through Persistent Work-Stealing and Offloading for a Distributed Actor Model Library

Yakup Budanaz
Department of Informatics
Technical University of Munich
Garching, Germany
yakup.budanaz@tum.de

Mario Wille
Department of Informatics
Technical University of Munich
Garching, Germany
mario.wille@tum.de

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

***Abstract*—With dynamic imbalances caused by both software and ever more complex hardware, applications and runtime systems must adapt to dynamic load imbalances. We present a diffusion-based, reactive, fully asynchronous, and decentralized dynamic load balancer for a distributed actor library. With the asynchronous execution model, features such as remote procedure calls, and support for serialization of arbitrary types, UPC++ is especially feasible for the implementation of the actor model. While providing a substantial speedup for small- to medium-sized jobs with both predictable and unpredictable workload imbalances, the scalability of the diffusion-based approaches remains below expectations in most presented test cases.**

***Index Terms*—Asynchronous, Actors, Work-stealing, Distributed, Persistent, Offloading, UPC++, Library**

I. Introduction

Numerics of modern scientific applications introduce dynamic workload imbalances. Static mapping of the workload to compute nodes will fall short due to runtime deviations, and dynamic balancing of the workload is fundamental to minimize the time-to-solution and to not waste available resources [1]. For example, in adaptive mesh refinement (AMR, e.g., [2]–[4]), the accuracy of the solution will be adapted for certain regions, dynamically changing the workload in each refinement. In particle simulations, spatial domain decomposition will lead to imbalances when the domain is not homogeneous [5]. Vacuum regions will result in imbalances in workload, and the decomposition of the particles has to be dynamically changed to adapt for best performance. State-space search problems including unbalanced tree search, SAT, and N-Queens are often irregular and show unpredictable workloads [6], and therefore dynamic and predictive workload balancing is mandatory to maintain high performance.

In this work, we consider a solver for the shallow water equations (SWE) that avoids unnecessary computation by lazily activating patches of the computational grid only when a propagating wave enters the patch, thus dynamically changing the workload with each increment of the simulation time [7].

Workload imbalance can also be caused by the hardware, for example with features like dynamic voltage and frequency scaling (DVFS), where the frequency of the CPU is adapted dynamically the processing capabilities of each compute node may dynamically differ. Performance variability due to hardware, as reported in [8], can severely impede scalability. Even without faulty hardware run-to-run variability caused by the hardware [9] provides another reason why applications and runtimes need to dynamically migrate workload between compute nodes.

We implement a fully decentralized asynchronous reactive dynamic workload balancing feature for the distributed actor model library Actor-UPCXX[1], implemented with Unified Parallel C++ (UPC++) [10]. UPC++ is a C++ library that implements the asynchronous partitioned global address space model (APGAS). It provides one-sided remote *put* and remote *get* operations, and functions that can be executed on remote UPC++ ranks[2] called remote procedure calls (RPCs). The actor model [11] is an asynchronous message-driven model of concurrent computation, where the actor is the universal primitive model. Actors do not share their state (i.e., any simulation data), but communicate only through asynchronous one-sided messages. The messages sent are limited in size and the received messages are stored in buffers until their recipient consumes them. Discrete states of actors prevent data races and side effects, enabling the actor model for distributed computing. Various industry-oriented implementations of the actor model are already in use, such as Erlang [12] and the C++ Actor Framework [13]. The actor model is also a popular choice in network frameworks such as the Akka framework for Scala and Orleans [14], the framework for .Net. Charm++ [15] implements a computational model similar to the actor model and is being used in high-performance systems.

We present a simple diffusion-based approach [16, e.g.] for dynamic workload balancing in Actor-UPCXX and support both stealing and offloading of the workload, by persistently transferring actors between compute nodes. Actor stealing is based on work stealing [17, e.g.], where underloaded ranks steal actors from their overloaded neighbors; actor offload-

[1]Available under GPL at https://github.com/TUM-I5/Actor-UPCXX
[2]From hereon, we just refer to UPC++ ranks as ranks

978-1-6654-5410-0/22/$31.00 ©2022 IEEE
DOI 10.1109/PAW-ATM56565.2022.00009

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.

ing is based on task sharing, where overloaded ranks share their actors with underloaded neighbors. Our implementation requires no synchronization and is decentralized and fully asynchronous. The results are analyzed for whether a simple, fully asynchronous implementation of diffusion-based load balancing can mitigate dynamic load imbalances caused throughout a time-dependent simulation.

In Sec. II we discuss dynamic workload balancing approaches that are comparable to actor stealing and offloading and classify them depending on their fundamental properties. Section III describes the implementation of Actor-UPCXX and our dynamic workload balancing schemes, followed by the evaluation of the strategies in Sec. IV. Finally, we discuss our results in Sec. V, followed by a conclusion and presentation of possible future work for the actor-based load balancer in Sec. VI.

II. Related Work

To aid the following discussion and overview of related work on dynamic load balancing strategies, in runtimes as well as in applications, we first outline a classification of such strategies. Regarding centrality, dynamic load balancing strategies can be grouped as

(1) centralized,
(2) decentralized,
(3) hierarchical.

In a centralized strategy, a central node takes all decisions regarding workload redistribution based on workload metrics gathered during the runtime. The serial nature of centralized load balancing introduces a bottleneck that will create scalability issues, especially if the amount of memory is limited. Mapping of computational meshes or over-decomposed systems to computational nodes via a serial graph or hypergraph partitioner falls into the centralized category [18]. Actor-UPCXX adopts a similar approach for the initial workload distribution, where the METIS [19] graph partitioner is used to assign the actors to nodes under a cost model. The bottleneck of the centralized approaches can be mitigated by decentralized approaches, where the decisions regarding workload redistribution will be taken locally without the need for synchronization, then based on partial knowledge of the system. Therefore, decentralized approaches yield less effective workload redistribution compared to centralized strategies while offering better scalability. Hierarchical approaches try to overcome scalability problems by exploiting a tree-based hierarchical approach. Each node of the tree balances the load across the processors of its subtrees, and the root node acts as the leader of the group, similar to the central node in centralized approaches [20].

The dynamic workload balancing methods implemented for Actor-UPCXX are decentralized methods and do not require any synchronization. Therefore, we focus on other decentralized methods in the following. Decentralized workload balancers can be further classified as:

(a) push-based or
(b) pull-based.

In push-based variants, overloaded ranks offload workload to underloaded ranks; in pull-based variants, underloaded ranks steal workload from overloaded ranks. A common step for both push-based and pull-based variants is the exchange of workload information, which often occurs periodically. How often and when the load balancing methods are invoked is crucial for the performance of the system, as migration of workload incurs additional communication. The invocation of the load balancer can be grouped into two categories:

(i) periodical or
(ii) reactive.

If the load balancer is invoked periodically, migration phases are introduced between any two computation phases. Migration phases are easy to achieve with applications that have implicit synchronization points, such as simulations that synchronize after each discrete time step. Periodic load balancers can be integrated into such applications without requiring additional explicit synchronization. In a reactive load balancer, the load balancer will be invoked when a node identifies an imbalance that requires workload rebalancing.

Charm++ [15] implements an asynchronous message passing parallel programming paradigm similar to the actor model and offers many of the discussed workload balancing strategies. Charm++ defines multiple types of objects called '*chares*' that share data through messages. Charm++ allows asynchronous invocation of chares. Over-decomposition due to the chare objects allows the Charm++ runtime system to utilize dynamic load balancers [21]. Charm++ supports a plethora of load balancers for dynamic workload balancing and provides a load balancing framework for easy implementation of load balancing strategies that can be integrated into the framework.

