DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Implementation of a Distributed Actor Library using GPI

Amartya Das Sharma

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Implementation of a Distributed Actor Library using GPI

Implementierung einer verteilten Aktorbibiothek mittel GPI

Author:	Amartya Das Sharma
Supervisor:	Prof. Dr. Michael Georg Bader
Advisor:	Alexander Pöppl, Philipp Samfaß
Submission Date:	15th February, 2021

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15th February, 2021 Amartya Das Sharma

Acknowledgments

Throughout the many months of working on this project, I have received a great amount of support and assistance from many people. I would like to take this opportunity to express my gratitude.

My sincerest thanks to my excellent advisors Philipp Samfaß and Alexander Pöppl. Without their weekly, often hour-long meetings, and their prompt help whenever I needed it, this project would have been a much more difficult endeavor. I have learned a lot from all my interactions with my advisors, and they have given me the necessary tools not only to finish my thesis, but to further enhance and enrich my career.

I would also like to thank my supervisor Prof. Dr. Michael Georg Bader, for his support and patience during this project. I greatly appreciate his help in both academic and administrative matters, and for all the time he has spent on my behalf.

The support staff at Leibniz-Rechenzentrum (LRZ), who house the cluster on which this project was run, have also been invaluable. Even during the difficult times when this work was done, they have steadily and unflinchingly provided support for whatever libraries I required.

To all my colleagues and professors, and all my friends and family, I am very grateful to the support they provide, both knowingly and unknowingly.

Abstract

In the field of high-performance computing, with its myriads of architecture, hardware and communication protocols, programs are often written to be highly specialized to extract maximal performance from the particular environment. Therefore, in-depth knowledge of the hardware and software tools is often required even for simple programs, a hurdle that may be daunting for newer and experienced programmers alike. In this thesis, we attempt to solve this problem by providing an efficient and versatile abstraction layer that is easy and fast to use. The *actor library* provided separates computation and communication routines and adapts to many HPC environments. The library uses the GPI library as the communication API, and aims to replicate close to native performance on various problems. We test the library with a tsunami simulation model, and we see that our implementation performs and scales comparably to Actor-UPC++, the original library that the work was ported from.

Contents

Acknowledgments **iii**

Abstract **iv**

1. Introduction **1**
1.1. Programming on Supercomputers 1
1.2. The Case for an Actor Library 2
1.3. Objectives 2
1.4. Outline 2

2. Literature Review **3**
2.1. Actor Library 3
2.2. GPI 4

3. Introduction to the Actor Library **5**
3.1. Expectations 5
3.2. Features of the Library 6
3.3. Illustrative Example 7

4. GPI **10**
4.1. GASPI-API 10
4.2. Groups 10
4.3. Segments 10
4.4. Queues 11
4.5. Asynchronous Communication 11
4.6. Notifications 12
4.7. Synchronous Communication 12
4.8. Interoperation with MPI 13
4.9. Advantages of GPI 14

5. Implementation **15**
5.1. The Components of the Library 15
5.1.1. Actor 15
5.1.2. InPort and OutPort 16
5.1.3. Channel 16
5.2. Overview of the Library 16

5.3. The Initialization Process . . . 17
5.3.1. Local Object Initialization . . . 17
5.3.2. Exchanging Actor IDs with *syncActors()* . . . 17
5.3.3. Connecting Actors . . . 17
5.4. Interaction of Components . . . 18
5.5. Finalizing Initialization . . . 19
5.5.1. Communicating Segments . . . 19
5.5.2. Lookup Table . . . 20
5.5.3. Linking Channels . . . 21
5.5.4. Creating Segments . . . 23
5.6. Sending Data . . . 23
5.7. Receiving data . . . 25
5.8. The Run Cycle . . . 25

6. Shallow Water Equations **28**
6.1. Background . . . 28
6.2. Adapting SWE in Pond . . . 28
6.3. Experimental Setup . . . 32
6.4. Results . . . 33
6.5. Computation and Communication . . . 37
6.6. Tracing Function Runtime . . . 39

7. Summary **40**
7.1. Outline of Work . . . 40
7.2. Results and Conclusions . . . 41
7.3. Future Work . . . 41

A. Code for the Primer Example **42**

B. Guide to Running Actor-GPI **49**
B.1. Hardware and Software Setup . . . 49
B.2. Compiling and running Actor-GPI . . . 49
B.3. Compiling and running Actor-UPC++ . . . 50

