Dynamic Actor Migration for a Distributed Actor Library

Yakup Koray Budanaz
Dynamic Actor Migration for a Distributed Actor Library

Dynamische Aktormigration für eine verteilte Aktorbibliothek

Author: Yakup Koray Budanaz
Supervisor: Prof. Dr. Michael Georg Bader
Advisor: M.Sc. Alexander Pöppl
Submission Date: 15.08.2020
I confirm that this bachelor’s thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.08.2020

Yakup Koray Budanaz
Acknowledgments

I want to thank my advisor, Alexander Pöppl, for his helpful insights and for reading my cryptic emails during the thesis. I also want to thank Moritz Milla for assisting me in debugging, and Kerem Cakirer for listening to my rants on template errors, as well as helping me during the proofreading.
Abstract

With hardware architectures and software becoming ever more complex, applications in high performance computing are required to dynamically adapt to any workload imbalance that can be caused by multiple sources. In order to facilitate dynamic workload-balancing in the UPC++ Actor Library, Actorlib, the requirements for actor migration between nodes are implemented and possible approaches and strategies to migrate actors are proposed. The implemented migration strategies and the choices between various implementations are discussed and their results are analyzed. The analysis shows that optimal partitioning of the actors have a higher impact on the run-time compared to adapting to the finer workload imbalances in the system. Lastly better strategies that do not worsen the partitioning of the system is proposed.
Kurzfassung

Da die Hardwarearchitekturen und die Software immer komplexer werden, müssen die Anwendungen im Bereich von Hochleistungsrechnen sich dynamisch an das Ungleichgewicht der Arbeitslast, die von verschiedenen Ursachen erzeugt werden kann, anpassen. Um die dynamische Ausgleichung der Arbeitslast für die UPC ++ Akteurbibliothek, Actorlib, zu ermöglichen, werden die Voraussetzungen, die für die Migration von Akteuren zwischen Knoten nötig sind, implementiert und mögliche Ansätze sowie Strategien zur Migration von Akteuren vorgeschlagen. Die implementierten Migrationsstrategien und die Wahl zwischen verschiedenen Implementierungen werden diskutiert und ihre Ergebnisse analysiert. Die Analyse zeigt, dass eine optimale Aufteilung der Akteure einen höheren Einfluss auf die Laufzeit als das feineres Balancieren der Arbeitslast im System hat. Schließlich werden bessere Strategien vorgeschlagen, die die Partitionierung des Systems nicht verschlechtern.
## Contents

**B. Reproducing the Examples: Scripts**  
60

**C. Work Complexity**  
70

**D. Runtime 3D Coordinates**  
71

List of Figures  
73

List of Tables  
75

Bibliography  
76
1. Introduction

With complex and heterogeneous hardware architectures as well as temporary workload imbalances caused by the software, for example in simulations with adaptive mesh refinement, distributing the workload evenly on each computing node may result in sub-optimal run times. Due to the impossibility of statically predicting the workload imbalances, the applications have to dynamically adapt to it.

In this thesis, possible approaches that can extend the UPC++ Actor library with dynamic workload balancing are discussed. Proposed approaches include the global (random) ordering approach that re-partitions the actors periodically and the actor steal approach where under-utilized nodes steal actors from overloaded nodes. It is based on the work-stealing [10] approach and can balance the workload without interrupting the computation. Both approaches can be improved to regard the partitioning of the actors, with the introduction of territories, thus dodging unnecessary communication overhead caused by sub-optimal partitioning of the actors. The advantages and disadvantages of the approaches over one another, and the reasons behind the implemented components’ performance are discussed.

The results of the naive implementation for the global random reordering without territories show that the performance loss caused due to the communication overhead induced by random partitioning outweighs the performance gain due to better workload balancing. But still, the global reordering approach, if the overhead due to partitioning is ignored, will have acceptable and even better performance on coarse imbalances. Although not implemented as a part of the thesis, a composite approach that uses global ordering on coarse imbalances with an accompanying actor steal approach on fine workload imbalances while regarding the old partitions may improve run-times and result in better performance compared to the approaches without dynamic load balancing.

The rest of the Introduction chapter will go over the actor model, partitioned global address space model and the UPC++ library. Followed by an introduction to Actorlib and to the provided extensions to facilitate actor migration in the second chapter. After the introduction is completed, the approaches that can be used to migrate actors are discussed in third chapter under Possible Approaches, followed by explanation of the implemented components in the Implementation chapter. The implemented component, tests and their results are discussed in the Results chapter, followed by the Conclusion. Finally the possible works in future and improvements for the Actorlib library are discussed. The script generators, build configurations and additional visualizations for results can be found in the Appendix.
1. Introduction

1.1. Actor Model

An Actor is considered the universal primitive in Actor Model. When an actor receives a message it can send messages to other actors, create new actors or change its state to handle an incoming message [22] [31]. The communication is of finite length and has to be buffered. The communication is asynchronous and non-blocking. Actors do not share states, and can not directly cause side effects on other actors. Discrete states and asynchronous messaging qualify the actor model for parallel computing, as it hinders common causes for deadlocks and data races. High parallelity of the actor model also qualifies it for distributed as well as heterogeneous environments.

There are various industry oriented implementations of the actor model already in use in distributed environments and such as Erlang [4], the Akka [1] framework for Scala, the C++ Actor Framework (CAF) [18] [19] and Orleans [9] framework for .Net. Rinaldi et al. [44] use CAF and show that actor based implementations can keep up with the more specialized libraries while preserving the higher layer of abstraction. Scalable Distributed Erlang [21] [20], an extension for Erlang aimed for scalability in HPC applications has also been in use: for example in urban traffic simulations [49] [50].

1.2. (Asynchronous) Partitioned Global Adress State

The PGAS [23] programming model provides a globally shared and a local address space. Global address space allows the user to regard the global memory as if it was shared, providing higher lever of abstraction compared to MPI since no explicit send/receive calls are necessary and the memory can be accessed directly. The asynchronous PGAS model (APGAS) [15] extends the above mentioned PGAS with asynchronicity through asyncs, where statements are executed non-blockingly in the units that operate on data, so called places.

Languages like Coarray Fortran [40], UPC [26] use the PGAS model, whereas the newer approaches like X10 [17], UPC++ [6] and Chapel [16] embrace the asynchronous approach.

The MPI [27] is the de-facto standard for distributed memory computing, as GASNet-Ex [12] is for APGAS programming. GASNet is a low-level language-independent communication library. With hardware specific implementations, also called conduits, GASNet provides close to metal speed competitive with MPI [13]. Many (A)PGAS languages including UPC++ use the GASNet(-Ex) API as their communication backend.
1. Introduction

1.3. UPC++

Following the APGAS model, Unified Parallel C++ (UPC++) is an C++ library that relies on GASNet-Ex as its communication back-end. UPC++ allows access to remote memory via one-sided remote memory access communication operation: RMA; allowing the transfer of non-contiguous, structured and strided data. Another core feature of the UPC++ is the remote procedure call: RPC, which allows invocation of functions on remote places, named , ranks. Due to aggressive use of template meta-programming and lambda expressions, it does not require a compiler extension.

Being a library on top of C++ has an advantage on the availability of libraries. Commonly used libraries, such as EIGEN (C++) [28], LAPACK(C) (Fortran/C) [2] or NETCDF (Fortran,C,Java) [13] support mainly C,C++ and Fortran. Using these libraries for other languages such as (SD)Erlang requires the user to provide bindings (see [46]) for the libraries, which can increase development time.

The most important features of UPC++ is introduced in the following subsections. For further details and reading about the UPC++, see UPC++ Specification [11] and the UPC++ Programmer’s Guide [7].

1.3.1. Global Memory

In UPC++, each rank can allocate global memory in the shared segment and any rank can access the globally allocated memory with the global pointer that points to it. Besides the shared segment each rank has its own local memory.

In the following code snippet each rank allocates one integer in the shared segment and one in the local space.

```cpp
upcxx::global_ptr<int> gptr = //global space
upcxx::new_<int>(upcxx::rank_me());
int* lptr = new int; //local space
```

If a rank needs to work with an object allocated in the shared segment then two cases arise. If the allocated object has affinity to a local process, meaning that it was allocated on that rank, then it can be downcasted to a local pointer with the provided upcxx::local() method. In order to check for affinity UPC++ provides upcxx::is_local() method that returns true if the object has affinity to the calling rank. If the allocated object has affinity to a remote process, then the value or a copy has to retrieved with a remote memory access operation.
1.3.2. Remote Memory Access

If the object is *TriviallySerializable* then the value of the object can be retrieved with `upcxx::rget` or the value on the remote rank can be set with `upcxx::rput`. Similar to many asynchronous functions, they do not return/set the value instantly but return an `upcxx::future`, the result is then waited for with a call to `upcxx::future::wait()` method. An object is TriviallySerializable when it fulfills `std::is_trivially_copyable` meaning that the object (or all its members) can be copied directly byte to byte, for example with `std::memcpy`. This will not be case with most of the user defined classes, but any action that requires changing/copying of a remote object can be achieved with reconstructing the class from the serialized state/data in a remote procedure call on a remote rank.

1.3.3. Remote/Local Procedure Call

A remote procedure call allows execution of a function on a remote rank. It takes the recipient rank’s index where the function should be executed, a function, the function’s arguments and returns a future. Although other overloads and callbacks other then futures exist, every overload require the recipient rank’s index, a function as well as its arguments.

```cpp
template < typename Func , typename ... Args >
future_rpc_result_t < Func , Args ... >
rpc ( intrank_t recipient , Func & func , Args &&... args );
```

The future has the type `upcxx::future<>` if the function `Func` returns `void`. Otherwise it will be `upcxx::future<deserialized_type_t<T>>` if the function `Func` returns an object.
of type T. UPC++ does not enforce the to-be-serialized type and its deserialized type to be the same, hence the deserialized_type_t<T> as the return value. In order to be able to use complex classes in a RPC, a serialization method has to be provided.

A local procedure call (LPC) can be considered, in a simplified way, as a RPC that executes the function Func on the same rank. It is a method of upcxx::persona, and is executed during progress, where UPC++ run-time processes RPCs and LPCs. Signature is similar to the signature of a RPC, but without the rank index.

\[
\text{persona :: lpc ( Func \&\& func, Arg \&\&... args );}
\]

1.3.4. Progress and Persona

UPC++ runtime advances the states of its asynchronous operations only during the calls that initiate progress, meaning that the RPCs, LPCs and their callbacks are evaluated only when user grants UPC++ access to CPU cycles. Granting access is done through calls to UPC++ functions, which induce progress. It can be done through calls to upcxx::progress, waiting for futures via upcxx::future::wait or imposing a barrier with upcxx::barrier. The barrier special case also requires all ranks to reach that execution point.

Personas simplify the handling of the communication between threads and UPC++’s internal state. A persona is a collection of UPC++’s internal-state. With a call that initiates progress, each thread will advance the state of its personas.

1.3.5. Serialization

Any class that is not TriviallySerializable needs to provide a serialization method if it is to be used in a rpc or to be sent to another rank. UPC++ allows Field-Based serialization and custom serialization which can be achieved by defining the struct upcxx_serialization, or providing a specialization of upcxx::serialization<T>.

In Field-Based serialization user defines the fields to be serialized with UPCXX_SERIALIZED_ FIELDS(T...) directives. In case when a more complex serialization method is required the user can define the upcxx_serialization with static void serialize (Writer& writer, T const& object) and static U* deserialize(Reader& reader, void* storage) method that uses the UPC++ provided Writer and Reader classes to write/read data to/from a byte stream.

Same behavior can be achieved with defining the serialize and the deserialize methods in a template specification of the upcxx::serialization<T>. Specialization is necessary for providing serialization methods for classes regardless of where they are defined, indispensable when a class defined in the standard library needs to be serialized.
2. Actorlib

2.1. Actorlib

Actorlib [11] is an actor based high performance computing library written on top of UPC++ [6]. It provides an interface Actor for actor implementations, and the ActorGraph that handles the execution.

An actor in the library is described by the the following tuple: \((ID, r, I, O, F, R)\) with the name ID, placement (rank) r, set of InPorts I, set of OutPorts O, set of functions F, the finite state machine R. The communication between Actors is one-sided and performed asynchronously between Actors through ports. The ports (one OutPort and one InPort) that participate in the communication are connected to a single Channel where OutPort can write data and the InPort can read data from the Channel.

In order to send a message, the Actor’s OutPort writes a message into the Channel. The buffered message can be read from the InPort of the recipient Actor. In the implementation all three components require template arguments for the fixed token with type T, and the integer capacity c.

Each Actor starts with one trigger, meaning that it can call its act() method only once. Upon sending or receiving a message the trigger count is increased, ensuring that the Actors are able to handle any incoming message. The Actor works until it has no triggers left or it final state in R is reached.

As pictured below a message is sent from Actor A1 to Actor A2 in a write() call where the message is sent to the Channel on the remote rank with an RPC (1). The completion of the RPC is tracked with an LPC (3). Upon receiving the message the recipient Actor’s trigger count is increased (2).

---

1If the word refers to a component/class implemented in the Actorlib or in Actorlib-Extended the same name that appears in the library is used to simplify finding to corresponding code/implementation
2. Actorlib

If there are any buffered messages and the Actor A2 calls read on its Channel (1), then oldest message is dequeued from the Channel (1). Due to the read, the trigger of the sender Actor A1 is increased with an RPC (2). Since the actor A2 can not change the trigger count of the A1 directly, the actor changes the trigger count with an LPC after receiving the message (3). Similar to the write call the RPC is tracked (4).

In order to use Actorlib, the user has to implement specifications for the abstract Actor class. The only requirement imposed on the user is the definition of the act() method. The implemented Actors has to be inserted into the ActorGraph with ActorGraph::add that takes Actor pointers as input and the ports of the Actors have to be connected with connectPorts calls, requiring Global Actor Reference of both Actors and the names of the ports to be connected. Global Actor Reference (GlobalActorRef) is a user defined synonym for upcxx::global_ptr<Actor *>. It is the global pointer (on the global address space) to a pointer to an actor.