A simple diffusion-based load balancer for Charm++ that implements a local neighborhood workload averaging scheme is NeighborLB [18]. A push-based load balancing scheme of Charm++ comparable to actor offloading is the Grapevine load balancer [22], which is a periodic load balancer that creates a partial representation of the global load of the system through epidemic communication (gossip protocol) during the information propagation phase. Grapevine then uses the information collected for the probabilistic transfer of the workload between ranks. It shows performance improvements even up to 131k core counts (maximum number of cores tested in the paper) in imbalances induced by adaptive mesh refinement.

Another push-based load balancer implemented in Charm++ is the PackDrop load balancer [23]. PackDrop utilizes grouped tasks and migrates tasks in batches while preserving the locality of the tasks. The average load is computed with a global reduction of the per-rank load. Where underloaded ranks inform overloaded ranks via gossip protocol to further decide on the migration of tasks. The algorithm requires synchronization between the information propagation and task migration phases. Grapevine in general outperforms PackDrop

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.

Table I: Overview of load balancers similar to actor stealing and actor offloading following the classification of Sec. II.

Strategy	Centrality	Periodicity	Synchronization	Initiator	Persistence
GrapevineLB [22]	decentralized	periodical	barrier	push	persistent
PackDropLB [23]	decentralized	periodical	quiescence detection and reduction	push	persistent
PackStealLB [24]	decentralized	periodical	quiescence detection and reduction	pull	persistent
Chameleon [25]	decentralized	reactive	none	push	single-shot
GLB [26]	decentralized	reactive	none	pull	single-shot

Synchronization describes whether the strategy needs any type of synchronization, e.g., in the form of barriers, reductions of variables or quiescence detection through message counting. *Initiator* describes whether the scheme is a push- or pull-based strategy. *Persistence* describes whether the stolen tasks live in the migrated rank: single-shot means that the migrate task's lifetime spans only a single execution, regardless of whether the results are sent back after the computation, whereas in the persistent case the task's lifetime spans over multiple execution and continues to live on the node it is migrated to.

when the communication latency is less and underperforms in systems with slower interconnect.

Work-stealing is a prevalent choice for pull-based work migration. Work-stealing has a wide use in shared memory systems. The Cilk [17] runtime system and OpenMP schedulers [27] employ work-stealing schedulers. A distributed pull-based load balancing scheme that implements work stealing for distributed memory systems is the PackStealLB [24] of Charm++. In PackStealLB, the underloaded thieves steal a packed group of tasks from the victim ranks. Unlike PackDrop, it does not require global synchronization, but the load balancer has to be invoked, thus having a migration phase between computation phases. In the evaluation, PackSteal shows better performance compared to the PackDrop and Grapevine load balancers at 960 cores spanning over 20 compute nodes.

Samfass et al. [28] implement a distributed task stealing load balancer for fine-grained load imbalances on top of existing load balancers of the sam(oa)^2 PDE framework. The work stealing scheme is more reactive compared to the previously mentioned Charm++ load balancers, as the stealing can be initiated by any process that has detected an imbalance through continuous polling of the workload. The scheme utilizes communication threads to quickly process steal requests. The work stealing approach is superseded by Chameleon [25] which adopts a push-based approach. Charm++ is reported to outperform Chameleon in cases of higher imbalances, whereas temporary stealing of tasks in Chameleon is more suitable for more unpredictable and short-lived imbalances [29].

Saraswat et al. [30] implement work-stealing for distributed memory systems in X10 [31] with lifeline graphs. Lifeline graphs are chosen to be cyclic hypercubes and stealing ranks choose their victims from the lifeline graphs. GLB [26], [32] implements a work-stealing algorithm also based on lifeline graphs for the map-reduce model in X10. In the map-reduce model, every worker applies the map function on the local data, and in the reduction phase, the mapped results are reduced depending on the user-defined rules of the reduce method. In GLB ranks first attempt to steal from w random victims; if all attempts fail, then the rank tries to steal from its lifeline. Every rank polls periodically for workload information to detect imbalances. The computed results persist on the stealing rank and are used during the reduction phase.

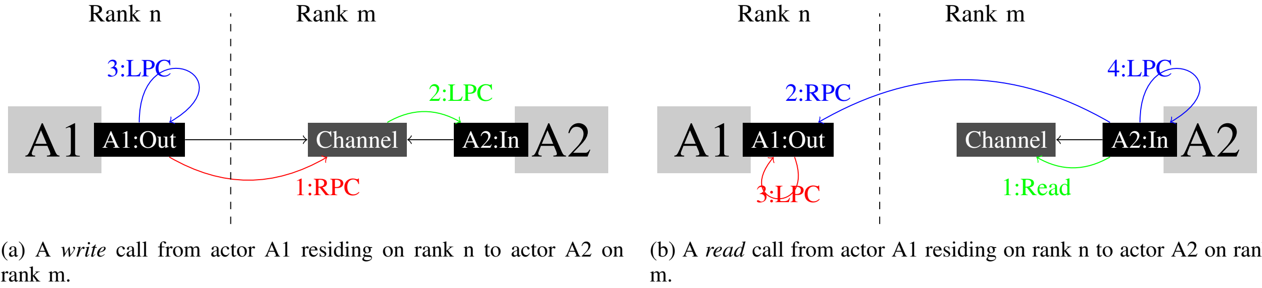
Fully decentralized load balancing schemes can achieve good scaling (e.g., [22], [33]), however, it requires careful engineering and aggressive optimizations. Scalable load balancers indicate a pattern of common properties: minimization of communication between ranks during information exchange, for example, through remote direct memory access (RDMA) [33], [34], or with epidemic-like gossip protocols [22], [23]. For stealing-based load balancers, mitigating the additional cost of failed attempts through preemptive aborting [33] or eliminating failures completely [34] is required for high scalability. Load balancers also utilize asynchronous communication to overlap communication with computation to further hide the cost of migrating work from the application [35], [23], [24], [23], [28], [29]. A summary of the properties of the mentioned load balancers can be seen in Table I.

III. Implementation

We extend Actor-UPCXX [36], a C++ library that implements the actor model using UPC++. Each actor is an object that is identified by its unique name and assigned to a part of the parallel simulation. An actor and its data can be serialized for sending it to another rank using UPC++. The user can specify certain operations the actor executes when a predefined state, which typically depends on the messages received from other actors, is reached. Actors are not allocated in the shared memory but every address is assigned to a global pointer in the global address space. Information regarding the global address of the actor and its name are saved in a global hash map that is replicated on every rank. The connections of actors are saved in a global graph structure that is also replicated on every rank.

In our implementation, actors communicate through one-sided asynchronous messages. The communication is determined by one-sided channels that connect two actors (which we call *neighbors*). Actor-UPCXX does not employ a central message queue, and the messages are buffered by the actors in a statically sized array. A *write* call to another actor can be issued through an RPC when there is space available in the remote buffer. The *write* call then creates local procedure call (LPC) callbacks to update the available size as depicted in Fig. 1a. Similar to *write*, a *read* calls transfers the message for consumption and produces callbacks to update the available sizes as depicted in Fig. 1b. An actor may *execute* its user-defined task, e.g., when every port of the actor has at least one message. Termination is detected when every actor in the system has terminated.

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.



(a) A *write* call from actor A1 residing on rank n to actor A2 on rank m.

(b) A *read* call from actor A1 residing on rank n to actor A2 on rank m.

Fig. 1: Structure of read and write calls. RPC is a remote procedure call, a function that is executed on a remote rank; LPC is a local procedure call, which is a callback attached to a completion (e.g., completion of an RPC).