List of Figures **53**

List of Tables **54**

Glossary **55**

Acronyms **56**

Bibliography **57**

1. Introduction

Computing power, over the past few decades, has evolved leaps and bounds beyond what the pioneers of the field could have envisioned. But as Moore's Law starts failing due to the limits of the physics of the universe itself, advanced computing has to go in the direction of massive computing clusters and efficient parallelization of existing code. As the questions we ask our computers become more and more complex (natural disaster simulations, trajectory mapping of interstellar objects *etc.*) and the expectations for fast, accurate answers become higher and higher, programmers of the future will undoubtedly need to look towards High Performance Computing (HPC) to fulfill these growing demands.

1.1. Programming on Supercomputers

Earlier approaches to satisfy high-performance application requirements involved combining multiple processors onto the same memory device. But shared memory architectures suffered from issues like memory access contention from different processors, which only got worse as the number of processors increased exponentially.

Modern supercomputers employ a different approach to provide millions of processors in a scalable way. Multiple shared-memory multi-core processors called *nodes* are connected through high quality network fabric. Nodes maintain private memory spaces, and inter-node communication is only possible through specialized hardware or software abstractions. Nodes themselves, as stated, are comprised of many processors, and can run hundreds of threads on top of many physical and logical cores. Nodes themselves can be thought of as *shared memory systems* while a network of interconnected nodes can be called a *distributed memory system*.

In a shared memory system, some challenges faced are orchestrating threads to share processing load equally, and balancing memory accesses to prevent competition/contention for resources. Both unbalanced workloads and free competition for memory can cause bottlenecks that significantly throttle the program output.

Threads in distributed memory systems need to communicate data to one another. This can be done in many ways, which include accessing the memory of a remote node directly, or through some system of message-passing, which can be unidirectional or bidirectional. Communication is often slow in these cases so it is important to overlap computation and communication in an optimal way to reduce downtime. Remote Direct Memory Access (RDMA) technology allows direct access to memory in remote nodes without involving the computing resources of the remote node, hence it is a fast asynchronous method of communication. This thesis utilizes the GPI library, which uses RDMA by creating large memory areas in individual ranks from where data can be freely read and written by remote

ranks asynchronously.

1.2. The Case for an Actor Library

Programming on an HPC cluster provides unique challenges as described, obstacles that the average programmer may not be cognizant of. As we move towards requirements of higher performance, a stepping stone will be much appreciated by most users, who can then quickly begin implementing whatever solution they need, without having to worry about message passing protocols or load balancing across threads. Not having to worry about industry best practices or debugging memory access issues, a user would be able to quickly model and run their problem on whatever HPC hardware they desire. The *actor library* aims to be that stepping stone, the abstraction layer that allows users that are not parallel programming experts to scale their program up in a reliable manner.

The actor library, taking inspiration from Object Oriented Programming (OOP) models, requires the user to divide their solution into functional units called *actors*. Actors are black-box state machines with predefined behavior, and only interact with other actors by passing data across one-way *channels*. The expectation is that the user only needs to define the actors and their behavior, and their connections as a directed multigraph, and the library will take care of the execution of the graph by placing actors on different threads in the cluster and handling the communication across threads and nodes implicitly.

1.3. Objectives

This thesis aims to extend an existing actor library model written in UPC++ and re-implement it using GPI. As UPC++ and GPI are both RDMA models, they both allow high-speed asynchronous data exchange, which is important for actor libraries. We aim to provide a competitive implementation of the library, and test it on useful large-scale scientific simulations. The library should be easy to use and set up, and provide sufficient abstraction away from the actual communication layer.

1.4. Outline

In chapter 2 we summarize work related to the actor library and Global Address Space Programming Interface (GPI). Chapter 3 goes more into detail regarding the particular implementation of the actor library that we have used in this project, and how it answers all the expectations that a user may have. Chapter 4 provides a background for GPI and how it is used, limited to the scope of the implementation. We explain the final implementation in chapter 5, going over the intricacies of the library, its various components and how they work together. Then in chapter 6 we use the library in a shallow water wave simulation application, comparing its performance with the existing UPC++ library. Finally, in chapter 7 we discuss future work and final conclusions.