2.2. Actorlib-Extended

In order the facilitate the actor migration several additions to Actor and to ActorGraph have been introduced.

An Actor in Actorlib-Extended is described by the the following tuple: \((ID, r, I, O, F, R, S, W)\) additional to the Actor from the Actorlib the Actor has states \(s \in S\) to make migration process...
possible. In order to track work a set of tracking variables $w \in W$ is needed.

![Finite State Diagram](image.png)

Figure 2.3.: The finite state diagram describing the transition of an Actor to a state where it is allowed to call `act()`.

A newly created Actor is in the (new) initial state ($q_0$), where it can be connected to the ActorGraph ($q_1$). If an Actor is disconnected it can be reconnected to the graph ($q_4$). A disconnected/not connected actor can send its copies to remote ranks or create copies ($q_1, q_4$). Connected Actors can be started and stopped ($q_3, q_6$).

<table>
<thead>
<tr>
<th>isConnected</th>
<th>isRunning</th>
<th>needsActivation</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$q_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$q_4$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>illegal descriptor</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>illegal descriptor</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$q_2, q_5$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$q_4$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$f$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>illegal descriptor</td>
</tr>
</tbody>
</table>

Figure 2.4.: The boolean tuple used in the Actor implementation to track the state of an Actor during the migration procedure described in 2.3.

A disconnected Actor $A$ in $q_4$ needs to be tracked, whereas a not-connected Actor in $q_1$ does not need to be tracked. It is due to already existing connections of the disconnected Actor $A$, restarting the ActorGraph will cause problems with the open end of the neighbors’ OutPorts connected to $A$, therefore such Actors have to be tracked and always restarted.

In order to reconnect an Actor to its old neighbors, the information on the connections
between In- and OutPorts needs to be preserved. Actorlib-Extended provides \( \text{PortGraph} = P(X, (X, X)), X = (V, I) \cup (V, O) \) alongside the ActorGraph \( G = (V, E) \), in order to track connections without requiring remote procedure calls.

### 2.3. Serializing Actors

For the migration, the Actor has to be removed from the ActorGraph, and its state has to be serialized into a raw byte stream. In order to serialize complex objects, access to run time type information is beneficial and under specific conditions necessary, because every class has to have the same type before and after the serialization. Derived classes for example require run time type information for the correct serialization regarding their dynamic type, thus a method to revert the type erasure is required. Actorlib-Extended uses template meta programming techniques with the help of \text{type_traits} library and the \text{std::variant} to deduce the right type of an Actor.

Instead of only supporting adding actors to the ActorGraph, the extension allows removing of actors as well. Removal of Actors is required, since migrating requires sending a copy to a remote rank and deleting the obsolete actor on the old rank.
3. Possible Approaches

Actor migration shares similarities with process migration [39]. In order to migrate an Actor, its state has to be extracted, serialized and transferred to its destination; later a copy has to be reconstructed from the transferred state, restarted and the connections have to be restored. The steps are similar to the steps required for migrating a process.

The actors can migrated on interrupts and a specific metric can be used to calculate migrations, a similar approach is applied in CoCheck [47], where processes are migrated on checkpoints. Although the high-level idea is similar, the implementation specific details fundamentally differ due to the differences between processes and Actors.

The implemented migration approach can be considered the naive approach of migrating, where the graph of actors is re-partitioned during an interrupt. In order to avoid the dead-time caused by globally interrupting every rank, the ranks that are under-utilized can steal actors from victim ranks, again according to a specific metric, without blocking the ActorGraph. Both approaches can be improved to prevent migrations from creating random partitions.

3.1. Common Requirements for all Approaches

Regardless of the used strategy or methods, each approach needs to perform a set of actions in order to be able to migrate Actors between ranks. The Actor has to be stopped, and all neighbors prevented from sending messages to the stopped actor. No neighbor may write to stopped Actor’s Channels or read from its Channels, since it will change the state of the OutPort (which should not change during migration). After it is made sure that the Actor’s message buffers on Ports will not change their states, the Actor can be prepared for the migration. It is possible that not every Actor will need to prepare for migration. Preparation is needed for example when the Actor writes to a file, then the file has to be closed and the ownership has to be released.

After the Actor is disconnected from the ActorGraph and prepared, the copy can be sent to another rank. The lingering original actor can be deleted any time after the copied actor is sent and reconstructed. After the reconstruction the Actor needs to be reconnected and the buffers of the Channels has to be refilled if any buffered messages were sent along with the actor. Actor can be restarted after the connections to all other neighbors are restored. The
process can be visualized with the following dependency graph:

![Dependency graph for actor migration](image)

Figure 3.1.: Dependency graph for actor migration. Before an action \( v \) can be performed all actions \( \forall v' : (v, v') \) must be performed. Deletion of the lingering Actors is omitted.

The Graph \( G := (V, E) \) with \( V \) the set of actions, consist of a tuple of statements that perform the high-level description, and \( E \) the set of edges with \( e := (v, v') \in V \) describing the dependency between actions where \( v \) requires the completion of \( v' \) before it can be executed.

### 3.2. Global (Random) Ordering

Global Ordering is implemented on top of ActorGraph and interrupts it after each timeout to migrate Actors between ranks. With a given user-defined or default metric, it chooses the Actors to migrate. The default metric implemented does not try guess the serialization time or the remaining time of the job. It also does not keep track of the number of partitions in the graph of actors, and may cause arbitrary partitioning of the graph, hence the name global (random) ordering. Global (random) ordering can be improved with additional checks to prevent arbitrary partitioning or to keep the number of messages sent to remote ranks minimal. Such a proposal is discussed below in Territorial Expanse!

Global (Random) Ordering performs the following tasks:

- Interrupt computation
- Calculate migrations
- Migrate actors
- Restart computation
3. Possible Approaches

The actor graph runs under the control of the `DynamicActorGraph` where ActorGraph returns after exceeding the timeout interval. A metric is used to choose the Actors for migration and their destination ranks. Afterwards the chosen Actors are then sent to their destinations. Global (random) ordering approach sends the actors in bulks, only waiting at the end of the steps described in the dependency graph for actor migration.

```c++
//bulk sending
prepare(a,b,c);

//separate sending
prepare(a,b,c);
send(a).wait();
send(b).wait();
send(c).wait();
```

Figure 3.2.: Bulk sending vs Separate sending. Functions are simplified to contrast the differences

When sending in bulks, the rank can start with the serialization of `b` instead of waiting for the arrival of the actor `a` at the remote destination. Bulk sending will yield faster delivery time compared to separate sending, mainly due to dead-time in separate `future::wait()` calls being utilized to serialize multiple Actors, postponing the wait call. Bulk sending should be preferred over separate sending. The details of the implementation is explained in the Implementation chapter.

In order to prevent global (random) ordering from creating arbitrary partitions it can be improved with a component that calculates migrations that keep the messages sent to remote ranks and number new partitions minimal.

3.3. Territorial Expanse!

Migrating actors between ranks allows more homogeneous distribution of the work between ranks but the neighbors of the migrated Actors have to send the messages to the new ranks of the migrated actors. The speed of the delivery of messages is crucial for fast execution of the job and sending messages to separate ranks induce additional overhead compared to the messaging within the same rank. If the other rank is on another node then the overhead increases.

If the message travels within the same rank until its last recipient, then there exists a `Message Chain`. The problem of finding longest message chains can be reduced to a balanced
3. Possible Approaches

graph partitioning problem. Effects of breaking a chain is exemplified in the Message Chain section under Results.

3.3.1. Message Chain and Borders

Migrating a random Actor of rank \( n \) to another rank \( m \) can result in an island in a sea of rank \( n \) actors; thus breaking the message chain, increasing the communication overhead. In the following example migrating the actor in the center from rank \( n \) to rank \( m \) breaks the message chain, therefore 2 messages have to be sent to and received from rank \( m \) in each iteration.

\[
\begin{array}{c}
\text{n} \\
\downarrow \\
\text{n} \\
\downarrow \\
\text{n} \\
\downarrow \\
\text{n} \\
\downarrow \\
\text{n} \\
\end{array} 
\Rightarrow 
\begin{array}{c}
\text{n} \\
\downarrow \\
\text{n} \\
\downarrow \\
\text{n} \\
\downarrow \\
\text{n} \\
\downarrow \\
\text{n} \\
\end{array}
\]

Figure 3.3.: A sea of rank \( n \) actors break when the actor in the middle is migrated to the rank \( m \), creating an island consisting of one \( m \) actor and breaking the message chains

In order to minimize the communication overhead, a rank should only migrate its actor \( A_i \) to a rank which it directly communicates with. Such an actor can be considered as a **border Actor**.

\[
\begin{array}{c}
A0 \\
\downarrow \\
A1 \\
\downarrow \\
A2 \\
\downarrow \\
A3 \\
\downarrow \\
A4 \\
\end{array} 
\]

Figure 3.4.: \( A2 \) communicates with an actor from rank \( m \) and from rank \( l \), therefore it can be migrated to rank \( m \) or \( l \) only

The actor \( A2 \) is a **border actor** that can send messages to \( A3 \) from rank \( m \) and to \( A4 \) from rank \( l \). To keep the number of messages that need to be sent to another rank minimal, the actor \( A2 \) is only allowed to migrate to the rank \( m \) or \( l \) and to no other rank.
3. Possible Approaches

3.3.2. Partitions

The actor graph is directed Graph $G = (V, E)$ with $V$ the set of nodes, and the set of connections between actors $E$ as directed edges between Actors such that: $e \in E : (v, v') = v$ sends message to $v'$.

In order to achieve maximal speed of communication, one has to assure that a message sent from rank $n$ to another rank shall not return back to rank $n$ along its path. This constraint can be satisfied by partitioning the actor graph in $k$ many partitions with each partition roughly containing $|V|/k$ many nodes while keeping the number of edges between components minimal. This problem is known as the $(k, v)$-balanced graph partitioning problem\cite{3}, and is NP complete. Due to its importance, there exists efficient approximation algorithms. The METIS \cite{35} library, also used in the pond application to distribute the actors on ranks, can be used to calculate an approximation for the $(k, v)$-balanced graph partitioning problem, providing an initial distribution of the actors.

3.3.3. Territory

Territorial Expanse aims to distribute work more homogeneously while trying to keep the number of messages that need to be sent to a remote rank minimal (as few new partitions as possible), therefore only the border actors can be migrated between ranks.

In SWE simulations the movement of water bodies over a domain is calculated. The domain of computation can be analogized to a map where each Actor (the case with the SimulationActor in pond) calculates the volume difference on the sub-area of the domain. The Actors of the $k$th rank can be considered as the $k$th rank’s territory, where each rank tries to maximize their workload by trading Actors on their borders with their neighbors. This approach again requires a definition of a metric to calculate the resources available to claim Actors from borders.

After the appropriate distribution of currency, similar to the Market approach, each rank that has more than the mean amount of currency, may request actors from its neighboring ranks that have less then mean amount of currency. An appropriate actor can be sought in a process where a rank with higher than the mean currency iterates over its neighbor ranks and bids all of its currency for the acquisition of an Actor. If there is an Actor that has less value than the difference of currencies then the Actor is acquired and the value is decreased from
acquirer’s currency:

Data: currency of rank n (self)
return vector of claimed actors
claimedActors
for rank_m ∈ neighboring ranks do
  if ∃ actor : currency(m) − currency(n) ≥ value(actor) then
    claimedActors ← actor
    currency(m) ← currency(m) − value(actor)
  end
end
return claimingActors

Algorithm 1: Requesting Actors from neighbor ranks

With the under-utilized rank requesting Actors from overworking neighbors, the Territorial Expanse can be viewed as an adaptation of the Receiver Initiated Diffusion [51] approach.

3.3.4. Use with Changing Number of Ranks

The acquisition of bordering Actors allows migration of Actors while keeping the increase of the communication overhead minimal, but the approach fails if a new rank is introduced to the system. Passing a part of each rank’s territory to the newcomer rank will increase the communication overhead due to the messages that constantly need to travel from and to rank n + 1, the distribution of the Actors with the new number of ranks is required.

An approach without calculating a re-distribution for the ranks, is plausible first after the introduction of n new ranks, where the number of ranks before the introduction of the new ranks is also equal to n. In the case of an rank increase from n to 2n ranks, half of each rank’s territory can be assigned to one of the new ranks. Since all ranks need to send multiple Actors, it will benefit from bulk sending. With enough Actors migrated, calculating a new partition and redistributing the Actors may have less communication overhead and compensate for the extra time needed for the migration, especially if the job is expected run for a longer time.

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\Rightarrow
\begin{array}{cccc}
1 & 1 & 2 & 2 \\
1 & 1 & 2 & 2 \\
0 & 0 & 3 & 3 \\
0 & 0 & 3 & 3 \\
\end{array}
\]

Figure 3.5.: Example redistribution of the territory when 2 ranks join existing ranks

In real-life scenarios the from-scratch assigning the parts of territories to new ranks will not be simple as the above mentioned example. The re-distribution of the whole graph can be partially avoided and the Actor migration can be limited by partitioning the graph with fixed

3.4. Actor Stealing

*Global Random Reordering* and *Territorial Expanse* requires all of the ranks to participate in the migration process, therefore during the calculation of Actors to be migrated, no rank can progress. The global lock has to be compensated with the better distribution of the workload.

Due to highly serial structure of the actor migration (see the dependency graph), sending Actors in bulks and waiting at the end of the each step will have a better run time compared to separately sending the Actors. Although bulk sending has its advantages, the global lock required by the current implementation will cause periods of dead-time for non-participating ranks and the benefit of the better workload balance will not exceed the time lost due to the waiting ranks. The difference between time gained and lost will increase as the number of ranks increase.

The time lost because of the waiting ranks during the reordering process can be saved by implementing an approach that does not need to block the ActorGraph. Naturally implementing this approach will require more effort than implementing Global Random Ordering or Global Ordering with Territorial Expansion, due to additional synchronization constraints.