A. Enabling Migration of Actors

In this paper we present actor stealing and actor offloading to dynamically migrate actors between UPC++ ranks. Actor stealing and actor offloading are both diffusion-based load balancers and share common features. The main difference between the two strategies lies in their communication pattern. Actor stealing is a pull-based strategy, whereas actor offloading is push-based.

In actor stealing an underloaded rank chooses a victim rank to steal an actor. The stealing rank tries to reserve (cf. Alg. 1) the victim actor. The reserve procedure works as follows: marking the actor for migration (cf. Alg. 2), pinning its neighboring actors, and stopping the victim actor. Then the actor is transferred by sending the serialized actor with its data and buffered messages via remote procedure calls (RPCs). After the serialization is completed, pinned actors are unpinned and the migrated actor is restarted (cf. Alg. 3).

In actor offloading, a rank that detects an imbalance may decide to offload an actor to a rank that is underloaded. Similar to the actor stealing approach, the actor has to be reserved (cf. Alg. 1), marked (cf. Alg. 2), and the neighboring actors have to be pinned. After the transfer is completed, the pinned neighbors are unpinned and the offloaded actor is restarted. The offloading is performed by the actor's local rank, therefore it uses fewer RPCs during the reservation and pinning of the actors (cf. Alg. 4).

Both migration strategies perform every action through asynchronous calls (RPCs). All procedures create RPCs if the input actors reside on remote ranks. In the provided algorithms (i.e., 1–4) this is depicted as a right arrow ($\rightarrow$) pointing to its recipient rank.

The workload imbalance within a set of ranks R is defined as the factor of maximum workload to minimum workload. The imbalance is considered large enough for migration, if the imbalance factor σ is higher than a pre-defined constant which is typically given as $\sigma = 1.05$ (cf. [24], [23]).

To model the workload, our implementation currently offers

- task counting and
- time counting.

Task counting is inspired by the asynchronous many task (AMT) paradigm [35], [18], where work is encapsulated into tasks. Actor-UPCXX considers every execute operation of an actor as a task. The amount of available tasks can be saved in a rank-local variable that is in the global address space allowing effective transfer through RDMA. Task counting assumes that every task has identical cost. Time counting gathers the cost of every actor that depends on the time spent executing the actor.

Before the migration of an actor v from rank a to rank b can commence, the actor v needs to be reserved for migration. Reservation includes marking the actor v for a possible migration, pinning the neighboring actors that directly communicate with the actor so that they are not migrated during the migration of v, and stopping the actor v. Marking and pinning are two different locking operations used to facilitate asynchronous migration. In the marking procedure, an actor is marked for migration. In pull-based methods, the stealing rank marks the actor. In push-based methods, the offloading rank marks its own actors. Being marked for migration prevents the actor from participating in any other migration attempt until it is unmarked. The mark informs the runtime of a possible migration of the victim actor. The pinning procedure is required as the actors that communicate with the migrating actor (i.e., their neighbors) are not allowed to migrate during the migration of the victim actor. After a successful reservation (i.e., marking the victim and pinning its neighbors) the actor is serialized and transferred over the interconnect. Serialization requires move constructors and an implementation of the serialization interface provided by UPC++.

Due to the implementation of the used actor library and the design choices of the stealing procedure, reserving actors and using locks on actors is unavoidable. Both start and the destination ranks will continue with the execution of other actors during the migration procedure, as it is fully asynchronous. Due to the implementation of the channels between actors (see Figs. 1a and 1b), the implementation requires the neighbors of the victim actor v to flush their messages to actor v and notifies the neighboring actors that they are not allowed to write messages to channels of v until the migration is completed.

There are three constraints for the pinning and marking of actors:

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Reserve(v):
Marks the victim actor, pins the neighborhood of the victim actor and stops the victim. The pinning and marking of an actor may fail – therefore it has to be checked whether the calls succeed. The neighborhood is denoted with Γ.

```
1  Procedure reserve(v):
2  marked ← mark(v);
3  if marked then
4  |   pinned ← pin(Γ(v));
5  |   if pinned then
6  |   |   stopped ← stop(v);
7  |   |   if stopped then
8  |   |   |   return true;
9  |   |   else
10 |   |   |   unmark(v);
11 |   |   |   unpin(Γ(v));
12 |   |   |   return false;
13 |   |   end
14 |   else
15 |   |   unmark(Γ(v));
16 |   |   return false;
17 |   end
18 else
19 |   return false;
20 end
```

Algorithm 2: Mark(v):
Attempts to mark an actor v for migration; may issue an RPC to a remote rank, if the recipient actor resides on a remote rank.

```
1  Procedure mark(v):
2  if v.at() == rank_me() then
3  |   return v.mark(rank_me());
4  else
5  |   return v.mark(rank_me()) → v.at();
6  end
```

1) A marked actor can't be pinned,
2) a pinned actor can't be marked,
3) a terminated actor can't be pinned or marked.

Actor-UPCXX supports an additional constraint to limit the count of concurrently marked actors on a given rank by n. Every new mark attempt for a steal will fail as long as there are n marked actors. If the pinning of any actor within the neighborhood fails, then the pinning of the neighborhood fails. To prevent side effects, any previously pinned or marked actors have to be respectively unpinned or unmarked.

B. Actor Stealing

Actor stealing is the adaptation of the task stealing approach for the actor model. Underloaded ranks that are idle or have detected a significant load imbalance may initiate the procedure to steal an actor from a remote rank.

Our implementation does not require entering a migration phase and is fully reactive, similar to [25]. A steal is initiated when a rank detects an imbalance above a certain imbalance factor. We do not wait until the rank is idle, as it would prevent overlapping communication with computation. Reserving and locking the victim actor is necessary due to the absence of a migration phase but the steal procedure does not require any synchronization. Stolen actors persist on the rank to which they have been migrated.

The actor stealing approach implemented is (2) decentralized, (b) pull-based, and (ii) reactive. There is no migration phase and the migration procedure can be invoked by any rank that has detected a workload imbalance above a certain imbalance factor. The stealing procedure requires no synchronization.

The stealing procedure begins with the choice of the victim rank. When to initiate the stealing procedure is an integral part of the implementation. Initiating a steal when the rank has run out of work (meaning no actor can execute) is against the principle of overlapping communication with computation as the stealing rank will have to wait idle until new messages or the stolen actor arrives. Therefore, every rank actively polls for the workload information of remote ranks to detect imbalances.

The set of remote ranks that can be chosen as victims can be set as:

- global or
- local.

If set to *local* then the rank r may choose neighboring actors as victims. Neighboring actors of rank r either send messages to r or receive messages from r. If set to *global* then any actor residing on any rank can be chosen as the victim.

Further, the type of polling can be set to:

- random or
- busy.

If set to *random*, a random rank from the available set of victim ranks is selected as the victim rank. A steal request is issued with the workload of the stealing rank provided. Even though the victim rank always acknowledges the steal request, it only serves the request, if its workload is higher than that of the stealing rank and above a pre-defined imbalance factor. If the polling type is set to *busy*, the remote rank that has the highest workload from the set of allowed victim ranks is chosen as the victim. The victim may again refuse to let its actors to be stolen, if the workload difference falls under the imbalance factor before the steal request is processed.

Before stealing an actor v, it has to be successfully reserved. After the reservation, the victim actor is disconnected from its neighbors such that no neighbor can send messages to the victim actor. After the migration of the actor from victim to stealing rank, the closed connections are reconnected, the pinned actors are unpinned and the actor is then restarted completing the steal procedure.

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.