2. Literature Review

In this chapter, we summarize related work from which the present work takes inspiration. This will be followed by full chapters on the actor library and GPI, to further elaborate on their roles in the context of the current work.

2.1. Actor Library

The first actor model was proposed by Hewitt *et al.* [1] as a concept to help programming in artificial intelligence applications. This work was extended, among others, by Greif [2] who stated the importance of ordering of communications between actors and how the system is characterized by this ordering. Baker and Hewitt [3] postulated some laws for communication of actors, some of which were proven by Clinger [4]. Agha [5] provided the first formal notation of the model for concurrent programs, and discussed actors being independent units with predefined behavior that solve tasks, and that all computation that occurs within actors is a result of data that is communicated to them. Agha also introduced the concept of each actor possessing a *mail queue* where all incoming communication to the actor is held in the order in which they arrive.

The mail queue approach became a fundamental part of actor models henceforth, and their implementation seems to fall in one of two categories. In one, messages remain buffered in queues from where actors can fetch messages at their leisure in First In First Out (FIFO) fashion, which is preferable in asynchronous communication models. In the other, passing a message is viewed as an event and the receiving actor has to read the data immediately after receiving a signal to do so. Failure to retrieve the data means the data is lost and the actor cannot access it anymore. This is usually done in synchronous communication models.

In parallel systems, asynchronous models are used in a much greater capacity than synchronous ones. Examples include Erlang [6] and C++ Actor Framework [7][8]. Libraries like Akka [9] and Pykka [10] offer both asynchronous and synchronous (blocking) methods.

Our library is built taking most of its inspiration from the Unified Parallel C++ (UPC++) actor library created by Pöppl *et al.*[11] and its predecessor created by Roloff *et al.* [12] written in X10. Both UPC++ and X10 are Asynchronous Partitioned Global Address Space (APGAS) constructs [13][14]. This allowed for an easier porting of the library to GPI, which also follows the Partitioned Global Address Space (PGAS) model as stated earlier.

PGAS and APGAS [1] models work very well for actor models, because of the communication features they provide. In PGAS models, threads have local and global memories, and can access other threads' global memories to read and write data through specific pointers using

[1] An APGAS model is a PGAS model where the threads can be dynamically created by the programmer.

asynchronous methods. This ties in nicely with the concept of actors having states and memory inaccessible to other actors, and high-speed asynchronous communication using a mail queue approach to store messages until they are read.

ActorX10 defines the actor model it uses as an extension of the state machines of the FunState model of computation [15]. In their work, Roloff *et al.* formally define the model, introducing the actor graph as a dynamic multigraph distributed over threads, and actors that behave like state machines communicating in channels that act as FIFO queues. The publication also showcases two implementations of the model, and discusses mapping the actor problem onto PGAS models [12].

Pöppl *et al.* in their work [11], with a vision to scale up the ActorX10 model and remove bottlenecks, created the UPC++ actor library using the same actor formalism. As UPC++ uses the GASNet-EX middle layer, it works on fewer layers of abstraction than X10. The resulting library is tested alongside ActorX10 to solve a tsunami simulation scenario, where it outperformed X10 on all tests.

The current work aims to use similar actor formalism as the UPC++ actor library and fulfill similar expectations from its users, details of which are found in chapter 3.

2.2. GPI

The GPI library, formerly known as the Fraunhofer Virtual Machine (FVM) [16] was created to fully exploit the capabilities that an asynchronous communication library using an RDMA model would have. The library was created with the InfiniBand architecture in mind. In the original work, GPI was compared to the two popular standards of the time, Message Passing Interface (MPI) and Global Address-Space Networking (GASNet). GPI was shown to exhibit communication speeds comparable to raw InfiniBand communication despite being an abstraction layer on top of it, and performed comparable to or better than both MPI and GASNet for all message sizes [16].

Grünewald *et al.* in their later work [17] introduced the Global Address Space Programming Interface (GASPI) API, which restated and further developed the initial ideas of FVM, which was to utilize fast one-sided RDMA driven communication in a Partitioned Global Address Space. The GASPI API was implemented in GPI version 2, and has been used since.