Actor stealing can be considered as an adaptation of the *work stealing* [10] approach to the actor model on distributed memory systems. Where idle or under-utilized ranks steal Actors, the atomic unit of computation in the actor model, from victim ranks in order to achieve better work-load distribution.

The idea in *actor stealing* is to migrate Actors from ranks with higher load to ones with lower load without blocking the ActorGraph, with the help of Communicator and Coordinator Actors. Coordinators are required mainly to not the break the actor model abstraction. Communicators are used to find the victim ranks and compare workloads between ranks according to the user defined or the default metric. The concept of Communicators is based on the communication thread implemented in Chameleon [36].

3.4.1. Message Injection

A call to the migrate procedure requires injecting messages to a running ActorGraph. The migrations can be controlled by a component that interacts with to the ActorGraph, or it can be implemented internally as a part of the ActorGraph with a new set of Actors without breaking
the actor model abstraction. The procedure can be described with the following pseudo-code:

\textbf{Data:} ActorGraph = G(V, E), actor to migrate \( m \), rank\(_{id}: i \\
\textbf{for actor } a \textbf{ where } \exists e \in E = (a, m) \textbf{ do} \\
| \textbf{prohibit\_write}(a,m) \\
\textbf{end} \\
\textbf{for actor } a \textbf{ where } \exists e \in E = (m, a) \textbf{ do} \\
| \textbf{prohibit\_read}(a,m) \\
\textbf{end} \\
\textbf{stop}(m) \\
\textbf{disconnect}(m) \\
\textbf{prepare}(m) \\
\textbf{send\_actor}(m,i) \\
\textbf{reconnect\_actor}(m) \\
\textbf{refill\_ports}(m) \\
\textbf{restart\_actor}(m) \\
\textbf{for actor } a \textbf{ where } \exists e \in E = (a, m) \textbf{ do} \\
| \textbf{allow\_write}(a,m) \\
\textbf{end} \\
\textbf{for actor } a \textbf{ where } \exists e \in E = (m, a) \textbf{ do} \\
| \textbf{allow\_read}(a,m) \\
\textbf{end} \\
\textbf{return} \\

Algorithm 2: Migrate: Migrating an actor to rank \( i \) without blocking the actor graph. The wait calls to asynchronous functions are omitted.

UPC++ does not promise in-order delivery of remote procedure calls, therefore after the \texttt{prohibit\_write}(a,m), \texttt{prohibit\_read}(m,a) and \texttt{stop}(m) calls, the system has to assure that no remote or local procedure calls that needs the participation of the to-be-migrated Actor are in the flight. After the messages in flight are processed by the system the Actor can be migrated to its new rank and reconnected to the ActorGraph. The calls for \texttt{prohibit\_write}(a,m), \texttt{prohibit\_read}(m,a) and \texttt{stop}(m) can be sent as messages to other Actors, where the receiver Actor changes its state to the state described/required by the message.

It is possible to change the state of another Actor from another Actor with a function invoked by a \texttt{upcxx::rpc} but this will break the actor model abstraction and should be avoided. For further reading on mixing multiple concurrency models with actor model, see [48].

The actor has no information on where it runs and it cannot migrate itself to another rank, therefore it has to communicate with the system (ActorGraph) and inform the actor graph that it is willing and ready to be migrated, and later receive the green-light for migration; or it can send the information required to copy itself to the Coordinator and it can create a new Actor from the received data, and the Actor will then receive a message from the Coordinator to delete itself and commit suicide, where the Actor calls \texttt{delete this}; on itself. It is only
possible if the actors are dynamically allocated on the heap.

3.4.2. Communicator

Finding appropriate Actors for migration and deciding to steal one from a remote rank requires tracking of the accumulated run-time (or a similar metric) and comparing own value with the values of remote ranks. To calculate and compare these values a Communicator has to be implemented, unique for each rank that performs the following actions:

- receive the values for work done by the rank’s actors
- send the value of work done to other ranks
- receive the work done values from other ranks
- decide to steal an actor

If the rank decides to steal an Actor (or a particular actor from a remote rank) then a call to the Coordinator from the Communicator can fire the migration process.

3.4.3. Coordinator

The migration process fires as the Coordinator receives a message to steal an Actor from the Communicator. The Coordinator then finds a victim rank that can send an Actor and forces it to send an actor. (In a sense actor pillaging is a more appropriate name here). To tasks of the Coordinator is as follows. The approach independent actions, such as sending a message to refill ports or sending a message to allow re-connection are omitted:

- send "prohibit write" messages and receive acknowledgments
- send "prohibit read" messages and receive acknowledgments
- send "stop" messages and receive acknowledgments
- send the data needed to reconstruct an actor
- receive the data needed to reconstruct an actor
- send message to create data for serialization
- receive the data for serialization
- create an actor from the received data
- send the "kill yourself" message

Extra overhead introduced due to the Coordinator and the Communicator can be com-
3. Possible Approaches

pensated by the work done by other ranks during the migration, but it will be faster to send n actors with global ordering compared to n Actors with actor steal; therefore coarse imbalances can be balanced with the global ordering strategy, while finer imbalances are handled with the actor stealing strategy.

Sending n actors during an interrupt is not just because of the less communication overhead faster, but due to Bulk Waiting where the separate future: :wait calls can be postponed until the end of the each step as can be seen in 3.1. The run-time advantage of bulk waiting is discussed in Results chapter, under Bulk Waiting.
4. Implementation

The Migration Support for the Actorlib provides the functionality to migrate Actors between UPC++ ranks. In order to send an object between ranks, it has to be explicitly serialized, unless it is trivially serializable by UPC++. In order to perform the serialization, the data required to encode the state of the class and any messages buffered must be sent to other rank and then reconstructed with the data.

The Actor in Actorlib with Extension consist of \( (ID, r, I, O, F, R, S, W) \). It adds a tuple of state tracking variables and new states during the migration to the Actor defined in the original paper\cite{41}. The component \( r \) (rank id) and the \( F \) (functions) are not required for the serialization, since the functions and the rank identification is already available on the other rank. An InPort \( i \in I \) is encoded with a name and a channel: \( (n, ch) \). The Channel \( ch \) is a finite storage of type \( t \) with a buffer capacity of \( c \). OutPort is encoded with a name and a binding to a channel \( cb: (n, cb) \). The state of InPorts, and accordingly the OutPorts and the messages buffered in the Channel are serialized and then reconstructed in the remote rank. The implementation specific details are further discussed under this Chapter.

The Chapter is separated into 5 sections, where each section describes one of the main components required by the migration support, with the first chapter explaining the additional features added to Actorlib in order to enable serialization of Actors and its dependencies, for example various components and classes from Actorlib and SWE were provided with move constructors and serialization methods. The 2nd to 5th sections describe the 4 main components implemented. PortGraph saves the connections between Actors, Dynamic ActorGraph provides a system to introspect run-time type information and to allow interrupting the ActorGraph. Migration allows sending Actors between ranks and reconstructing Actors on remote ranks. The last main component Market provides a default metric to the choose Actors for migration.

4.1. Added features

**Move constructors** The serialization process uses remote procedure calls to send the data and to perform checks on it. It is not possible to use \texttt{rget} or \texttt{rput}, as the Actors are not trivially serializable. This means that Actors cannot be copied with a byte copy, for example with \texttt{std::memcpy}. \texttt{upcxx::rpc} requires its arguments to be move constructible. The following code snippet will not compile:
class NoMove {
    public:
    int a = 42;
    NoMove(NoMove&& other) = delete;
};

//...

NoMove nm;
upcxx::intrank_t to = (upcxx::rank_me() + 1) % upcxx::rank_n();
bool check = upcxx::rpc(to, [] (NoMove nm) {
    return nm.a;
}, nm).wait();

The following error message is encountered during the compilation:

```
error: no matching function for call to 'rpc(int, main()::<lambda(NoMove)>, NoMove&)' |
| }, nm).wait();
```

Therefore move constructors have to be provided for all classes.

**Serialization Structs** For all of the classes that does not fully consist of `serializable` or `triviallySerializable` members, a serialization method has to be provided. A serializable member has either one of the serialization methods defined by UPC++ available, or it is triviallySerializable. A class is triviallySerializable if the class is copy constructible with a byte copy, for example with `std::memcpy`. UPC++ provides several options to serialize complex user defined classes. The preferred option while implementing the Actorlib-Extended was to provide the member `struct upcxx_serialization with the static void serialize<Writer> and static T *deserialize<Reader,T> functions.`

**Serialization specification** UPC++ provides serialization for the most of the standard library containers. Although UPC++ can serialize the `std::optional` when compiled with gcc, it could not when compiled with Intel C++ compiler. Therefore additional specification for the serialization of the `std::optional` was provided in `SerializeSpec` file in the library. A template specialization for `template <class T> struct upcxx::serialization` is provided for `std::optional<T>`. The template specialization is required for serializing externally defined classes, such as classes defined in the standard library. For further information refer to the Serialization chapter of the UPC++ specification [5]. See the Appendix for the used compiler and library versions.

**Support to remove Actors** The actor graph from Actorlib supported adding Actors but not removing them. The serialization adds a copy of the Actor on the remote rank and it
4. Implementation

requires removing, as well as deleting, the old Actor. In order to add the copy on a remote rank, it has to be removed from the ActorGraph.

Add/Remove Actor to/from remote rank

In order to add/remove Actors to/from a remote rank, one can try to call the `addActor` or the `rmActor` within a RPC. But the calling these functions in a RPC results in a double procedure call, because add and remove operations broadcast the entry or the removal to all ranks. A double RPC can not terminate, therefore functionality to add/remove Actors to/from remote ranks was implemented. Following code snippet exemplifies a double RPC where rank 0 issues a RPC to rank 1, which further executes a RPC to rank 2:

```cpp
if(upcxx::rank_me()==0){
    int back = upcxx::rpc(1, [](){
        return upcxx::rpc(2,[](){
            return 42;
        }).wait();
    }).wait();
    std::cout << "What have I done?" << std::endl;
}
```

The code snippet will not terminate and the message will never be printed. If compiled in debug mode the following error will be received:

```
You have attempted to wait() on a non-ready future within upcxx progress, this is prohibited because it will never complete.
```

Tracking Work Done

A straightforward approach to choose the Actors and ranks for the migration is to identify under-utilized and over-utilized ranks. In order to compare the work done, tracking it is required. The ActorGraph can measure the time spent in each actor’s `act()` method or approximate the run-time of act method with Tokens and save the total working time in member variables.

4.2. PortGraph

When reconstructing an Actor $A_i$ on a different Rank $r$, all incoming and outgoing connections to and from other Actors must be preserved. A Channel connects an InPort to an OutPort. From a given port, the connected Actors’ names can be extracted.

After a copy of the Actor $A_i$ (originally on $i$) is sent to a remote rank ($j$), it has to reconnect to same neighbors it had. This requires extracting the information of the neighbor Actors
through their ports. If a port connects to an Actor on the original or a third rank \((l \neq j)\), a RPC is issued to the rank \((l \neq j)\). In this case or when the send call was fired from a third rank \((m)\), it will result with a double RPC \((m \rightarrow i \rightarrow j\) or \(i \rightarrow j \rightarrow l)\), which does not terminate.

In order to allow migration between any rank, the connections between Actors have to be saved in a separate structure that needs to be accessible to each rank without any RPC. The desired functionality is implemented in the PortGraph class.

### 4.2.1. PortGraph

The Connection between Actors can be modeled as a tuple of Actor and In-/OutPort combinations, e.g., \(((A_i, OutPort_{ij}), (A_l, InPort_{lm}))\) The connection is one-sided therefore the PortGraph is a directed graph. The edges of an Actor is saved in a Node class.

```cpp
#include <vector>

class Node{
    std::vector<std::pair<std::string,string>> outgoing;
    std::vector<std::pair<std::string,string>> incoming;
};
```

The outgoing vector saves all the outgoing edges from the Node, the incoming vector saves all the incoming edges to the Node. The first entry of an element in outgoing is the OutPort_{ij} of the represented Actor \(A_i\), second entry is the combination of the other Actor’s name \(A_l\) and other Actor’s InPort name \(InPort_{lm}\). The first entry of outgoing is the OutPort_{in} of the represented Actor \(A_i\), second entry is the combination the other Actor’s name \(A_k\) and other Actor’s InPort named \(InPort_{kp}\).

Double saving the edges allows \(O(1)\) access to incoming edges; otherwise, it would result in iterating the whole graph. The wrapper functions for addActor connectPorts of the DynamicActorGraph adds the corresponding entries automatically to the port graph.

![Figure 4.1.: A Simple Graph of Actors](image)

The \(Actor_0\) connects to \(Actor_1, Actor_1\) to \(Actor_2\) and \(Actor_2\) to \(Actor_0\). When assumed that every InPort has the name "IN" and OutPort "OUT". The node for \(Actor_0\) looks as following:

```cpp
Node actor0;
```
4. Implementation

```c++
actor0.edgesTo = {("OUT","actor-1§§IN")};
actor0.edgesFrom = {("IN","actor-0§§OUT")};
```

Combination and splitting of the string is performed automatically by the PortGraph.

4.3. Dynamic ActorGraph

Abstract base class `Actor` is used as an interface for Actor implementations and the ActorGraph erases the type of the Actor implementations. Sending Actors requires the run-time type information of the Actor for the reconstruction at the remote rank, therefore a component for the run-time type introspection is needed. Intended behavior is achieved with variadic templates, `std::variants` and the help of the `type_traits` library, similar to the approach proposed by Bayser and Cerquaira [8].