Algorithm 3: Steal(v):
Disconnects the victim actor from its neighbors, serializes and sends the actor to its new rank. After the transfer is completed, reconnects the stolen actor and unpins the neighbors of the actor. The neighborhood is denoted with Γ.

```
1  Procedure steal(v):
2  workloads ← gather_workload();
3  v ← find_victim(workloads);
4  reserved ← reserve(v, Γ(v)) → v.at();
5  if reserved then
6  |   disconnect(v, Γ(v)) → v.at();
7  |   migrate(v, rank_me()) → v.at();
8  |   reconnect(v, Γ(v));
9  |   restart(v);
10 |   unmark(v);
11 |   unpin(Γ(v));
12 end
```

Algorithm 4: Offload():
Disconnects the actor to be offloaded from its neighbors, serializes and sends the actor to its new rank. After the transfer is completed, reconnects the actor and unpins the neighbors of the actor. The neighborhood is denoted with Γ.

```
1  Procedure offload():
2  workloads ← gather_workload();
3  (v, to) ← find_recipient(workloads);
4  reserved ← reserve(v, Γ(v));
5  if reserved then
6  |   disconnect(v, Γ(v));
7  |   migrate(v, to) ;
8  |   reconnect(v, Γ(v)) → v.at();
9  |   restart(v) → v.at();
10 |   unmark(v)→ v.at();
11 |   unpin(Γ(v));
12 end
```

C. Actor Offloading

Actor offloading is the adaptation of the task offloading approach for the actor model. The implementation is similar to actor stealing, the main difference lies in the change of communication between start and endpoints and the workload exchange. Every rank checks regularly for the load imbalance within the neighboring ranks. The choice of the interval is important as we do not support batch migration of actors. If an overworking rank detects a significant load imbalance above the imbalance factor, it initiates the offloading procedure to a neighboring rank that has the least workload. Our implementation of offloading also does not require entering a migration phase and is fully reactive. The actors offloaded again persist on the rank they are migrated to. Actor offloading is (2) decentralized, (a) push-based and (ii) reactive.

In contrast to actor stealing, an offload request can't be rejected. Before offloading an actor v, it has to be successfully reserved. After the reservation, the victim actor is disconnected from its neighbors such that no neighbor can send messages to the victim actor. After the migration of the actor from victim to stealing rank, the closed connections are reconnected, the pinned actors are unpinned and the actor is restarted completing the offloading procedure. The integral part of offloading is the same as it was in the steal procedure, but the communication start and endpoints are reversed.

IV. Evaluation

All following results have been obtained from the CoolMUC-2 Cluster hosted by the Leibniz Supercomputing Center (LRZ)[3], which is equipped with 28-way Intel Xeon E5-2690 v3 compute nodes and FDR14 Infiniband interconnect. The UPC++ version used is 2022.03.0. UPC++, and Actor-UPCXX were compiled with the Intel oneAPI compilers version 2021.4.0. OpenMPI v4.1.2 and HWLoc 2.6.0 are used by the communication backend of UPC++, GASNet-EX [37] for the job launch. Parallelization is achieved using the UPC++ sequential backend by creating one rank for each physical core.

In the following, three scenarios are presented to test the implemented dynamic workload balancing strategies: static workload (IV-C), node slowdown (IV-D) and lazy activation (IV-E). The static workload scenario has a predictable and static workload. The node slowdown scenario creates an unpredictable dynamic workload by creating an artificial slowdown on a subset of nodes. The lazy activation scenario creates a coarse, dynamic, and relatively predictable dynamic workload imbalance. The presented scenarios are used to test the implemented strategies under various use cases. Every strategy and scenario is tested using Pond (IV-A), a proxy application for shallow water equations.

A. Pond – a Proxy Application for Shallow Water Simulations

The dynamic load balancing strategies implemented in Actor-UPCXX are evaluated on Pond [36], [38], a proxy application that implements a finite volume solver – following the modeling approach and discretization presented by LeVeque et al. [39] – for the shallow water equations (SWE):

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = 0, \quad (1)$$

with water height h, and momentum hu and hv in the two spatial directions x and y.

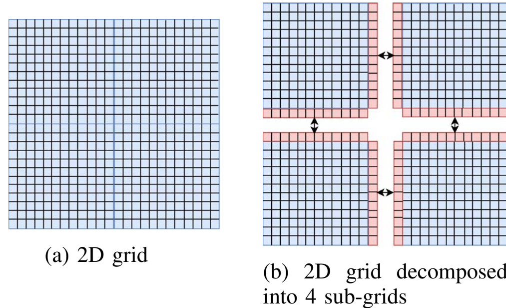
Following the actor model and using the Actor-UPCXX library, Pond organizes the 2D Cartesian discretization grid as patches (see Fig. 2): every patch is assigned to an actor permanently. Pond uses over decomposition of actors to compute cores, by assigning multiple actors (i.e., n by n patches) to each compute core. Ghost layers (see Fig. 2 and below) are used to synchronize data between patches.

[3]https://doku.lrz.de/display/PUBLIC/Linux+Cluster

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.

Name	Centrality	Periodicity	Synchronization	Initiator	Persistence
ActorSteal	decentralized	reactive	none	pull	persistent
ActorOffload	decentralized	reactive	none	push	persistent

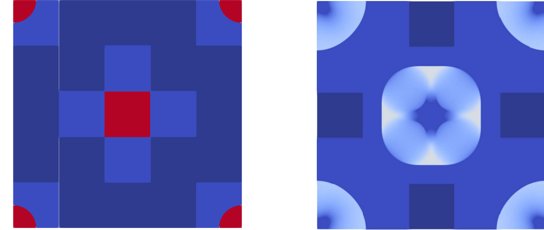
Table II: Properties of the implemented actor stealing and offloading approaches, to be compared with Table I.



(a) 2D grid

(b) 2D grid decomposed into 4 sub-grids

Fig. 2: An example grid decomposed into four patches, which Pond will assign to four actors. Cells of the patches are marked in blue. Ghost layers are marked in red. Arrow directions show the communication of the one-sided messages (communication of Pond actors is bidirectional).



(a) Initial decomposition: the center patch and those with displacements on the four corners are active, also the patches that share their patch boundary with the central displacement are active.

(b) As the waves propagate inwards from the corners and outwards from the central displacement, new patches are activated – only patches on the edges that have not met a propagating wave remain idle.

Fig. 3: Illustration of lazy activation for a 2D grid with 5×5 patches. Active patches are marked with a lighter shade of blue. Colors blue $\rightarrow$ white $\rightarrow$ red indicate water height (from low to high).

In each time step of the simulation, each patch performs three operations: (1) Update the cells in the ghost layer according to boundary conditions or with values communicated by neighbor patches. (2) For each edge compute approximate fluxes between the adjacent cells (via computing an approximate Riemann problem) and accumulate the fluxes as net updates. (3) Update the cell quantities using the net updates and the time step size Δt.

After each time step, ghost cell values are updated, by sending one-sided messages between Pond's actors. The simulation starts with such a ghost-layer exchange. In Fig. 2 the domain in the left subfigure (cf. Fig. 2a) is decomposed into four patches on the right. Ghost layers that hold the information of the neighboring cells are marked red.

B. Load Balancing Scenarios - Experiment Description

We have tested the dynamic load balancing strategies on the following scenarios: *static workload*, *node slowdown* and *lazy activation*. All presented scenarios are strong scaling tests. Static workload is a scenario where the computational costs of executing actors do not change over time. It acts as a baseline. Node slowdown mimics faulty hardware by slowing down the execution of actors on certain ranks for a specified time interval. This aims to create an unpredictable hardware-induced load imbalance. Lazy activation keeps patches idle at the start, and only activates them ("lazily") as soon as the wave propagates into the respective patch [7]. Idle actors do not have to update the (initial) solution but check for incoming waves at the boundaries. Hence, idle actors have much smaller costs compared to active actors. This creates a coarse imbalance at the start of the simulation where then actors are activated as the simulation progress. For an example of decomposition and the resulting active patches depending on the used scenario, see Fig. 3.