Breitbart *et al.* in their work [18] compare GPI to MPI, and compare its feature set to fellow PGAS programming models: Unified Parallel C (UPC) and Coarray Fortran (CAF). They show that GPI performs much better than MPI for smaller messages, and conclude that compared to UPC and CAF, GPI allows more control over memory synchronization, but requires the programmer to explicitly manage it. Lena, in their work [19], introduce GPI2 for GPUs to make use of GPUDirect RDMA technology, and test its communication performance.

The latest specification for the GASPI API can be found at the website of the GASPI forum [20] [21]. A basic overview is given in chapter 4.

3. Introduction to the Actor Library

Having looked at the literature pertaining to actor models historically, in this chapter we redefine and explain the actor library as it is implemented in this work. We define the expectations that an actor library is required to meet and we go through an example application to illustrate its functions.

3.1. Expectations

The actor library, as discussed, is supposed to act as an abstraction layer, letting the programmer separate their problem into units with unique behaviors that communicate with each other. This gives rise to a Finite State Machine-like construction, where the units (actors) cannot interfere with the internal behavior of other units, but exist in a directed multigraph where they can communicate based on predetermined rules. Of course, a single actor does not provide any concurrency, and this coupled with the fact that an actor behaves like a black box with respect to other actors, means that it does not need to use locks to protect its own internal state. The actors share messages with each other until all the actors reach a final state. The actors themselves may reside on any thread in any node, and different assignments of actors to ranks should not change the programmer's design of the multigraph or the behavior of the actors with respect to each other.

Some of the advantages that this library provides:

- The programmer can divide their problem into an easy model that resembles Object Oriented Programming models, which means they only have to define behaviors and connections, significantly reducing the programmer's workload. No matter how the actors are distributed across ranks, the programmer can use the library functions and expect them to work as required.
- There is no need to handle communication issues across different nodes or processes explicitly, letting the designs stay hardware-agnostic and highly portable.
- Communication routines are handled in the background and are never exposed to the programmer, which help create code that is easier to debug and is more robust to scaling. The programmer is not expected to handle low-level communication, which means they never need to worry about data overlap, race conditions or mismatching parameters. Of course, due to the nature of the multigraph, race conditions may still be created, but the asynchronous one-sided communication enforced by the model mitigates it to a great extent.

- The actual communications can be done with any communication library, letting the programmer decide which library they want acting as a back-end based on the problem and the hardware. The designs stay back-end-agnostic and the programmer does not have to learn different libraries to test them.

3.2. Features of the Library

To address these expectations, the actor library implemented herein follows the model introduced by Pöppl [11] [12], which in turn is a simplification of the FunState formalism [15]. The following paragraphs explain the model as it was explained in their work.

A problem is to be solved by dividing it into functional units (actors) which communicate with each other only with data (via one-directional channels). These actors and channels together form a graph. Hence an *Actor Graph* G is comprised of actors and channels *i.e.* $G = (A, C)$ where A is the set of actors (the vertices of the graph) and C the set of channels (the edges of the graph).

A typical actor $a \in A$ is an independent unit, only able to interact with other actors by sending them data. An actor only acts when one of the channels connected to it has some data to act on, hence we can say an actor is *triggered* to act when its channels are updated (by some other actor). The actor then takes in the data on its incoming channels, performs some computations (that may change its internal state) and put the data on the relevant outgoing channels, upon which it removes a trigger from its internal triggers and goes dormant until it has a nonzero number of triggers again. If it sees that it has reached its final state, it marks itself as inactive and signals the graph G and its neighboring actors of its termination. If the actor is presently in a certain configuration with respect to its internal variables and behavior (summarized as its *state*), and it receives some data through its channels (or some data is removed from its channels), it is predictable what the new state of the actor will be after it has finished its computation and communication.

Hence, we can say an actor $a = (\mathrm{ID}, r, I, O, F, R) \in A$ is a tuple, consisting of

- a unique name ID
- a placement (*e.g.*, a rank) r
- a set of *input ports* I, to which incoming channels are connected to, and through which actors access incoming data (unique channels are assigned unique input ports)
- similarly, a set of *output ports* O, identical to input ports except for interfacing with outgoing channels instead
- a set of functions F. Some of these functions poll the connected ports for data availability and internal state to determine the next course of action, while leaving the data on the channels and the state untouched. These are *accessor* or *guard* functions. The other functions read the data, empty or fill the channels, and change the internal state of the actors. These are *mutator* or *action* functions.