4.3.1. Type Extension for ActorGraph

```c++
//an Actor implemented by the user
Actor* a1 = ::new ActorImpl1();
//another Actor implemented by the user
Actor* a2 = ::new ActorImpl2();
```

The instances `a1` and `a2` both have the static type `Actor` but differ in their dynamic types. Type erasure is required to deal with multiple types of Actors but the run-time type information has to be regained in order to migrate the instances, with their true types. While it can be tempting to delegate the serialization to virtual helper functions to serialize and deserialize the true type, it is impossible due to the template parameters Writer and Reader that cannot be passed down to virtual functions. The function signatures for serialize and deserialize methods are provided below:

```c++
struct upcxx_serialization{
    template<typename Writer>
    static void serialize (Writer& writer, ActorImpl const & object) { ... }

    template<typename Reader>
    static ActorImpl* deserialize(Reader& reader, void* storage) { ... }
};
```

Trying to serialize `a1` or `a2` forwards the call to the `Actor` and not to their respective dynamic types. It results in the loss of the attributes specific to the derived class and the `act()` method.
Therefore the pointers \( a1 \) and \( a2 \) need to be downcasted to their dynamic types before the serialization.

### 4.3.2. MultipleActorAGraph

Downcasting to the right type requires access to type information on run-time. The `type_traits` library can be used to acquire the run-time information about the class. This approach requires the user to specify and implement a `downcaster(Actor* a)` function that downcasts the pointer of the abstract `Actor` class to the type of the derived `Actor` class, as well as a `variant` or any containing every possible `Actor` implementation to enable sending it to a remote rank.

```cpp
ActorWrapper downcaster(Actor* a){
  if(a->getDynamicName().compare("aimpl1") == 0){
    ActorImpl1* x = dynamic_cast<ActorImpl*>(a);
    return ActorWrapper(x);
  } else { ... }
}
```

Forcing the user to write the `downcaster(Actor* a)` and other functions that work with the `Actor` implementations will result in repetitive code that could have been avoided. The `downcaster` and similar functions can be implemented with variadic templates, thus relieving the user from the burden of writing switch cases iterating through all types.

`MultipleActorAGraph<A,As...>` builds a wrapper around the `ActorGraph` class. The `ActorWrapper<A,As...>` uses `std::variant` to save an `Actor` implementation and to send the wrapper to remote ranks in a unified function. The `ActorWrapper` and the underlying `std::variant` utilizes recursive functions that iterate through the types and executes statements for the type saved in the variant.

```cpp
template<class T>
void rec_visit(Actor* a){
  if (same_type(type(a),type(T))){
    do_something();
  }else{
    throw std::runtime_error("Unsupported Type");
  }
}

template<class T,class U,class... Ts>
void rec_visit(Actor* a){
  if (same_type(type(a),type(T))) { 
    do_something();
  }else{
```
4. Implementation

With the run-time type introspection system, arbitrary Actors can be sent between ranks. Instantiating the type extension with only one type will still encapsulate the only Actor implementation in a `std::variant` class and the functions that are used to extract the run-time type information will introduce unnecessary overhead, which can be easily avoided. Since the type is trivially deducible (single type), a less-overhead version is of the `MultipleActorAGraph` is provided, the `SingleActorAGraph`.

4.3.3. SingleActorAGraph

If there is only one type of Actor, then the recursive functions can be replaced by a more direct approach, as the ActorWrapper is not necessary and the pointer to Actor can and only point to the sole implementation of the Actor class, and can be directly downcasted.

```
downcaster(Actor* a) {
    T* t = dynamic_cast<T*>(a);
    do_stuff(t);
}
```

4.3.4. Sending Actors

Actor migration requires an Actor to be serialized and to be sent to a new rank. The procedure for the sending of an Actor from rank $i$ to $j$ depends on the rank $x$ that initiates migration. If the migration from $i$ to $j$ is initiated from $j$ then $j$ retrieves the actor with an RPC, if from $i$ then the rank sends the Actor with an RPC to $j$ and get its global pointer, if initiated from a rank $x \neq i,j$ then the actor must be transferred to $x$ and then sent from $x$ to $j$. The additional step is mandatory, because sending the Actor from $i$ to $j$ already requires a RPC, the second RPC on $x$ is unavoidable. The simplified pseudo-code for the send procedure can be found below. The `recons` function reconstructs the Actor in heap space and returns a global pointer to the allocated Actor.

```
upcxx::future<GlobalActorRef> send(upcxx::intrank_t j, GlobalActorRef a) {
    upcxx::future<GlobalActorRef> glob;
    if (upcxx::rank_me() == a.where())
        & & upcxx::rank_me() == j) {
        // sending $i \rightarrow i$ is meaningless
```

26
4. Implementation

```cpp
std::string s =
    "No Stationary Send (i -> i) is allowed";
throw std::runtime_error(s);
}
else if (upcxx::rank_me()==a.where()) {// send only
    glob = upcxx::rpc(j, recons, *a);
}
else if (upcxx::rank_me()==j) {// retrieve only
    glob = retrieve(*a);
}
else {// retrieve, then send
    Actor aCopy = retrieve(a).wait();
    glob = upcxx::rpc(j, recons, aCopy);
}
return glob;
```

The details for the serialization is explained under Serialization.

4.4. Migration

When the ActorGraph is not running the Actors can be sent between ranks via the Migration component. Migration accepts a map consisting of Actor names and the destination ranks that have claimed the Actors. The class Market is implemented to provide a default way to choose the Actors and their destinations for the migration.

Migrating Actors is a highly serial process, and each step $i$ requires the completion of all steps from $i-1$ to 1. The 5 steps consist as follows:

- Stop
- Preparation
- Send
- Reconnection
- Restart

4.4.1. Stop

This step is it is automatically preformed by the DynamicActorGraph. It calls the run method of ActorGraph in a while loop with timeout limits until all Actors terminate. The ActorGraph stops each actor before returning to the DynamicActorGraph after exceeding the timeout interval.
4. Implementation

4.4.2. Preparation

An Actor has to save its state. For example it needs to serialize the state of its In- and OutPorts. The data for the ports can be generated in the preparation phase during a call to the `prepare()` procedure. If additional preparation is needed user can provide an implementation for the `virtual prepare_derived()` method. For example: the SimulationActor in the pond application (with file output enabled and compiled with NetCDF) closes the NetCdf file and releases the ownership in the preparation phase. In the preparation phase each rank calls the prepare method on their local Actors.

```cpp
// for all Actors that are to be migrated
for (std::pair<GlobalActorRef, upcxx::intrank_t>& pr : migrationMap){
    GlobalActor ref = getActor(pr.first);
    if(ref.where() == upcxx::rank_me()){
        // if local call prepare
        Actor *a = *ref.local();
        a->prepare();
    }
}
```

4.4.3. Send

In the send step each rank iterates over the map of Actors to their destination ranks and sends the local Actors to their destinations. The arrival of the Actors is completed when all the futures are waited on. The futures are combined to achieve faster migration time.

```cpp
std::vector<GlobalActorRef> send(
    const std::vector<GlobalActorRef>& refs,
    const std::unordered_map<std::string, upcxx::intrank_t>& migList)
{
    // vector of futures
    std::vector<upcxx::future<GlobalActorRef>> refs;

    for (auto& el : refs){
        if (el.where() == upcxx::rank_me()){
            Actor* a = *el.first.local();
            auto it = migList.find(a->getName());
            if (it != migList.end()){
                upcxx::future<GlobalActorRef> ref =
                dag->send_actor_to_async(it->second, el);
                refs.push_back(std::make_pair(ref, *it));
            }
        }
    }
```
4. Implementation

```cpp
std::vector<GlobalActorRef> = wait_on(refs);
```

The call to `send_actor_to_asnyc` serializes the Actor and reconstructs it on the recipient rank and returns a future to a global pointer. Since the function is called from the rank that Actor a has affinity to, only the reconstruction on remote rank is required (and no retrieve).

Combining futures that return values will return the tuple of these values instead. Assume that `conjure` method retrieves an int from another rank:

```cpp
#include <tuple>

//...

int conjuredval = conjure().wait();
std::tuple<int, int, int> vals = upcxx::when_all(conjure(), conjure(),
conjure()).wait();
```

Due to the compile time constraints of the `std::tuple` it is impossible to call `when_all` on the migration list, or any container with a dynamic size. The `upcxx::when_all` accepts a variadic number of arguments and does not work with iterators. To prevent separate waiting (of futures) a semi-bulk waiting approach is implemented where the migration list is divided into parts with each length of 16. If there are less then 16 elements then the approach tries to wait respectively on 8 or 4 futures. It waits separately on the remaining futures.

```cpp
int i = 0;
auto interncopy = migList;
while(i<interncopy.size()){
    if ( (size-i) > 16){
        wait_on16(interncopy);
        cull_front(interncopyt, 16);
        i+=16;
    }else if ( (size-i) > 8){
        wait_on8(interncopy);
        cull_front(migList, 8);
        i+=8;
    }else if ((size-i)>4){
        wait_on4(interncopy);
    }
```
4. Implementation

```c
   cull_front(migList, 4);
i+=4;
} else {
   wait_on(interncopy);
   cull_front(migList, 1);
i+=1;
}
}
```

The implemented `wait_onX` functions waits on the futures of the first \( X \) elements of the `interncopy`, therefore the completed futures are removed from the container with a call to `cull_front` that culls the first \( X \) elements.

### 4.4.4. Re-connection

After each Actor has arrived at their destination rank, they are re-connected to their neighbors. The list of neighbors are already saved in the PortGraph and remain unchanged during computation. The re-connection step cannot be executed during the migration step; because re-connecting an Actor, whose neighbor are still being migrated will result in a segmentation fault or the re-connection failing.

![Figure 4.2: A simple ActorGraph where A0 sends data to A1 and A1 to A2](image)

Figure 4.2.: A simple ActorGraph where A0 sends data to A1 and A1 to A2

![Figure 4.3: A0 can connect to its neighbors only after the migration of A1 is completed and A1 is re-connected to the graph](image)

Figure 4.3.: A0 can connect to its neighbors only after the migration of A1 is completed and A1 is re-connected to the graph

It is impossible to connect \( A0 \) to \( A1 \) until the \( A1 \) arrives at its destination. In the re-connection step all In- and OutPorts of an Actor are connected to the same ports they were connected to before the migration. The information is saved in the PortGraph. `ActorGraph::reconnect_to_neighbors` (or the asynchronous variant) can be called to connect the Actors to their neighbors. Due to the length of the code only the pseudo code for the re-connection is provided.
### 4. Implementation

**Data:** ActorGraph = $AG(V, E)$, PortGraph

\[ = PG(X, F) : X = (V, O) \cup (V, I), \quad F = ((V, O), (V, I)), \quad I = \text{InPorts}, \quad O = \text{OutPorts}, \]

Actor $v$

vector incoming

vector outgoing

for $(v,o)$ where $\exists (w, i) \in (V, I) : ((v, o), (w, i)) \in E$ do

| outgoing $\leftarrow (o, (w, i))$

end

for $(v,i)$ where $\exists (w, o) \in (V, I) : ((w, o), (v, i)) \in E$ do

| incoming $\leftarrow ((w, o), i)$

end

for $(o,(w,i)) \in \text{outgoing}$ do

| reconnectPorts($v,o,w,i$)

end

for $((w,o),i) \in \text{incoming}$ do

| reconnectPorts($w,o,v,i$)

end

return

**Algorithm 3:** Pseudo-code representation of the implemented $\text{connect\_to\_neighbors}$ function. Retrieves data from PortGraph and reconnects the ports

All information on the incoming and outgoing connections are retrieved from the PortGraph, and all of the connections are restored

### 4.4.5. Restart

After all Actors are connected, the Actors have to be prepared for the next cycle. The buffered messages have to be restored. Any migrated Actor will have its tracking variable $\text{needsActivation}$ set to true. In a call to $\text{run\_w\_migration}($double interval$)$, each actor with the boolean variable $\text{needsActivation}$ set to true are reactivated.

Each rank calls the reactivation procedure on their local Actors (similar to other steps). During the reactivation the buffered messages are restored and the capacity is set to the state before migration. Essentially it is same as copy assigning the ports the saved states.

### 4.5. Market

A metric has to be defined to choose the Actors for migration. The Market component offers a metric to choose appropriate migrations. It assigns prices to Actors proportional to work they have performed, ranks with higher currency can buy Actors from ranks with lower
currency. Currency is assigned to ranks again proportional to total work they have done. With the Market the DynamicActorGraph can be viewed as a multi-agent system that tries to balance its work-load among its nodes. Assigning the currency depends on how the work is tracked, the Actorlib-Extended provides two options:

4.5.1. Tracking Work Done

- Tokens
- WorkTime

Tokens

If the ActorGraph contains only one type of actor, it can be assumed that the \texttt{act()} will need roughly the same amount of time for any Actor. Instead of measuring the time spent in the \texttt{act()}, a WorkToken can be used to approximate one run-time of the \texttt{act()} method. This approximation will only hold if each Actor \texttt{act} method is identical and working in homogeneous environment.

WorkTime

The time spent in a \texttt{act()} call, can be saved in a member variable in order to track the work done. Although requiring more calculations, it is more precise. Distributing Actors in a way that all nodes compute for X seconds in a X seconds interval can assure an equal balance of the workload even in heterogeneous environments, whereas token approach will be feasible only for homogeneous environments.

4.5.2. Distributing Currency

ActorGraph’s \texttt{run_w_migration(double interval)} method accepts a timeout argument in seconds, after the timeout, the migration phase fires (if enabled) and can be used to migrate Actors between rank. Per default it uses the metric provided by Market to calculate a map to migrate Actors to their destinations. The Market calculates work done by each rank and puts appropriate Actors on display, where they are claimed by ranks which have worked less.

\[
\text{Rank}_i := \{\text{actor} \mid \text{actor instantiated at Rank}_i\}
\]

Each Rank indexed from 1 to a natural number \(n\), consists of various amount of Actors.
4. Implementation

\[ \text{work}_{\text{actor}} := \text{Tokens} \mid \text{WorkTime} \]

The work done by an actor is an unsigned integer value representing Tokens or WorkTime. New options can be implemented by extending the ActorGraph and Market classes.

\[ \text{Work}_{\text{rank}_i} := \sum_{\text{actor} \in \text{Rank}_i} \text{work}_{\text{actor}} \]

Work done by the rank is the sum of work done by its Actors.

\[ \text{Currency}_{\text{rank}_i} := \max_{r \in \text{Ranks}} \{ \text{Work}_r \} - \text{Work}_{\text{rank}_i} \]

The currency of a rank is the difference between its work and the maximal work. Hence the Actors with less work can claim more Actors or Actors where the run method requires more time to complete.

\[ \text{Value}_{\text{actor}_i} := \text{Work}_{\text{actor}_i} \]

The value of an actor is the work done by it.

\[ \text{Slacker}_i := \min_{\text{actor} \in \text{Rank}_i} \{ \text{Work}_{\text{actor}} \} \]

Worst actor is the actor that was worked for the least of amount of time / tokens. In the default metric it is the first choice to put on display.