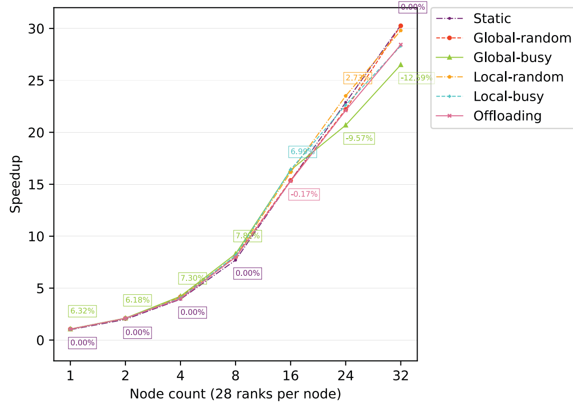
Our experimental setup is configured as follows: for the static workload and node slowdown scenarios, Pond runs on an 18000×18000 grid. The grid is divided into 250×250 patches with 1 patch per actor and with a simulation time of 1.0 seconds. For the victim choice strategy, in static workload, time counting is used and in node slowdown, task counting is used. For the node slowdown scenario, the ranks with ids from 0 up to 4 times the number of compute nodes are chosen for the artificial slowdown. These ranks are slowed down during the time interval $[60.0; 360.0]$ by a factor of 3. Task counting is used for the victim choice. For the lazy activation scenario, the grid size is increased to 36000×36000. Time counting is used as the strategy for victim selection.

C. Static Workload

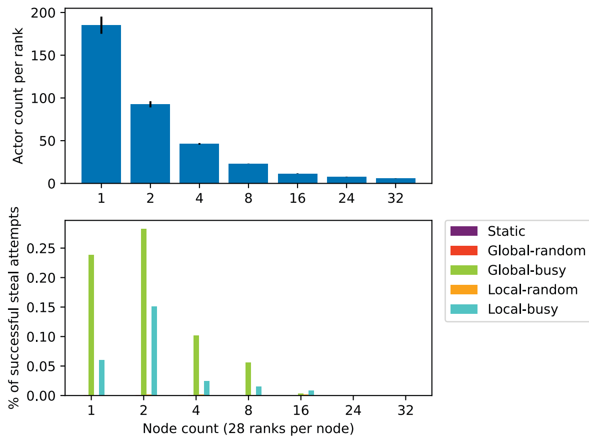
The initial workload distribution of the actors can be modeled as a graph partitioning problem and the static mapping of actors to compute nodes is calculated with METIS. Any workload imbalance throughout the execution, if present at all, may only be caused by the hardware or due to imbalances in the partitioned graph. The scenario aims to analyze the impact of dynamic workload balancing strategies with static and well-balanced workloads. The static workload scenario acts as a base case where the system has a predictable workload, it aims to test the impact of the dynamic load balancing strategies.

The migration strategies manage to improve the performance of up to 24 nodes (cf. Fig. 4a). Global-busy, local-

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.



(a) Speedup relative to the “Static” basic case on one node without migration. Boxes annotate best and worst results for each node count (runtime improvement compared to “Static”, box color indicates the respective strategy).



(b) Top: Number of actors per rank, with variance plotted on top of the bar; bottom: percentage of successful steal attempts.

Fig. 4: Speedup (a) and actor statistics (b) for the “Static Workload” scenario for 1 to 32 nodes.

busy, and local-random improve the performance roughly by 7% up to 16 nodes while the performance of global-random and offloading drops to 0% at 16 nodes. For 24 and 32 nodes no migration strategy attempts to migrate actors. Due to the additional cost of workload exchange, the time-to-solution increases roughly by 5% on average. Global strategies collect the runtime information of every rank compared to local strategies only collecting the workload information of their neighbors. For this predictable, well-balanced workload, global strategies have a natural disadvantage due to the bigger overhead of global workload exchange compared to the local strategies that collect the workload of their neighbors only. The graph partitioning shows less variance as the number of partitions increases (cf. Fig. 4b). In small node counts, there is a high variance enabling room for improvement through migrations. For any job that is partitioned between more than four compute nodes, every rank has either c or $c+1$ actors. Therefore, at higher node counts fewer migration attempts will be performed, and the attempts are less successful.

D. Node Slowdown

The node slowdown scenario is an artificial scenario to test the performance under unpredictable workload imbalance, for example, which might occur due to variable hardware performance. This scenario aims to create such a dynamic and unpredictable workload imbalance by artificially slowing down a subset of ranks to test the reaction speed and quality of the implemented migration strategies.

A subset of ranks $[a;b)$ is slowed down artificially during the time interval $[t1;t2]$. Any actor residing on a rank $r \in [a;b)$ that is affected by the slowdown is forced to require triple the time to complete the execution of a task. This is achieved by measuring the execution time and waiting idly for twice the execution time. The node slowdown scenario has a heavy impact on the throughput as it causes a bottleneck in the system. An actor requires the messages from its neighbors from time step t_n to compute the time step t_{n+1}. The bottleneck's effect propagates to neighboring ranks, causing them to wait additionally due to the slowed-down processing of other actors.

In addition to the stealing modes local-busy, global-random, etc., we define an additional mode “Ideal LB” that shall reflect the theoretical upper bound for the speedup achievable in a system without communication latency or synchronization cost. We assume that Ideal LB immediately balances the workload at the start $t1$ of a slowdown by instantly transferring $\frac{2}{3}$ of the actors of the affected ranks $r \in [a,b)$ evenly to the remaining ranks (and transfers them back immediately at the end of the interval $t2$).

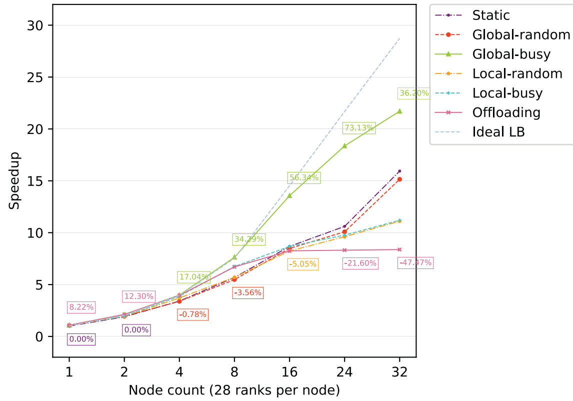
In the results we see that all pull-based approaches react better than “Offloading” (which is push-based) in unpredictable imbalances (cf. Fig. 5a). Global strategies now achieve better performance than local strategies, as the overhead is overcompensated by better migration decisions. On 32 compute nodes, it can be observed that the amount of successful steals drops close to 0 resulting in less improvement of the performance (cf. Fig. 5b).

E. Lazy Activation

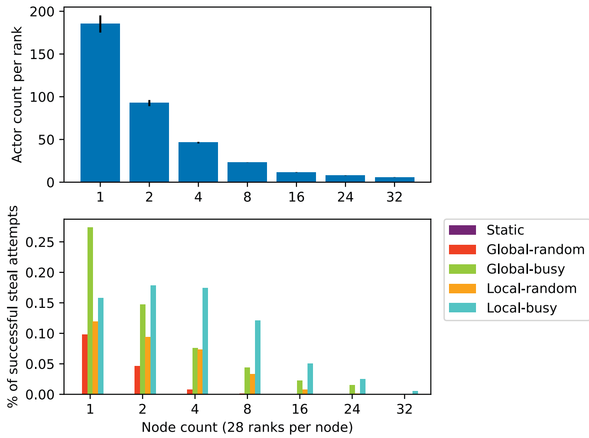
In the lazy activation scenario, the workload is determined by the evolving solution. Actors are activated as waves enter the respective patch, such that workload changes dynamically until every patch is activated. More migrations are therefore carried out in the beginning to even out the initial imbalance, whereas fewer and fewer migrations are needed the more patches are dynamically activated in the simulation.