Hence $F = (F_{guard}, F_{action})$ where F_{guard} is the set of guard functions and F_{action} the set of action functions.

- a finite state machine $R = (Q, q_0, T)$ which is activated whenever there is a change in the data in the channel associated with the input or output ports of the actor. The tuple consists of a set of states Q, the initial state q_0 and a set of transitions T, which again, consists of tuples $t = (q, k, f, q') \in T$, Here $q, q' \in Q$ are the initial and final states of the transition t respectively, k is the activation pattern and f the actions taken to change state. k may employ some functions from set F_{guard} to check for data and space availability on the ports. And lastly, $f \in F$, the set of functions available to the actor.

A typical channel $c_{n,t} \in C$ is a FIFO-style container (a queue), that can hold a maximum of n tokens of type t. The type t should at least be every primitive type available in C++, and additionally *std::vector*, although support for more containers is appreciated. Hence $C \subseteq A.I \times A.O$, where I and O are input and output ports respectively.

In Figure 3.1 we can see an example of two actors connected by a channel. Their list of functions and states are not shown for clarity purposes.

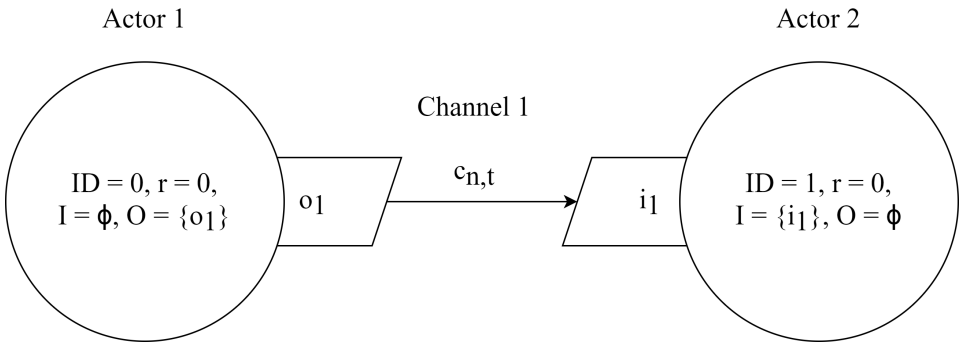


Figure 3.1.: Two actors and a channel.

3.3. Illustrative Example

Suppose we wanted to create a simple application that generates lists of primes. We could have three entities in such a system, namely a random number generator, a routine that generates a list of primes smaller than some given number, and a routine that prints a list of numbers. Using the actor library and three threads, this is how the problem could be tackled:

1. *Rank Initialization*
   The program creates three ranks to concurrently run on three separate cores. Each rank creates its own local ActorGraph object.

2. *Definitions*
   Three actors are defined, one that generates numbers (henceforth called *PrimeSource-Actor*), one that creates lists of primes (henceforth called *PrimeCalcActor*) and one that

prints lists (called *PrimePrintActor*). They are all defined on their own separate ranks (inheriting from an abstract Actor object) and are added to their local ActorGraph.

3. *Synchronization*
   The actors are now synchronized across all ranks, so that each ActorGraph object has knowledge of all the actors available on all the ranks.

4. *Data Connections*
   Now we define the one-way data channels between the actors. PrimeSourceActor creates an OutPort called *outSource*, PrimeCalcActor creates an InPort called *inCalc* and an OutPort called *outCalc*, and PrimePrintActor creates an InPort named *inPrinter*. Now, outSource and inCalc are connected, as are outCalc and inPrinter. The *ActorGraph::connectPorts* method is supposed to handle this on every rank. This is illustrated in Figure 3.2.
   This provides a necessary abstraction; it is not necessary to know which ranks are the actors local to when making connections. If the two actors are on separate ranks, the communication would have to be done using a communication library. However, this will get handled silently in the background not needing the programmer to define additional subroutines or worry about load balancing.
   The actors never directly interact with the channels, the ports act as an intermediary between channels and actors.

5. *Execution*
   Finally, at each rank, the ActorGraph object starts executing its actors. Each actor is marked active and triggered, and the actors' *act()* method is called continuously until it finishes its work (and marks itself inactive). As the PrimeSourceActor actor is the producer, it is initially manually triggered an extra number of times to get the process going. If five random numbers are required, the actor is triggered five times manually. When the producer actors are done producing, they mark themselves inactive after they send end signals to the connected consumer actors, who in turn also mark themselves as inactive upon receiving the signal.