4.5.3. Putting Actors on Display

If unbalanceCheck is not enabled, each Rank can put one actor on display. If unbalanceCheck is enabled: then the ranks with abundance/scarcity of Actors are not bounded by the one-actor rule. A rank is treated as abundant in Actors if the Actor count of the rank is more than double the mean count of Actors (among all ranks), respectively considers it scarce in Actors if the count of Actors is less than the half of the mean count of Actors. The ranks with abundance are allowed put Actors on display as long as the abundance condition is fulfilled. The ranks with scarcity are not allowed to put Actors on display, in contrast they can claim multiple Actors if
they have the currency for it. Per default, ranks put Actor(s) with least Tokens/WorkTime on display. The Information of the Actor put on display is then broadcasted to every other rank.

Data: meanActorCount over all ranks

\[
\text{if } \text{unbalanceCheck then}
\]

\[
\text{if actorCount < meanActorCount/2 then}
\]

\[
\text{return}
\]

\[
\text{end}
\]

\[
\text{while actorCount > meanActorCount*2 do}
\]

\[
\text{put_actor_with_least_work()}
\]

\[
\text{actorCount} \leftarrow \text{actorCount} - 1
\]

\[
\text{end}
\]

\[
\text{put_actor_with_least_work()}
\]

\[
\text{end}
\]

Algorithm 4: Put on display procedure

4.5.4. Claiming Actors

After the ranks have put their Actors on display, those that have more than mean currency can claim Actors. The ranks are sorted in descending order according to their available currency and the rank with more available currency has precedence over a rank with less currency. Each rank tries to minimize their currency in the claim phase, buying the actor resulting in least remaining currency. Global ordering of ranks require one rank to coordinate the claim phase for all ranks. Therefore it receives the currencies of all ranks, calculates the Actor to destination map and broadcasts the result back to all ranks. The time requires for claim phase increases proportional to number of ranks and can cause a bottleneck with enough ranks. For measurements regarding the time required for the claim phase see \[5.2\]

The pseudo-code for the claim phase is provided below.
4. Implementation

Data: $\text{meanActorCount}$, currency of other ranks

get_currencies()

matches

for $\text{rank}_i \in \text{Ranks}$ do

    if $\text{unbalanceCheck}$ then

        if $\text{actorCount} > \text{meanActorCount} \times 2$ then
            continue;
        end

        while $\text{actorCount} < \text{meanActorCount} / 2$ do
            matches ← claim_best_match(i)
        end

    else

        matches ← claim_best_match(i)
    end

end

for $\text{rank}_i \in \text{Ranks} \land i \neq 0$ do

    broadcast(matches, i)

end

return

Algorithm 5: Claiming the Actors put on display. Performed by one rank, the results are then broadcasted to other ranks
5. Results

All tests were conducted on the Linux-Cluster CoolMUC-2 of Leibniz Rechenzentrum, with 28-way Haswell-based nodes and FDR14 Infiniband interconnect. For further details refer to LRZ documentation[1].

UPC++ used in measurements is compiled with the Intel C++ compiler with the MPI wrapper and configured to use the Gasnet-Ex's ibv-conduit for InfiniBand Verbs for the communication between nodes. MPI wrapper was required solely for the job launch (and was not used for communication). All the measurements, with the exception of some test cases, are performed with the Pond application built with the extensions for the Actorlib library. Pond is a shallow water proxy application based on SWE[14] and SWE-X10[42]. The results that do not derive from Pond have explanation for the test code used for the measurements. The measurements that derive from Pond use the rank based execution strategy, file output is enabled and the state of the cells are written 10 times to NetCDF files (one file for each actor) in order to keep similarity to real-life use cases. The used scenario is the scalable pool scenario, an out-of-box available scenario applicable to any grid size. For the CMake configurations used to build the pond executables, see the Appendix[8] especially the table B.1.

Shallow water equations can be derived from Navier-Stokes equations[37][41]. Shown below; x and y are the coordinates within the domain. g refer to the gravitational constant, h to height of the water volume, hu and hv to the momentum in x and y direction, where $S(x, y, t)$ models a source term.

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

SWE applies finite volume discretization on Cartesian grid[37], yielding the following update scheme for the shallow water equation shown below. SWE and Pond use an approximate

[1]https://doku.lrz.de/display/PUBLIC/CoolMUC-2
5. Results


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

(5.1)

The Calculations focus on 3 topics. The overall time needed for serializing objects, the overhead induced by interrupts and the effects of global (random) ordering, with measurements focused on the overhead caused by random partitioning, messages to remote ranks and possible use cases for global ordering.

- Serialization Time
  - Movable Fields
  - Bulk/Separate Waiting
- Overhead induced by Interrupt
- Global (Random) Reordering
- Message Chains
- Use-Case: Invade!

5.1. Serialization

Serialization is the first requirement of Actor migration and its speed directly affects the time required for the migration. There are two main factors that can accelerate serialization: existence of movable fields and serializing multiple actors in bulks, thus utilizing the dead time caused by calls to future::wait by postponing the wait calls to a later time. The described factors are studied under Move Constructors and Bulk Sending in the following subsections.

5.1.1. Movable Fields

Actorlib-Extended uses RPCs to send Actors between ranks. \texttt{rpc} ( \texttt{inrank_t recipient, Func \&\& func, Args \&\&... args}) requires rvalue arguments, and any class that needs to be serialized needs to define or have implicitly defined move constructors. But the movable fields can result in a slightly better serialization time since the moving the class will require less copy operations compared to copy constructing the class, yet the amount of data that needs to be serialized does not change. In order to exemplify the impact of having movable fields
5. Results

compared to no movable fields, the following test is constructed. On \( n \) ranks, the lower half of the ranks \((i < \frac{n}{2})\) send 250 stub-actors to upper half of the ranks \((j > \frac{n}{2})\). The stub-actor does not provide any functionality but mimics the size requirement of a SimulationActor used in pond. The stub-actors are saved in a list and each rank iterates through the list sending actors. The stub-actor class consist of an float array \((PA)\) of size \(N \times N\) modeling the patch area, 40 float arrays of size \(N\) modeling message buffers of size 10 that sends and receives the ghosts cells of size \(N\) for the neighbor count of 4 \((Buf)\), and an array of integers of size 100 to represent the internal variables \((Var)\). The array \(PA\) and \(Buf\) are heap allocated, whereas \(Var\) is not, and all of the arrays are initialized with random floating point numbers within the range the \([0, X]\). The move constructor of the stub-actor class described with \textit{Move} moves the \(Pa\) and \(Buf\) arrays, while the class described with \textit{NoMove} copies the data pointed by the member variables for \(Pa\) and \(Buf\).

![Figure 5.1.: Effect of having movable fields, where each rank sends \(i < \frac{n}{2}\) sends 250 stub-actors to the rank \(j = i + \frac{n}{2}\)](image)

The results show a slight performance increase with movable fields. The test was compiled in the debug mode to avoid any optimizations. With the slight difference in serialization time and the possibility of the compiler optimizations, redesigning classes in order to exploit movable members does not need to be a high priority.
5. Results

5.1.2. Bulk Waiting

Actor migration is a highly serial process where each step depends on the all of the previous steps. Global (random) ordering has the advantage that the separate \texttt{future: :wait} calls can be postponed until the end of the each step. This will allow saving time through utilizing the dead time caused by separate wait calls with execution of other local statements and waiting at the end of the step where no other local statements remain to be executed. The difference of bulk and separate sending can be visualized with the following pseudo-code.

\begin{algorithm}
\textbf{Data:} Map $M : \text{Actor} \rightarrow \text{Rank}_i$
\begin{algorithmic}
\State prepare($M$)
\State disconnect($M$).\texttt{wait()}
\State future $\leftarrow$ send($M$)
\State delete_remains($M$)
\State future.\texttt{wait()}
\State insert($M$).\texttt{wait()}
\State restart($M$).\texttt{wait()}
\end{algorithmic}
\caption{Bulk Waiting}
\end{algorithm}

\begin{algorithm}
\textbf{Data:} Map $M : \text{Actor} \rightarrow \text{Rank}_i$
\begin{algorithmic}
\For{$(a,i) \in M$}
\State prepare($a$)
\State disconnect($a$).\texttt{wait()}
\State future $\leftarrow$ send($a$, $i$)
\State delete_remains($a$)
\State future.\texttt{wait()}
\State insert($a$).\texttt{wait()}
\State restart($a$).\texttt{wait()}
\EndFor
\end{algorithmic}
\caption{Separate Waiting}
\end{algorithm}

Figure 5.2.: Difference between Bulk- and Separate waiting, functions are simplified to contrast the differences

The following migration times are extracted from an \textit{invade!}-run, where $n$ new ranks join the $n$ already working ranks. The working $n$ ranks send half of their actors to the newly joined ranks $n$. The example works on a grid size of 8000x8000 with the patch size 250x250 (with 1024 actors). In both \textit{invade!} examples 512 Actors are sent. Most of the ranks send 18 Actors, while some send 19, due to remainder of the uneven division $\frac{512}{28}$. 

39
5. Results

In both measurements separate waiting requires $\approx 6$ times more amount of time to serialize same amount of Actors. It can be assumed that both approaches will require roughly the same time for migrating a single Actor, having a mechanism to send multiple Actors is definitely advantageous in a case of a coarse workload imbalance, for example in the invade! use case scenario. With the faster serialization time of bulk waiting, it can even compensate for the redistribution of the whole graph if the job is expected to run for a longer period of time. The slight increase of the migration time when the migration process starts at 150 seconds compared to the start at 15 seconds can be tracked down to do amount of data that needs to be serialized. After 150 seconds of run time, it is possible that Actors have more buffered messages, resulting in more data that needs to be serialized.

The second test on bulk sending uses the same test used for movable fields example (5.1). This time each rank from the lower-half sends 250 stub-actors to their corresponding rank from the higher-half. Once with bulk waiting, and once with separate waiting.

Figure 5.3.: Effects of Bulk-Waiting and Separate-Waiting on invade! scenario serialization. Serialization begins at 15, 150 seconds.
When solely serializing possibly continuously allocated data, and not calling other functions (such as reconnect, restart), then the difference between the time required for separate and bulk waiting decreases but the serialization time still favors bulk waiting over separate.

![Figure 5.4.: (B)ulk vs. (S)eparate sending](image)

<table>
<thead>
<tr>
<th>Action</th>
<th>Runtime required for action [microseconds]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>2.36 \cdot 10^6</td>
</tr>
<tr>
<td>Send</td>
<td>1.94 \cdot 10^6</td>
</tr>
<tr>
<td>Delete</td>
<td>1.88 \cdot 10^5</td>
</tr>
<tr>
<td>PrepareActors</td>
<td>1.38 \cdot 10^5</td>
</tr>
<tr>
<td>Claim</td>
<td>71,288</td>
</tr>
<tr>
<td>GlobalVar</td>
<td>7,295</td>
</tr>
<tr>
<td>Reconnect</td>
<td>6,506</td>
</tr>
<tr>
<td>Reinsert</td>
<td>4,412</td>
</tr>
<tr>
<td>Disconnect</td>
<td>3,764</td>
</tr>
<tr>
<td>PrepareIter</td>
<td>202</td>
</tr>
<tr>
<td>FinishUp</td>
<td>134</td>
</tr>
<tr>
<td>Refill</td>
<td>110</td>
</tr>
<tr>
<td>ReduceFlag</td>
<td>47</td>
</tr>
</tbody>
</table>

![Figure 5.5.: Time Decomposition of the Migration Phase of bulk waiting strategy in the invade! scenario. From 1 node to 2 nodes](image)

Above (5.6) is the time decomposition of the migration time from the invade! scenario (with the migration starting on 15 seconds) is visualized. The time for migration is dominated by sending of Actors. The share of send and other parallel operations is excepted to remain same with increasing number of nodes. Those are: PrepareActors, Delete, Disconnect, Reinsert,
5. Results

PrepareIter, FinishUp and Refill. The share of reduction operations; ReduceFlag, and GlobalVar is expected to increase with an increasing amount of ranks (or nodes). A reduce operation assigns a variable, which is available on every rank, the result of an operation applied to the variable on every rank (e.g. building the sum of the variable x from every rank). ReduceFlag is used to deduce whether the ActorGraph needs to run another iteration, reduced from each ActorGraph’s finished boolean return value. GlobalVar calculates the global relative variance of work done, from each rank’s tracked work. The parallel operations Delete, Reconnect, Reinsert, Disconnect, Refill, Send and PrepareActors correspond to their counterparts in the dependency graph discussed in Possible Approaches. Delete stands for deletion of the remains (old Actors). FinishUp and PrepareIter are parallel operations that are not part of the migration but are required to prepare for the next iteration of the ActorGraph, where specific variables are set and prepared. The Claim operation in the default Market implementation is a serial operation where a rank sorts all ranks in descending order depending on their currency and claims Actors for each rank. Claim operation will increase it percentage (on the migration time) as the number of ranks (or nodes) increase, faster than the reduce operations.

5.2. Migration and Claim Phase

The following migration-phase time decompositions are taken from migration-run. Each migration-run has the grid size 8000x8000 and patch size 250x250. The 3 decompositions differ in their node count; respectively 2, 4 and 8. The total time required for migration is noted by total. The duration for local functions (where only local actors are used) are the time required for rank 0’s completion, it is possible that rank 0 needs to wait for other ranks’ completion, thus the sum of all action do not necessarily sum up to total run-time. Two exceptions arise: Claim, as it is a global action, where all ranks complete the action at the same time and Total, the time when the migration phase ends is also the max completion time among ranks.