We again define a stealing mode “Ideal LB” that shall reflect the theoretically achievable upper bound for the speedup in a system that can perform an instantaneous transfer of actors

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.

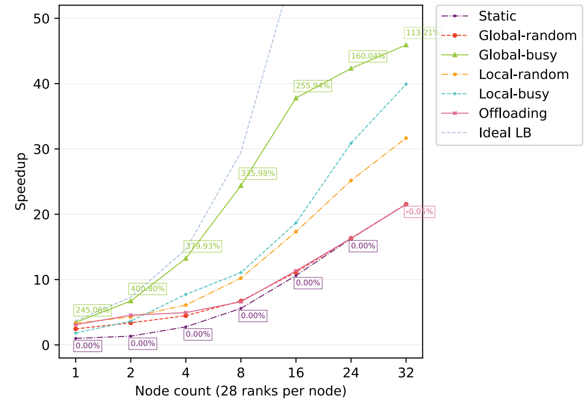


(a) Speedup relative to the "Static" basic case on one node without migration. Boxes annotate best and worst results for each node count (runtime improvement compared to "Static", box color indicates the respective strategy).

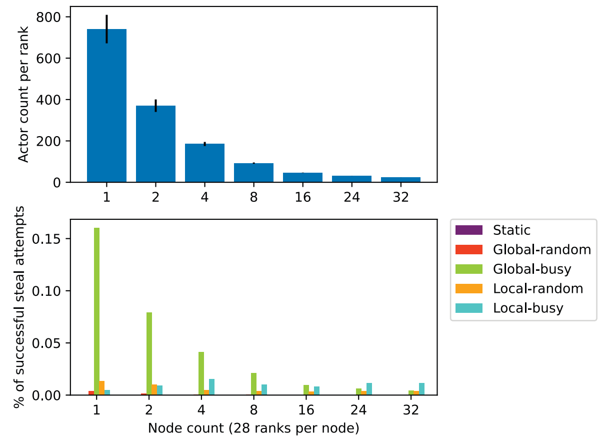


(b) Top: Number of actors per rank, with variance plotted on top of the bar; bottom: percentage of successful steal attempts.

Fig. 5: Speedup (a) and actor statistics (b) for the "Node Slowdown" scenario for 1 to 32 nodes.



(a) Speedup relative to the "Static" basic case on one node without migration. Boxes annotate best and worst results for each node count (runtime improvement compared to "Static", box color indicates the respective strategy).



(b) Top: Number of actors per rank, with variance plotted on top of the bar; bottom: percentage of successful steal attempts.

Fig. 6: Speedup (a) and actor statistics (b) for the "Lazy Activation" scenario for 1 to 32 nodes.

without migration cost or communication latency. The time-to-solution of the Ideal LB is determined as the average time spent in executing actors.

Global-busy now achieves the highest improvement in performance (cf. Fig. 6a). The performance of the global-random and the offloading strategies degrades swiftly, even for small node counts. The global-random strategy can only migrate actors in a tiny fraction of attempts and the offloading strategy can't control the interval to offload actors. As the average number of actors per rank decreases with the increasing node counts, fewer steal attempts succeed. Local strategies require more migrations than global strategies to achieve a similar quality in load balancing as actors need to hop through multiple ranks to travel between two non-neighboring ranks. Local strategies show good scalability as their performance does not degrade and even perform better on higher node counts, but still require more migrations than global strategies to achieve a similar load balancing quality (cf. Fig. 6b).

V. Discussion

Our implemented strategies for dynamic load balancing improve the runtime in predictable and unpredictable load imbalances. The pull-based global-busy strategy was able to improve runtime in node slowdown and lazy activation scenarios on every node count tested, while it caused a slight decrease in performance in the static workload scenario on high node counts due to the additional cost of workload exchange. The cost of workload exchange is much less in the local-busy strategy, as the local strategy only exchanges the

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.

local workload information, therefore it performs better in the static workload scenario. The global-busy strategy outperforms the local-busy strategy in the node slowdown as well as in the lazy activation scenarios, as the coarse imbalance compensates for the additional overhead of workload exchange.

Random strategies are not able to compete with busy strategies, as they result in a high percentage of failed attempts, which puts a strain on victim ranks that need to handle steal requests. The global-random strategy underperforms the local-random strategy, as the global-random strategy rarely succeeds in finding a rank that is able to serve the steal request.

The push-based offloading strategy fell short in predictable as well as in unpredictable workload imbalances due to its periodic workload exchange, thus not being able to migrate enough actors to achieve reasonable speedup.

The type of workload imbalance strongly impacts the performance of the used strategy. For example, in the lazy activation scenario, new patches that are activated are in the proximity of the actors that were previously activated. Thus local strategies have an advantage as they will be able to distribute the imbalance easily. In the case of node slowdown, the imbalance is condensed to a set of ranks and the local strategies are at a disadvantage as actors need to hop between multiple ranks to reach ranks that are not neighbors of the slowed-down ranks. Global strategies can migrate actors from the affected ranks with a single transfer.

Dynamic workload balancers can be useful even without load imbalances, as seen in the static workload scenario. As long as the imbalance is large enough to compensate for the additional workload exchange which can be seen up to 24 nodes.

VI. Conclusion & Future Work

We have provided a fully asynchronous and decentralized diffusion-based dynamic workload balancing scheme for the actor model that requires no synchronization and no migration phase. UPC++ and the higher level APGAS model accelerate the implementation of the phase-less migration strategies. UPC++ provides templates to serialize arbitrary types, allowing easy implementation of serialization to migrate actors and their data between remote ranks. The asynchronous nature of UPC++ enables the implementation of actor migration as a chain of RPCs, thus preventing the need for any synchronization between ranks. Features like remote memory access enable easy access to remote workload information.

Failed steal attempts become a hurdle in strong scaling due to the small number of actors per rank. Allowing steal attempts to fail and thus implementing a phase-less migration strategy has the advantage that it can instantly adapt to unpredictable imbalances providing swift reaction times in low- to medium-sized node counts.

We find that to provide scalable stealing and offloading approaches, it will be necessary to decrease the percentage of failed steal attempts and to decrease the number of total migrations required to balance the freshly induced imbalances. Changing the over-decomposition of the system by dividing the problem into fewer actors will result in fewer migrations required, but will result in more failed steal attempts. Viable approaches to decrease the required number of migrations and failed steal attempts could follow the approaches of the Charm++ load balancers:

- abolishing pinning through appropriate data structures,
- allowing migration of actors in packs.

The requirement to pin the neighbors of migrating actors can be avoided by decoupling the communication from the actor executions. Decoupling can be achieved through a central messaging queue similar to Charm++ or the data warehouse used in Uintah [35]. In the case where a message is sent to a migrating actor that may not accept messages, the central message queue will ensure that the message is buffered until the actor is migrated and reroute the message after the migration is completed. With central message queues, the pinning will be no longer necessary as steps such as $disconnect$ and $reconnect$ in the migration procedures can be skipped. Without the requirement for pinning, the actors can be packed and migrated in bulk in case of coarse load imbalances similar to PackStealLB [24] and PackDropLB [23] to decrease the number of migrations (and attempts) necessary to balance coarse imbalances. A central messaging queue can still be implemented with UPC++ while preserving the asynchronous nature of the actor model.

Acknowledgements

This research was funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft), Project number 14671743, TRR 89 Invasive Computing. We would like to thank Philipp Samfass for his advice, which has been of great help during this research, and Alexander Pöppl for laying the foundations with Actor-UPCXX and Pond.

References

[1] B. Hendrickson and K. Devine, "Dynamic load balancing in computational mechanics," *Computer Methods in Applied Mechanics and Engineering*, vol. 184, no. 2, pp. 485–500, 2000.

[2] C. Burstedde, L. C. Wilcox, and O. Ghattas, "`p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[3] D. E. Charrier, B. Hazelwood, and T. Weinzierl, "Enclave tasking for dg methods on dynamically adaptive meshes.," *SIAM Journal on Scientific Computing (SISC).*, vol. 42, no. 3, pp. C69–C96, 2020.

[4] P. Samfass, J. Klinkenberg, M. T. Chung, and M. Bader, "Predictive, reactive and replication-based load balancing of tasks in chameleon and sam(oa)2," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ACM, 2021.

[5] J.-L. Fattebert, D. Richards, and J. Glosli, "Dynamic load balancing algorithm for molecular dynamics based on Voronoi cells domain decompositions," *Computer Physics Communications*, vol. 183, no. 12, pp. 2608–2615, 2012.

[6] A. Narang, A. Srivastava, R. Jain, and R. K. Shyamasundar, "Dynamic distributed scheduling algorithm for state space search," in *Euro-Par 2012 Parallel Processing* (C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, eds.), (Berlin, Heidelberg), pp. 141–154, Springer Berlin Heidelberg, 2012.

[7] 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 Middlewear (ESPM2)*, pp. 32–39, 2016.

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.

[8] B. J. N. Wylie, "Exascale potholes for HPC: Execution performance and variability analysis of the flagship application code HemeLB," in *2020 IEEE/ACM Int. Workshop on HPC User Support Tools (HUST) and Workshop on Progr. & Perf. Vis. Tools (ProTools)*, pp. 59–70, 2020.

[9] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "Run-to-run variability on Xeon Phi based Cray XC systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[10] D. Bonachea and A. Kamil, "UPC++ v1. 0 Specification, Revision 2022.3. 0," tech. rep., Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2022.

[11] C. Hewitt, "Actor Model for Discretionary, Adaptive Concurrency," *CoRR*, vol. abs/1008.1459, 2010.

[12] J. Armstrong, "Concurrency Oriented Programming in Erlang," *Invited talk, FFG*, 2003.

[13] D. Charousset, R. Hiesgen, and T. C. Schmidt, "CAF - the C++ Actor Framework for Scalable and Resource-Efficient Applications," in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, (New York, NY, USA), p. 15–28, Association for Computing Machinery, 2014.

[14] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, "Orleans: Distributed Virtual Actors for Programmability and Scalability," *MSR-TR-2014–41*, 2014.

[15] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, (New York, NY, USA), pp. 91–108, Association for Computing Machinery, 1993.

[16] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279–301, 1989.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[18] A. Kulkarni and A. Lumsdaine, "A comparative study of asynchronous many-tasking runtimes: Cilk, Charm++, ParalleX and AM++," 2019.

[19] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.

[20] G. Zheng, E. Meneses, A. Bhatelé, and L. V. Kalé, "Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers," in *2010 39th International Conference on Parallel Processing Workshops*, pp. 436–444, 2010.

[21] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 647–658, 2014.

[22] H. Menon and L. Kalé, "A Distributed Dynamic Load Balancer for Iterative Applications," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2013.

[23] V. Freitas, A. de L. Santana, M. Castro, and L. L. Pilla, "A Batch Task Migration Approach for Decentralized Global Rescheduling," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 49–56, 2018.

[24] V. Freitas, L. L. Pilla, A. d. L. Santana, M. Castro, and J. Cohen, "PackStealLB: A scalable distributed load balancer based on work stealing and workload discretization," *Journal of Parallel and Distributed Computing*, vol. 150, pp. 34–45, 2021.

[25] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. S. Müller, "CHAMELEON: Reactive load balancing for hybrid MPI+OpenMP Task-Parallel applications," *Journal of Parallel and Distributed Computing*, vol. 138, pp. 55–64, 2020.

[26] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. Saraswat, and M. Takeuchi, "GLB: Lifeline-based global load balancing library in X10," in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, (New York, NY, USA), p. 31–40, Association for Computing Machinery, 2014.

[27] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *OpenMP in a New Era of Parallelism* (R. Eigenmann and B. R. de Supinski, eds.), (Berlin, Heidelberg), pp. 100–110, Springer Berlin Heidelberg, 2008.

[28] P. Samfass, J. Klinkenberg, and M. Bader, "Hybrid MPI+OpenMP Reactive Work Stealing in Distributed Memory in the PDE Framework sam (oa)ˆ2," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 337–347, IEEE, 2018.

[29] S. Convent, *Investigating Scheduling and Load Balancing Characteristics of Task-based Programming Models for Hybrid HPC Applications*. PhD thesis, Universitätsbibliothek der RWTH Aachen, 2019.

[30] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), p. 201–212, Association for Computing Machinery, 2011.

[31] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), p. 519–538, Association for Computing Machinery, 2005.

[32] 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*, PPoPP '14, (New York, NY, USA), p. 53–66, Association for Computing Machinery, 2014.

[33] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–11, 2009.

[34] V. Kumar, K. Murthy, V. Sarkar, and Y. Zheng, "Optimized distributed work-stealing," in *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, pp. 74–77, IEEE, 2016.

[35] A. Humphrey and M. Berzins, "An Evaluation of An Asynchronous Task Based Dataflow Approach For Uintah," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, pp. 652–657, 2019.

[36] A. Pöppl, S. Baden, and M. Bader, "A UPC++ Actor Library and Its Evaluation On a Shallow Water Proxy Application," in *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, pp. 11–24, 2019.

[37] D. Bonachea and P. H. Hargrove, "GASNet-EX: A High-Performance, Portable Communication Library for Exascale," in *Languages and Compilers for Parallel Computing* (M. Hall and H. Sundar, eds.), (Cham), pp. 138–158, Springer International Publishing, 2019.

[38] 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*, X10 2016, (New York, NY, USA), p. 24–29, Association for Computing Machinery, 2016.

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

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.

Artifact Description

The operating system installed on the CoolMUC-2 cluster is SUSE Linux Enterprise Server 15 SP1.