The run-time of each action depends strongly on the internal-state of each rank, therefore more detailed analysis of parallel actions require further analysis and more detailed measurements. The total migration time decreased with increasing number of nodes, mainly due to each rank having less actors to operate on, as the grid size and patch size remains the same but the same of amount of actors were distributed among more ranks. The global claim action requires 14,809 microseconds for 2 nodes, increasing to 20,956 and 44,993 with respectively 4 and 8 nodes. The time required for the claim action increases, as expected, with the number of nodes but the migration time is still dominated by the send and delete actions. The claim phase will start to have a noticeable impact on the migration time by a substantially higher amount of nodes.

The decompositions are provided on the next page:
## 5. Results

### 2 nodes, Runtime required for action [microseconds]

<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>$3.46 \cdot 10^6$</td>
</tr>
<tr>
<td>Send</td>
<td>$2.37 \cdot 10^6$</td>
</tr>
<tr>
<td>PrepareActors</td>
<td>$6.05 \cdot 10^5$</td>
</tr>
<tr>
<td>Delete</td>
<td>33,239</td>
</tr>
<tr>
<td>Disconnect</td>
<td>10,200</td>
</tr>
<tr>
<td>Claim</td>
<td>14,809</td>
</tr>
<tr>
<td>GlobalVar</td>
<td>6,496</td>
</tr>
<tr>
<td>Reconnect</td>
<td>1,979</td>
</tr>
<tr>
<td>Reinsert</td>
<td>7,261</td>
</tr>
<tr>
<td>Refill</td>
<td>10,200</td>
</tr>
<tr>
<td>GetReady</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>210</td>
</tr>
</tbody>
</table>

### 4 nodes, Runtime required for action [microseconds]

<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>$2.69 \cdot 10^6$</td>
</tr>
<tr>
<td>Send</td>
<td>$7.47 \cdot 10^5$</td>
</tr>
<tr>
<td>PrepareActors</td>
<td>$6.54 \cdot 10^5$</td>
</tr>
<tr>
<td>Delete</td>
<td>$3.93 \cdot 10^5$</td>
</tr>
<tr>
<td>Disconnect</td>
<td>22,172</td>
</tr>
<tr>
<td>Claim</td>
<td>20,956</td>
</tr>
<tr>
<td>GlobalVar</td>
<td>6,029</td>
</tr>
<tr>
<td>Reconnect</td>
<td>1,465</td>
</tr>
<tr>
<td>Reinsert</td>
<td>8,525</td>
</tr>
<tr>
<td>Refill</td>
<td>199</td>
</tr>
<tr>
<td>GetReady</td>
<td>330</td>
</tr>
</tbody>
</table>

### 8 nodes, Runtime required for action [microseconds]

<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>$2.59 \cdot 10^6$</td>
</tr>
<tr>
<td>Send</td>
<td>$1.72 \cdot 10^6$</td>
</tr>
<tr>
<td>PrepareActors</td>
<td>12,685</td>
</tr>
<tr>
<td>Delete</td>
<td>$4.56 \cdot 10^5$</td>
</tr>
<tr>
<td>Disconnect</td>
<td>10,999</td>
</tr>
<tr>
<td>Claim</td>
<td>44,993</td>
</tr>
<tr>
<td>GlobalVar</td>
<td>1,440</td>
</tr>
<tr>
<td>Reconnect</td>
<td>988</td>
</tr>
<tr>
<td>Reinsert</td>
<td>10,222</td>
</tr>
<tr>
<td>Refill</td>
<td>147</td>
</tr>
<tr>
<td>GetReady</td>
<td>290</td>
</tr>
</tbody>
</table>

Figure 5.6: Time Decomposition of the migration phase with bulk waiting strategy in migration-run.
5. Results

5.3. Overhead Induced by Interrupt

Implementing a way to interrupt the ActorGraph is one of the requirements for the global (random) ordering. The following measurements show the run-time difference between the old run method of ActorGraph and the new interruptable run method. In the old run method, the computation is completed in one call to the `ActorGraph::run()` method, also referred to as a plain-run. The interruptable run method takes a double value as timeout interval in seconds `ActorGraph::run(double timeout)`, the ActorGraph returns after the timeout interval is exceeded or the computation is completed, the information can be read from the returned boolean `finished` value. The run method for the interruptable run calls the run method with a timeout interval of 128.0 seconds in a `while(true)` loop until the ranks returns `true`, also referred as an interrupt-run.

![Runtime vs Node Count](image)

Figure 5.7.: Overhead caused by Interrupt

It is hard to see the difference between the run-time with (interrupt-run) and without interrupts (plain-run). The overhead induced by the interrupt method is negligibly small. The close-to-zero overhead is partly due to the runtime environment. All the scenarios/tests were run in a relatively homogeneous environment. Although all ranks stop execution at the same time (negligible even with high ping when computing over a net of nodes over ssh), if a rank or node is visibly slower than other ranks/nodes, then it can result with dead-time caused by the speed difference in executing the in-flight RPCs and LPCs. Therefore tests in a highly heterogeneous environment should be conducted before claiming that the overhead caused interrupt-run is negligible (compared to plain-run) in any environment. As it is, it is viable to interrupt ActorGraph to recalculate the maximum time step, to create checkpoints or for migration with bulk waiting.

---

44
5. Results

5.4. Migration with Global Random Reordering

In the figure below, the run-times for the job with a cell count of 8000x8000, end time 16 and patch size 250x250 to be found. The same job is executed with plain-run and interrupt-run strategies as explained above. In the migration-run the plain run method `run()` is changed with the `run_w_migration(double timeout)` method, where a migration phase is executed during an interrupt. Viable actors are chosen for the migration and sent to their destinations. The migration-run used in this measurement implements the Global (Random) Ordering and uses separate waiting, which is slower compared to bulk waiting (see 5.3). Interrupt and migration have their timeout values set to 128.0 seconds.

![Figure 5.8: Runtimes for pond with grid size 8000x8000, end time 16, patch size 250x250 with strategies plain | interrupt | migration on varying number of nodes](image)

The run-time difference between plain- and interrupt-run show a small variation but the migration-run performs far below expectations, with migration on a single node requiring 31000 seconds for completion. Before going into details on the reasons/culprits behind the
5. Results

slowing down, the run time difference is exemplified once more with the figure 5.11.

Work-Complexity

The small jobs that run on 1 or 2 nodes were tested on grid sizes of 2000x2000, 4000x4000 and 8000x8000 with end times of 4, 8 an 16. Due to small difference between interrupt and plain, and the immense gap between plain and interrupt. The values that are plotted on 3D space are transformed into 2D coordinates. With the following scheme:

The smallest job with grid size of 2000x2000 with end time 1 corresponds to 1 unit of work-complexity. Doubling the end time will double the work-complexity. For example the job with 2000x2000 grid size, but end time 2 will have 2 units of work-complexity. The same factors apply to grid size as well. Doubling the grid size doubles the work complexity. The mapping from 3D coordinates to 2D coordinates has its flaws. Since file output is enabled and jobs save the state of the cells 10 times, doubling the area has more effect on the run time compared to doubling of end time. In a case where multiple jobs have the same work complexity, their mean is calculated and plotted.

![3D coordinates values for run-time that depend on grid size and end time, on 1 node](image1)

(a) 3D coordinates values for run-time that depend on grid size and end time, on 1 node

![3D coordinates values for run-time that depend on grid size and end time, on 1 node](image2)

(a) 3D coordinates values for run-time that depend on grid size and end time, on 1 node

Figure 5.10.: Values plotted depend on size and end time are transformed into a 2D plane. The grid size of 2000x2000 with end time 1 corresponds to 1 unit of work complexity. Doubling the end time doubles the work-complexity, doubling the size quadruple the work-complexity.

The figure 5.10 show the 2D coordinates for run-time that depend on the work complexity. The following figure (5.11) uses the 3D coordinates mapped to 2D through Work-complexity. In the appendix, although hard to read, the 3D plot for the run-times is available, see Appendix D, figure D.1 and D.2
5. Results

Although all strategies below (5.11) show a linear increase over work-complexity, the run-time of migration strategy increases roughly 3 times faster compared to the plain strategy.

![Figure 5.11.: Runtime Plain-Interrupt-Migration](image)

The main culprits behind the run-time increase is the random partitioning caused by the migration of actors to remote ranks, creating new partitions increasing the communication overhead, and the global lock required for the migration. The time required for the migration phase for example in the invade scenario (5.3) amounts to 14.3 seconds with separate waiting and 2.35 with bulk waiting. The time lost during the global ordering through migration cannot account solely for the run-time increase. The effect of breaking message chains (and the partitioning) is studied in the following section.
5.5. Message Chains

A message chain exists if the messages travel within a rank. If the message is sent to another rank then the message chain breaks at that connection. The chains are broken on edge cuts. The rank that receives the message may share memory with the sender (same node) or not (different nodes), which can further affect the delivery time of the message. In order to measure the delivery time on different conditions, 3 basic scenarios are constructed.

In the first scenario (a), referred to as chainSM or $0 \rightarrow 1 \rightarrow 1$, the message is sent between 3 Actors in a cycle and all the Actors are stationed in the same rank indexed with 0. The message chain is not broken. In the second scenario (b), referred to as retourSM or $1 \rightarrow 2 \rightarrow 1$, the message from Actor A0 to A1 needs to change ranks since the Actor A1 is stationed on the rank indexed with 1 while others are on 0. Same happens with the message sent from A1 to A2. All the ranks are in the same node, thus share memory. In the third scenario, similar to the second one, referred to as retourDM or $0 \Rightarrow 2 \Rightarrow 1$, the Actor A1 is on a different rank but the rank is on another node, and does not share memory with other ranks. The messages have to be sent to/ received from other node.

In the test a vector with size 500 and type float, initialized to random numbers, is passed between the actors, while each Actor performs a floating point operation on each element of the vector before passing the it further. Each Actor stop after processing and passing the vector for $3 \times 10^5$ times.
5. Results

![Figure 5.13: Message Travel Time for Separate Cases. Measurements in ms.](image)

<table>
<thead>
<tr>
<th>Connection type</th>
<th>Arrival time [milliseconds]</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainSM</td>
<td>28,885</td>
</tr>
<tr>
<td>retourSM</td>
<td>$1.27 \cdot 10^5$</td>
</tr>
<tr>
<td>retourDM</td>
<td>$1.79 \cdot 10^5$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 \rightarrow 1 \rightarrow 1</th>
<th>1 \rightarrow 2 \rightarrow 1</th>
<th>1 \Rightarrow 2 \Rightarrow 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3672</td>
<td>15883</td>
<td>22422</td>
</tr>
<tr>
<td>3605</td>
<td>15952</td>
<td>22561</td>
</tr>
<tr>
<td>3587</td>
<td>15815</td>
<td>20813</td>
</tr>
<tr>
<td>3608</td>
<td>15830</td>
<td>22815</td>
</tr>
<tr>
<td>3588</td>
<td>16010</td>
<td>22760</td>
</tr>
<tr>
<td>3635</td>
<td>15637</td>
<td>22600</td>
</tr>
<tr>
<td>3596</td>
<td>15838</td>
<td>22583</td>
</tr>
<tr>
<td>3594</td>
<td>15846</td>
<td>22666</td>
</tr>
</tbody>
</table>

It is expected that the 1 \rightarrow 1 \rightarrow 1 will have the shortest arrival time for $3 \cdot 10^5$ messages, and the 1 \Rightarrow 2 \Rightarrow 1 the longest. Even when the ranks share the memory, the huge run-time difference between 1 \rightarrow 1 \rightarrow 1 and 1 \Rightarrow 1 \Rightarrow 1 is to be noted.

5.6. Use Case: Invade!

When new ranks ($m$ many) join a running computation, the workload has to be redistributed. This is done through the initial $n$ ranks invading unclaimed ranks by sending their Actors. The global ordering approach with bulk waiting is tailor-made for the invade! scenario. Since the territorial expanse! is not implemented as a part of the thesis and a hard coded example would require heavy communication between pond application and the Actorlib library, pond executables used in this use case randomly distribute the actors before starting the computation, creating a far-from-optimal random partitioning of the actor graph. Thereby the performance loss due to random partitioning can be disregarded, as it will apply to each and every execution.

The invade! scenario requires special configuration for pond specific to the scenario. The executable for the plain run needs to redistribute Actors randomly before starting the computation, and the invade! executable needs to start with Actors distributed only to the first half of
the ranks 1..n. After the first interrupt the Actors are migrated to the empty ranks (n + 1)..2n. Normally
the first timeout interval is set to 128.0 seconds, but in this use case the timeout interval is 64.0 seconds,
therefore the migration phase begins after 64.0 seconds. The test was conducted with the end-times 1 and 2 on a grid with size 8000x8000 and patch size of 250x250.

The *invade!* scenario used in this use case is different to the scenario used in the 3.2 under the subsection *Bulk Waiting in invade!* where each rank sends half of its Actors. In order to fully randomize the partitions each actor is sent to a random new rank, with the constraint that each rank gets equal amount of Actors. If the division \( \frac{\text{gridsize}}{\text{patchsize}} \) is uneven then some ranks get \( n + 1 \) Actors, whereas some get \( n \). Thereby ensuring that the new actor partitioning will be random and far from optimal. *Invade!* uses the bulk sending strategy.
5. Results

Figure 5.14.: Invade $n \rightarrow 2n$ next to plain $2n$. Both with random partitioning of actors due to the missing implementation for territorial expanse. The simulation time is 1 second for the measurements above and 2 for the measurements below.

In both end-times invade! from $n$ to $2n$, run-time is competitive with run-time of plain $2n$. The value of invade-run is expected to be slightly higher compared to the corresponding plain-run, as in the invade! scenario the computing is performed with half of the resources until the first interrupt. The exception holds for the values calculated with an end time of 1 second. It can be seen that invade-run outperforms plain-run in the measurements with the
end-time equal to 2 seconds. The behavior may be caused due to the better partitions being created during the migration process compared to the fully random partitioning (not because of the implementation of the migration phase but due to sheer luck).