Used modules to compile both UPC++ and Actor-UPCXX were:

```
1) admin/1.0
2) tempdir/1.0
3) lrz/1.0
4) spack/22.2.1
5) intel-oneapi-compilers/2021.4.0
7) hwloc/2.6.0-gcc11
8) openmpi/4.1.2-intel21
9) metis/5.1.0-intel21-i64-r64
10) cmake/3.21.4
```

UPC++ was configured with:

```
./configure \
--with-cc=mpicc \
--with-cxx=mpicxx \
--with-mpi-cc=mpicc \
--with-mpi-cxx=mpicxx \
--with-default-network=ibv \
--enable-ibv \
```

Parallelization in Pond was achieved by employing one UPC++ rank per physical core, therefore the sequential back-end of UPC++ was used. Environment variables used to configure UPC++ during the compilation and running Pond were as follows:

```
export UPCXX_CODEMODE="opt"
export UPCXX_THREADMODE="seq"
```

Pond was configured with the following CMake options. Any variable that is marked (a—b—...) that was changed depending on the desired configuration is described later. The CMake command template looks as follows:

```
cmake \
-DCMAKE_C_COMPILER=mpicc \
-DCMAKE_CXX_COMPILER=mpicxx \
-DCMAKE_EXPORT_COMPILE_COMMANDS=ON \
-DENABLE_FILE_OUTPUT=OFF \
-DBUILD_RELEASE=ON \
-DENABLE_O3_UPCXX_BACKEND=ON \
-DENABLE_MEMORY_SANITATION=OFF \
-DIS_CROSS_COMPILING=OFF \
-DINVASION=OFF \
-DTIME=OFF \
-DTRACE=OFF \
-DMIGRATION=(0|2|3) \
-DREPORT_MAIN_ACTIONS=OFF \
-DTHREAD_SANITIZER=OFF \
-DLAZY_ACTIVATION=(ON|OFF) \
-DANALYZE=OFF \
-DINTERRUPT=OFF \
-DSTEAL_ONLY_ACTABLE_ACTOR=OFF \
-DSTEAL_FROM_BUSY_RANK=(ON|OFF) \
-DGLOBAL_MIGRATION=(ON|OFF) \
-DCMAKE_BUILD_TYPE=RelWithDebInfo \
-DMORE_LOCAL_VICTIM_CHOICE=ON \
..
```

Pond supports the following migration types:

- 0 = No Migration
- 1 = Global Migration (deprecated, not used in this paper)
- 2 = Asynchronous Stealing (Actor Stealing)
- 3 = Asynchronous Offloading (Actor Offloading)

The options for the stealing strategy can be set with GLOBAL_MIGRATION and STEAL_FROM_BUSY_RANK. The option GLOBAL_MIGRATION (shortened as GLOBAL) and STEAL_FROM_BUSY_RANK (shortened as BUSY) can be combined together to create the stealing strategies used throughout the paper.

Configuration options:

GLOBAL / BUSY	ON	OFF
ON	global-busy	local-busy
OFF	global-random	local-random

LAZY_ACTIVATION was enabled (=ON) for the lazy activation scenario and disabled (=OFF) for the static workload and node slowdown scenarios.

The skeleton of the Slurm script used to submit and run Pond looks as follows:

```
#!/bin/bash
...

module load slurm_setup
module unload intel-mpi
module load hwloc
module load metis
module load openmpi/4.1.2-intel21

export GASNET_PHYSMEM_MAX='40 GB'
export GASNET_BACKTRACE=1

export GOING_AWAY_LIMIT=1
export STEAL_COOLDOWN=(0|1)
export RMA_TASKCOUNT=1
export USE_TIME_SPENT=1
export SAMPLE_OUTPUT=1

export SLOWDOWN=(0|1)
export SLOWDOWN_RANK_BEGIN=0
export SLOWDOWN_RANK_END=$((4*nodecount))
export SLOWDOWN_TIME_BEGIN=1
export SLOWDOWN_TIME_END=6

${UPCXX_DIR}/bin/upcxx-run -N ${nodecount}\
-n $(( corecountpernode * nodecount ))\
-shared-heap 128MB \
```


Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.

```
${HOME}/Actor-UPCXX/./pond-${job} \
-x ${size} -y ${size} \
-p ${patchsize} \
-c 10 --scenario 3 \
-o ${SCRATCH}/${workdir}/${str}/out/out \
-e ${endtime}
```

In the following section, the used environment variables that influence the execution of Pond are explained.

GASNet is the communication backend of UPC++. GASNET_PHYSMEM_MAX has to be specified to avert the costly test of finding the PHYSMEM_MAX in each run. GASNET_BACKTRACE, when set to 1, prints the stack trace in case the program unexpectedly terminates.

GOING_AWAY_LIMIT and STEAL_COOLDOWN influence the behavior of the stealing strategies. GOING_AWAY_LIMIT=g can be any positive integer. It provides an upper limit for the number of actors that can be marked on a single rank to g. If g is positive and any request that would increase g to g+1 is rejected. If GOING_AWAY_LIMIT is unset or set negative, then there is no limit to concurrent marked actor count. STEAL_COOLDOWN adds a short timeout between steal attempts so that the runtime is not overflooded with steal requests. SAMPLE_OUTPUT enables the printing of some statistics in regular intervals. In every run that had stealing enabled (MIGRATION=2), the following environment variables were set: GOING_AWAY_LIMIT=1, RMA_TASKCOUNT=1. Every run (regardless of the migration type used) had the following environment variable set: SAMPLE_OUTPUT=1. USE_TIME_SPENT=1 was employed to enable time counting in the static workload and lazy activation scenarios, whereas it was set to USE_TIME_SPENT=0 to enable task counting in the node slowdown scenario.

If RMA_TASKCOUNT is enabled, then workload exchange is performed through RMA operations, even though the name is RMA_TASKCOUNT, it supports RMA for both time counting and task counting.

Environment variables that have the prefix SLOWDOWN_* are only related to the node slowdown scenario. If SLOWDOWN is set to 0 then the remaining SLOWDOWN_* environment variables are ignored. If set to 1 then SLOWDOWN_RANK_BEGIN=a describes the lowest rank id that is affected by the node slowdown, and SLOWDOWN_RANK_END=b describes the lowest rank id that is not affected by the node slowdown. Thus, ranks with ids within [a;b) are affected by the slowdown. SLOWDOWN_TIME_BEGIN=t1 describes the start of the slowdown interval in minutes. Slowdown starts after t1*60 seconds after the computation begins, SLOWDOWN_TIME_END=t2 describes the end of the slowdown interval in minutes. Slowdown ends at the (t2*60)th second after the computation has started. Every run of node slowdown had the slowdown-related variables set to SLOWDOWN=1, SLOWDOWN_RANK_BEGIN=0, SLOWDOWN_RANK_END=$((4 * nodecount)), SLOWDOWN_TIME_BEGIN=1, SLOWDOWN_TIME_END=6. The node slowdown scenario increases the execution time of actors by a factor of 3, this is a hardcoded *constexpr* value.

UPCXX_DIR is a location set by the user for the UPC++ library and the upcxx-run job launching script. *nodecount* describes the number of nodes UPC++ will run on, which is used by the *-N* argument of upcxx-run. *Corecountpernode* describes the amount of available physical cores per compute node. For the CoolMUC-2 cluster this number is 28. The amount of ranks is equal to the *nodecount* times *corecountpernode*, and is evaluated with $((*corecountpernode* * *nodecount*)), which is then used by the *-n* argument that describes the total number of processes. The *job* environment variable is the suffix of the Pond executable. This is used to run the desired Pond configuration. These names are user-defined. *Size*=N describes one dimension of the $N \times N$ grid. In our runs, Pond was run on a square grid, therefore the same environment variable *size* was used for both x and y dimensions. *Patchsize*=P describes the size of one dimension of the actor $P \times P$ patch, which is used by the *-p* option for Pond. Even though no output of the simulated domain was generated, the argument *-o* for the output name has to be provided. The *endtime* environment variable describes the simulation time, which is used by the option *-e*. The simulation terminates as soon as every actor reaches the *endtime*. The used *pond-scenario* was the "scalable multi-drop", which has the scenario id *3* and is set with –*scenario=3*.

For the complete data set (e.g., raw output files, input reader scripts and build scripts), see https://doi.org/10.5281/zenodo.7133771.

Authorized licensed use limited to: Technische Universitaet Muenchen. Downloaded on July 11,2023 at 12:29:04 UTC from IEEE Xplore. Restrictions apply.