Although random partitioning of the actors will not be the case in a real life scenario (at least uncommon), re-partitioning of coarse actor-load imbalances is viable if the re-partitioning does not introduce unnecessary partitions to the graph. With implementation of the proposed territorial expanse it can yield better run-time compared to plain-run.
6. Conclusion

In this thesis the UPC++ Actor Library, Actorlib, was extended to support dynamic actor migration in order to dynamically balance the workload. Two approaches were proposed: global (random) ordering where the graph is re-partitioned without regarding the old partitioning, with Actors redistributed among nodes after each iteration, and the actor steal approach, where idle or under-performing ranks steal Actors from other ranks, similar to the work-stealing approach. Both approaches can be improved to not produce new partitions randomly, where only specific Actors that only connect to one’s ranks (from remote ranks) are stolen or redistributed. The so-called territory of the each rank, consisting of one’s Actors, can expand only by claiming territories of neighbor ranks.

The naive implementation of global (random) ordering that does not regard the territory perform visibly worse compared to the approach without dynamic actor migration, as the overhead induced by random partitioning of the graph outweighs the performance gained through better workload balancing. Although the run-time for global ordering is slower, sending \( n \) actors in bulks with the global ordering is faster compared to sending \( n \) actors separately therefore it can still be used in cases with coarse imbalances. For example in use-case *invade!* where \( n \) new ranks join the computation that do not posses any actors receive actors from already working \( n \) ranks.

The improved global ordering approach that regards territories can provide better runtimes compared to the approaches without dynamic workload balancing, as it will not introduce random partitions and the introduced communication overhead will remain minimal. The approach can be further improved where global ordering can be used to balance coarse workload imbalances where the finer imbalances are handled with the actor-stealing approach. Similar to proposed composite workload-balancing in Chameleon [36].
7. Future Work

The implemented naive approach to migrate Actors is not able to speed-up the computation with exceptions to some use cases mainly due to two reasons:

- Global Lock
- Unnecessary re-partitioning

Due to global lock on ranks, no rank can work during the migration procedure in the Global (Random) Ordering approach. Therefore the default Market implementation will yield better performance only when each rank actively participates in the process, meaning each rank receives or sends at least one actor. Default Market implementation with Global (Random) Ordering is therefore only applicable in a case of re-partitioning the ActorGraph, for example when new ranks join the computation.

7.1. Territorial Expanse!

Migrating inner Actors, that connect only the Actors from the same rank, increases the edge cut, and therefore the communication overhead. In order do prevent this overhead additional constraints are required. To prevent migrating inner Actors a mechanism that tracks the borders between ranks, as explained in Possible Approaches chapter is mandatory.

In order to successfully implement Territorial Expanse, information about the partitions (especially the borders/edge cuts between partitions) is necessary. In the current implementation, the job of partitioning the Actors is delegated to user. If the partitions are to be regarded during the migration (or when a re-partitioning is needed) then heavy communication with the user is mandatory.

This can be achieved by providing the Actorlib with the information on the partitions or with lazy-instancing, where the user defines all the Actors, their types and connections. After all the the definitions are provided the Actorlib can partition the Actors and create the instances with a call to the hypothetical `initiate()` method.

Since the library needs the use of a partitioning tool in both cases, it is feasible to leave partitioning to the Actorlib. The invade scenarios (where new ranks join the computation and
work has to be redistributed) or the territorial expanse (diffuse based work redistribution) approach can directly use the partitioning tool with the available information about the partitioning in their decisions.

7.2. Actor Stealing

Market implementation requires all Actors to participate in the migration process and ActorGraph is blocked during this process. It is possible to steal an actor without blocking the whole ActorGraph as explained in Possible Approaches. A Communicator and a Coordinator can be implemented in each rank, to track work asymmetry and to migrate Actors without blocking the ActorGraph. Although bulk sending with interrupts will be better on coarse work asymmetry, actor stealing is required to even out more fine grained imbalances.

Actor stealing approach requires more communication overhead and additional steps to the migration. It has to make sure that no LPC or RPC will be enlisted to Actor’s persona, and ensure that the persona has executed its all enlisted procedure calls. Whether the benefit of actor stealing will exceed the overhead induced by the additional requirements and extra communication overhead should be further studied.

7.3. Checkpointing

With longer running jobs it is likely that a hardware failure or an event that requires the halting of the job will occur, therefore it is beneficial to have checkpoints. Serializing an Actor over UPC++ is similar to sending an object over the wire with any other data interchange format. The functions for serialization can be used as reference implementations for save mechanism. Serializing the data to disk, will enable the continuation of the job at later time, in case of an interrupt or failure. Checkpointing should not introduce much overhead, except for the write operations, due to the low overhead induced by interrupt.
8. Improvements for the Library

The Actorlib and the added extension both may benefit from various quality-of-life upgrades, that help in debugging and increase readability in the code. Possible improvements that decrease the memory consumption, may ease the debugging process or can improve execution time are listed here.

8.1. Concepts

In the current state of Actorlib and Actorlib-Extended, various static assertions are performed to ensure that the input types are derived from Actor base class and to check whether the implementations can be used by the components as intended. Many assertions can be avoided with concepts, introduced with C++20. It would increase readability of the code and wipe out a possible source of bugs.

8.2. Dynamic Step Size Calculation

Unmodified run() method of the old Actorlib does not support pausing, therefore the maximum time step was calculated at the initialization of Pond and and remains constant throughout the simulation. With the implemented interrupt method, the maximum time step can be recalculated during interrupts.

8.3. Enforce a Finite State Machine instead of act() definition

The Actorlib requires solely the definition of the act() method, the Actorlib-Migration requires in addition the definitions of a upcxx::serialization struct and an in_actable_state() method. The method in_actable_state() returns true if the finite state machine is in a state where a call to act() would result in work being done (likely resulting in communication injection). Providing a framework for defining a final state machine would make the in_actable_state() method obsolete and the enforcement would simplify interaction with the Actor implementations.
8. Improvements for the Library

Existing libraries can be used instead of from-scratch implementations. For example the Meta State Machine library [30] of boost (also see [33]) can be used.

8.4. Integer Indexing

Each rank saves the names of every Actor instance in a map: \( D : std::string \rightarrow GlobalActorRef \). The duplication of the names in each rank can be avoided with an adaptation of a distributed hash map (e.g. [34]) as mentioned in the original paper [41]. But the memory consumed by the program can be further decreased with less work. Replacing the \( std::string \) indexing with \( int \), or \( uint16_t \) based indexing will require less memory. A minimal generator can be implemented with 2 static member variables which are used to assign a unique rank and an instance index to each Actor. The same can be applied for In/OutPorts of an Actor. A basic implementation could be implemented as follows:

```cpp
static uint16_t rankident;
static std::atomic<uint8_t> instanceident;

//...
Actor::rankident = upcxx::rank_me();
Actor::instanceident = 0;

//...
Actor::Actor(...){
    //to be called in the constructor for Actor
    this->identification.second = instanceident;
    instanceident++;
}
```

The rankident is required since a migrated actor needs to keep the identifier of its initial rank. It is safe to assume that the name of an In- or OutPort will be bigger than the size of \( uint8_t \) and the name of an Actor will be bigger than 3 bytes, the size of \( std::pair<\text{uint16_t},\text{uint8_t}> \). Int indexing will be more beneficial in the PortGraph component, since it saves the combinations: \( (\text{OutPort}, \text{OtherActor}, \text{InPort}) \) and \( (\text{OutPort}, \text{OtherActor}, \text{InPort}) \) to and from all neighbors. Saving the identification of Actors instead of their names will require minor changes in the most functions of the library.
8. Improvements for the Library

8.5. Warnings

Intel Compiler (see Appendix for the used version) has issues with `std::variant<T, T...>` therefore one has to be careful when changing any component that uses `std::variant<T, T...>`: the `ActorWrapper<T, Ts...>` and `MultipleActorAGraph<T, Ts...>` classes. Some workarounds for the following bugs were implemented:

Converting/Default Constructors

MultipleActorAGraph utilizes `std::variant` in order to wrap an Actor to another rank. Converting constructors (CMPLRIL0-32507) and default constructor usage (CMPLRLIBS-2742) result in bugs with the Intel C++ Compiler, rendering MultipleActorAGraph, therefore the applications making use of multiple Actor implementations impossible with Intel Compiler. The Pond application uses SingleActorAGraph, which consists of only one type of Actor, therefore does need `std::variant`, and can be compiled with the Intel C++ Compiler. The `MultipleActorAGraph` currently does not make use of converting constructors, and the class `ActorWrapper` is not instantiated with a default constructor. The future developer must be careful when meddling with `ActorWrapper` and the `MultipleActorAGraph`.

Templated Constructor of ActorWrapper

The `template<class T,class... Ts> MultipleActorAGraph<T,Ts...>` uses its template arguments to deduce the type of an Actor from the type-erased Actor pointer. A templated constructor of `ActorWrapper` of type `template<class Ins> ActorWrapper<T,Ts...>(Ins ins)` can’t be used, since it results in a recursive resolution situation. Further information section General Utilities Library in C++ standard [32]. Although the `template<class Ins> ActorWrapper<T, Ts...>(Ins ins)` constructor should not compile, it did compile with g++ (once again, see Appendix for the used version).
A. Sources

Here are the used software, compilers, libraries, as well as their versions listed. The testing and development was performed under Linux.

SWE (used by Pond): [https://github.com/TUM-I5/SWE](https://github.com/TUM-I5/SWE)

Actorlib and Pond: [https://bitbucket.org/apoeppl/actor-upcxx/commits/branch/master](https://bitbucket.org/apoeppl/actor-upcxx/commits/branch/master)


UPC++ (used version 2020.3.2) [https://bitbucket.org/berkeleylab/upcxx/wiki/Home](https://bitbucket.org/berkeleylab/upcxx/wiki/Home)

GCC version 9.3.0 (used mainly for testing during development)

Clang++ version 9.0.1 (used to check whether any GNU extension were used)

Valgrind version 3.15.0 (used to detect memory leaks)

Intel C++ Compiler version 19.0.5 (used for execution)

Intel MPI version 2019.7.217 (used only for job launch)

Metis version 5.1.0 (used for the initial distribution of Actors)

NetCDF version 4.7-hdf5 (used for file output)

Loaded modules for a default job launch. For more detail check the scripts present in the repository.

module load intel
module load intel-mpi/2019-intel
module load netcdf-hdf5-all/4.7_hdf5-1.10-intel19-serial
module load metis/5.1.0-intel19-i64-r64
B. Reproducing the Examples: Scripts

All the examples used in this work, and those running on 1 or 2 nodes are generated with the generator.sh script. The output directories are also generated with generator.sh. The executables are generated with the builder.sh script. Which also copies the executables to the right working directory (with some exceptions). The chonkergenarator.sh and chonkerssubmitter.sh has the same functionality respectively with generator.sh and submitter.sh but generate the output for nodes >2. The generation done separately because the output script changes accordingly to the cluster used where 1, 2 nodes use the cm2_tiny cluster where 3-31 nodes used cm2_std and the >31 used cm2_large.

The UPC++ can be compiled with compile-upcxx-intel.sh or compile-upcxx-gnu.sh which compile the UPC++ respectively with Intel compiler (with MPI wrapper for job launch) and with GNU C++ compiler.

Used scripts can be found under scripts in the repository.

The generator script is shown below. The function used to scale end time is omitted to preserve simplicity. The time scaling function takes the required time for completing work noted by 1 unit of work complexity (2000x2000-1) on 1 node as input, and scales it with the end-time of the job, the size of the job and divides with the dual logarithm of the number of nodes. The chonkergenerator script sets some environment variables differently and the output scripts differs in 3 lines, compared to the ones generated by generator script therefore the chonkergenerator is also omitted.

The generator script creates folders for given job configurations and corresponding folders in location saved in the $SCRATCH environment variable for the output files, in order to separately save the .out,.err and the output NetCDF files (under out). Further details can be found in the script code as comments.

```bash
#!/bin/bash

# job types, the executables should have names of type
# pond-x for every x in jobs array
jobs=(plain migration interrupt)

# grid sizes (one edge)
sizes=(2000 4000 8000)
```
#patch size (one edge) of an actors
patchsize=250

#number of ranks per node
corecountpernode=28

#number of nodes
nodecounts=(1 2)

#end-times for the simulation
endtimes=(4 8 16)

#base time required to calculate 2000x2000-1 on 1 node
basetime=600

#name of the group
workdir=expanse

create jobscripts
and give permissions
if [ -d jobscripts/ ]
then
 echo "jobscripts exists"
else
 echo "create jobscripts"
 mkdir jobscripts
 chmod -R 775 jobscripts
fi

#check if scratch variable is set
if [ -z "$SCRATCH" ]
then
 echo "scratch not set"
 SCRATCH=${PWD}
fi

#create ${workdir} in scratch
and give permissions
if [ -d ${SCRATCH}/${workdir} ]
then
 echo "${workdir} in scratch exists"
else
 mkdir ${SCRATCH}/${workdir}
 chmod -R 775 ${SCRATCH}/${workdir}
fi

#change directory to jobscripts
B. Reproducing the Examples: Scripts

# and generate the scripts

cd jobscripts

# generate folders for each task configuration
# a set of for loops that iterate through each array defined at the
# beginning of the progem (sizes,nodecount,endtime,jobtype)
for size in ${sizes[@]}
do
  for nodecount in ${nodecounts[@]}
do
    for endtime in ${endtimes[@]}
do
      for job in ${jobs[@]}
do

        # generate the job name
        # e.g: pond-plain-2-2000x2000-4
        str=pond-${job}-${nodecount}-${size}x${size}-${endtime}
        if [ -d ${str} ]
          then
            # overwrite old scripts, if exists
            rm -r ${str}
            mkdir ${str}
          else
            mkdir ${str}
          fi

        cd ${str}

        # set script name
        scriptname=pond-${job}.sh

        # calculate total number of ranks
        corecountforjob=$(( corecountpernode * nodecount ))

        # scale time for the job
        # the function is omitted due to simplicity
        limit=basetime
        convert_time

        # write the script
        # cat requires the ident to be 0
        cat <<<EOF >$scriptname
#!/bin/bash
#SBATCH -J ${str} #jobname
#SBATCH -o ${SCRATCH}/${workdir}/${str}/%x.%j.out
#SBATCH -e ${SCRATCH}/${workdir}/${str}/%x.%j.err
#SBATCH --mail-type=end,fail,timeout
#SBATCH --mail-user= <...> #write your own mail adress
# Wall clock limit:
#SBATCH --time=${hours}:${mins}:${secs} #times needs to be of form XX:YY:ZZ
#SBATCH --no-requeue #default values from lrz docu
#Setup of execution environment
#SBATCH --export=NONE
#SBATCH --get-user-env
#SBATCH --clusters=cm2_tiny #tiny cluster for tiny jobs
#SBATCH --nodes=${nodecount} #get node count
#SBATCH --ntasks-per-node=${corecountpernode} #get rank count per node
#module load slurm_setup #required for slurm job
#load requied modules for job execution
module load intel
module load intel-mpi/2019-intel
module load netcdf-hdf5-all/4.7_hdf5-1.10-intel19-serial
module load metis/5.1.0-intel19-i64-r64

#path to UPC++ installation
export UPCXX_INSTALL=~/upcxx-intel
#path to upcxx-run for job execution
export PATH=\$PATH:~/upcxx-intel/bin
#41 GB is the max number in mpp2 (both _inter and _tiny)
export GASNET_PHYSMEM_MAX='32 GB'

#for details refer to UPC++ user gude
#n ranks on N nodes, shared sgment size is 512 mb, script has the name pond-${job}
#grid sizes taken from the values currently active in loop
#10 times file output
#scenario 2 for scalable pool drop scenario
#output in scratch directory
#end time from the active variable in loop
upcxx-run -n ${corecountforjob} -N ${nodecount} -shared-heap 512MB ./pond-${job} \\ -x ${size} -y ${size} -p ${patchsize} -c 10 --scenario 2 \\ -o ${SCRATCH}/${workdir}/${str}/out.out -e ${endtime}
B. Reproducing the Examples: Scripts

# if script is done write a finished.txt file
# used to deduce if the job completed and was not timed out
cat > ${SCRATCH}/${workdir}/${str}/finished.txt

# end output
EOF

chmod 775 $scriptname
cd ..
done
done
done
done
go back to working dir and generate folders in scratch
cd..
echo ${pwd}
echo "done generating script folders"
echo "generate folders in scratch"

# same loop as the one above
for size in ${sizes[@]}
do
    for nodecount in ${nodecounts[@]}
do
        for endtime in ${endtimes[@]}
do
            for job in ${jobs[@]}
do
                # generate name
                str=pond-${job}-${nodecount}-${size}x${size}-${endtime}
                if [ -d ${SCRATCH}/${workdir}/${str} ]
                then
                    # do not delete the old output
                    # rm -r ${str}
                    echo "old results for "${SCRATCH}/${workdir}/${str}" exists, \
                    did not touch"
                else
                    # if the first time then generate the folder
                    mkdir ${SCRATCH}/${workdir}/${str}
Before the jobs are launched or the executables are compiled, the UPC++ has to be compiled. It can be done by call to a compile-upcxx-*sh. The details are commented in the script.

```bash
#!/usr/bin/env bash

# if the file does not exist download it from
# UPC++ website
echo download upc++...
FILE=upcxx-2020.3.0.tar.gz
if [ -f "FILE" ];
  echo "FILE exists"
else
  wget https://bitbucket.org/berkeleylab/upcxx/downloads/upcxx-2020.3.0.tar.gz
fi

# extract file
tar xvf upcxx-2020.3.0.tar.gz
# create upcxx-intel in compliance with the generator and build scripts
mkdir ~/upcxx-intel
cd upcxx-2020.3.0/

# load modules for compiling UPC+
module load intel
module load intel-mpi
module load netcdf-hdf5-all
module load metis/5.1.0-intel19-i64-r64
module load cmake/3.15.4

# start building upc++...
export UPATH=~/upcxx-intel

# required settings for the compilation
# mpi for job launch, intel compiler and default gasnet conduit is ibv
B. Reproducing the Examples: Scripts

./configure --prefix=$UPATH --with-cc=mpiicc --with-cxx=mpiicpc \ 
--with-mpi-cc=mpiicc --with-mpi-cxx=mpiicpc \ 
--with-default-network=ibv --enable-ibv

# make and make install
# make check with ibv requires launching the tests per hand
# therefore it should be done per hand in an interactive node
make -j16 all
make install

After generating the files and compiling the UPC++ the executables can be compiled. The builder script depends to a large extent on the required CMake configuration of the executable, there only the core builder script is to be found which compiles only one type of executable. The configuration required for all of the jobs is explained in detail after the builder script. The real script used to build executables uses a types array instead of an type environment variables and is hardcoded into the script. It compiles one executable for each entry in the types array and copy each executable to the folder which has the same type name. The behavior of the used script can be achieved with core script by calling it 3 times with different types.

#!/bin/bash

#script for building and copying right executables
#get path for UPC++ installation
export UPCXX_INSTALL=~/upcxx-intel
#/upcxx-intel

#load required modules
module load intel
module load intel-mpi/2019-intel
module load metis/5.1.0-intel19-i64-r64
module load cmake/3.15.4
module load netcdf-hdf5-all/4.7_hdf5-1.10-intel19-serial

#delete files from old build
make clean
rm CMakeCache.txt
rm -r CMakeFiles
rm cmake_install.cmake
rm Makefile
rm pond-$(type)

cmake . -DCMAKE_C_COMPILER=mpiicc -DCMAKE_CXX_COMPILER=mpiicpc \ 
-DCMAKE_PREFIX_PATH=$UPCXX_INSTALL -DINTERRUPT=OFF -DMIGRATION=OFF
make -j 16
mv pond pond-plain

#check for job scripts
if [ -d jobscripts ]
then
    echo "jobscripts already exists"
else
    echo "no sense if chonkerjobscripts are not generated"
    exit
fi

#copy to folder where the scripts were generated
cd jobscripts

#copy executables to already created files from the generator.sh
for i in pond-\${type}-*/
do
    echo $i
    cp ../pond-\${type} "$i"
done

After the executables are placed next to the scripts they can be submitted.

#!/bin/bash
maxjobs=50
counter=0
workdir=group2

cd jobscripts

#submit all job with the type \${type}
for i in pond-\${type}-*/
do
    #check if the script did finish
    file=${SCRATCH}/${workdir}/${i}/finished.txt
    if [ -f "$file" ]
    then
        #do not submit if it was finished
        echo "$i was already completed"
    else
        #stop if max job count for one user is reached
        ...
if [ ${counter} -le "50" ]
then
    #submit the job
    cd ${i}
    sbatch pond-${t}.sh
    counter=$(( counter + 1 ))
    cd ..
else
    echo "max number of submissions reached"
    exit
fi
fi

done

cd ..

echo "submitted ${counter} jobs"

Invade describes the invasion scenario from \( n \) to \( 2n \) nodes with random partitioning. Plain is the plain run (unmodified run), interrupt is the run method with timeout intervals (default is 128 seconds), migration-separate is the migration strategy (global random ordering) with separate waiting, migration-bulk is the migration strategy (global random ordering) with bulk waiting, msgchain is the message chain example (see \[5.13\] under section \[5.5\] Message Chains), where message are sent with various rank configurations. The last one, transport is the example where the serialization time difference between bulk and separate sending is analyzed (see \[5.3\] under Movable Fields \[5.1.1\]).

There are couple of CMake configurations that were added with Actorlib-Extended. They do not change the job type but add features to any job. PRINT enables a massive amount of print statements (debug information). HEAPARRAY might be required with big array sizes in Actors when compiled with the Intel C++ Compiler, BREAKDOWN enables tracking and printing of timestamps during migration phases (separate from PRINT enabled print statements) and DEBUGFLAG explicitly compiles with the -g in order to force linking the debug GASNet-Ex backend.

ON is for CMake option set to ON, OFF is for option set to OFF. _ is for do-not-care, the option can be ON or OFF without affecting the JobType. Some of the newly introduced variables are required only for use-cases (special test cases) and should be pruned before an release.
### B. Reproducing the Examples: Scripts

<table>
<thead>
<tr>
<th>JobType</th>
<th>Interrupt</th>
<th>Migration</th>
<th>Invade</th>
<th>Once</th>
<th>Old</th>
<th>TE</th>
<th>Randomness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invade</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>ON</td>
<td>ON</td>
</tr>
<tr>
<td>Plain</td>
<td>OFF</td>
<td>OFF</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>Interrupt</td>
<td>ON</td>
<td>OFF</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>Migration-Separate</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>_</td>
<td>OFF</td>
</tr>
<tr>
<td>Migration-Bulk</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>_</td>
<td>OFF</td>
</tr>
<tr>
<td>Plain-RP</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>MsgChain</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>Transport</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
</tbody>
</table>

Table B.1.: Required CMake config to get the desired JobType

Plain-RP requires the migration to be ON because the partitions have to be shuffled before starting the computation, requiring one migration phase and interrupt before the call to the run method. MsgChain and Transport are separate executables independent of Pond therefore they do not get affected by the CMake options. If migration is used alone then it describes Migration-Separate.
C. Work Complexity

A more visual explanation for the work complexity calculation. Doubling time can be considered as doubling the work that needs to be done. And doubling the area essentially means double the amount of cell updates.

Figure C.1.: Scaling of Work Complexity
D. Runtime 3D Coordinates

![3D graph showing runtime coordinates for jobs on 1 and 2 nodes for strategies plain and interrupt]

Figure D.1.: 3D coordinates for jobs on 1 and 2 nodes for strategies plain and interrupt
Figure D.2.: 3D coordinates for jobs on 1 and 2 nodes for strategies plain, interrupt and migration
# List of Figures

1.1. Global address space where each rank has allocated one local and one global integer ........................................... 4

2.1. Write call from Actor A1 to A2 ........................................... 7
2.2. Read call on Actor A2 ..................................................... 7
2.3. The finite state diagram describing the transition of an Actor to a state where it is allowed to call act() ..................................................... 8
2.4. The boolean tuple used in the Actor implementation to track the state of an Actor during the migration procedure described in 2.3 ..................................................... 8

3.1. Dependency graph for actor migration. Before an action \( v \) can be performed all actions \( \forall v' : (v,v') \) must be performed. Deletion of the lingering Actors is omitted. ..................................................... 11
3.2. Bulk sending vs Separate sending. Functions are simplified to contrast the differences ......................................................................................... 12
3.3. A sea of rank \( n \) actors break when the actor in the middle is migrated to the rank \( m \), creating an island consisting of one \( m \) actor and breaking the message chains ..................................................... 13
3.4. A2 communicates with an actor from rank \( m \) and from rank \( l \), therefore it can be migrated to rank \( m \) or \( l \) only ......................................................................................... 13
3.5. Example redistribution of the territory when 2 ranks join existing ranks ......................................................................................... 15

4.1. A Simple Graph of Actors ..................................................... 23
4.2. A simple ActorGraph where A0 sends data to A1 and A1 to A2 ..................................................... 30
4.3. A0 can connect to its neighbors only after the migration of A1 is completed and A1 is re-connected to the graph ......................................................................................... 30

5.1. Effect of having movable fields, where each rank sends \( i < \frac{n}{2} \) sends 250 stub-actors to the rank \( j = i + \frac{n}{2} \) ......................................................................................... 38
5.2. Difference between Bulk- and Separate waiting, functions are simplified to contrast the differences ......................................................................................... 39
5.3. Effects of Bulk-Waiting and Separate-Waiting on invade! scenario serialization. Serialization begins at 15, 150 seconds ......................................................................................... 40
5.4. (B)ulk vs. (S)eparate sending ......................................................................................... 41
5.5. Time Decomposition of the Migration Phase of bulk waiting strategy in the invade! scenario. From 1 node to 2 nodes ......................................................................................... 41
List of Figures

5.6. Time Decomposition of the migration phase with bulk waiting strategy in migration-run .............................................. 43
5.7. Overhead caused by Interrupt ................................................................. 44
5.8. Runtimes for pond with grid size 8000x8000, end time 16, patch size 250x250 with strategies plain | interrupt | migration on varying number of nodes ........................................ 45
5.10. Values plotted depend on size and end time are transformed into a 2D plane. The grid size of 2000x2000 with end time 1 corresponds to 1 unit of work complexity. Doubling the end time doubles the work-complexity, doubling the size quadruple the work-complexity ................................................. 46
5.11. Runtime Plain-Interrupt-Migration ......................................................... 47
5.13. Message Travel Time for Separate Cases. Measurements in ms. 1 → 1 → 1 = Each Actor is at the rank 1 (ChainSM). 1 → 1 → 1 = Second Actor is another rank 2, shared memory (RetourSM). 1 ⇒ 1 ⇒ 1 = Second Actor is at another rank 2, distributed memory (RetourDM). ............................................. 49
5.14. Invade $n \rightarrow 2n$ next to plain $2n$. Both with random partitioning of actors due to the missing implementation for territorial expanse. The simulation time is 1 second for the measurements above and 2 for the measurements below. .......... 51

C.1. Scaling of Work Complexity ................................................................. 70
D.1. 3D coordinates for jobs on 1 and 2 nodes for strategies plain and interrupt ................................................................. 71
D.2. 3D coordinates for jobs on 1 and 2 nodes for strategies plain, interrupt and migration ................................................................. 72
List of Tables

B.1. Required CMake config to get the desired JobType .................................. 69
Bibliography


[44] Luca Rinaldi et al. “Are Actors Suited for HPC on Multi-Cores?” In: ()