6. *Termination*
   When all the actors find themselves in the inactive state, the ActorGraph objects break out of the loop and the program signals completion.

Example code for this illustration is given in Appendix A. The code is written in C++ and all imports are assumed to be done, and are removed for brevity.

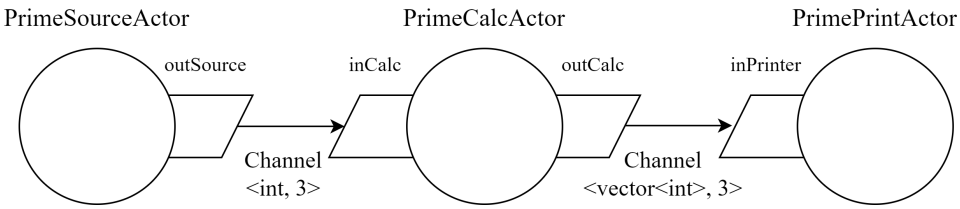


Figure 3.2.: Actor graph of Primes example.

4. GPI

In this chapter, we discuss the basics of GPI, and consequently the GASPI API it is based on. Since the API is not as generally well known as say, MPI or OpenMP, this chapter will hopefully serve as an adequate background to the implementation and testing portion of the work done. A more complete explanation can be found in the GASPI specification [21].

4.1. GASPI-API

Global Address Space Programming Interface (GASPI), as mentioned in chapter 2 is a Partitioned Global Address Space (PGAS) API. It provides C/C++ and Fortran with an API to let programmers take advantage of remote completion and asynchronous one-sided RDMA-driven communication in a Partitioned Global Address Space.

A unique advantage of GASPI is that it does not constrain the user to a fixed memory model. The user can define memory segments of arbitrary sizes and pin them to various memory areas of the computer, such as non-volatile RAM or video memory in a GPU, from where data can be freely read or written. It offers great flexibility in defining global memory areas and make these areas accessible across all ranks.

GPI implements the GASPI API in its latest releases.

4.2. Groups

If only a subset of GPI processes need to perform some particular collective task, GPI allows the user to define a *group* that comprises these processes. The user can then use group collective functions (such as barriers) on a group of processes, without touching the other running processes. Processes can be part of more than one group. All processes are part of the default group `gaspi_group_all`.

4.3. Segments

Global memory spaces in GPI are available in the form of *segments* which are defined by individual ranks. Each rank may have a different number of segments, and read from and write to any other ranks' segments. They offer a unique way of mapping a myriad of hardware memory segments to the software layer.

Segments are created on a local rank with the function `gaspi_segment_alloc`, which allocates required space on the node the process is running on. It also binds the segment to a unique numerical ID. To make the segment visible to other ranks, the function

`gaspi_segment_register` is called. GPI makes the `gaspi_segment_create` function available to combine these two processes. Existing memory allocated areas can be turned into new segments, and segments can be deleted if not needed.

When accessing a local segment, the rank needs a pointer to the beginning of the segment, using which it can write to/read from the memory area it needs. To access a remote segment, the rank needs the remote segment rank and the ID of the segment. Then it can copy to or from the segment, using a segment of its own. All data to be exchanged with a remote rank has to go through a local segment.

GPI limits the number of permissible segments on each rank, by limiting the number of segment IDs available. However, the segments can be very large. Hence, users are encouraged to use fewer segments that are larger in size, as segment creation is time-consuming.

4.4. Queues

All asynchronous communication in GPI is handled by using *queues*. All read/write requests are posted to individual queues that have unique IDs, where they are executed asynchronously. This helps in extremely fast communication across ranks and considerable overlap of computation and communication. Queues can be created and destroyed, and similar to segments, a maximum number of queues is enforced by GPI by limiting the number of permissible IDs.

Requests posted to queues are not guaranteed to be executed in the same order. Multiple ranks may post requests to the same queue. It is guaranteed that all requests in all queues will be handled eventually *i.e.* no request will be delayed indefinitely. Queues are not flushed automatically, so it is recommended to do so periodically. Queues have fixed capacity and can overflow, hence checking for free space before posting a request is important.

Calling `gaspi_wait` on a queue in a rank blocks execution and waits for local completion of all requests in a queue. After execution, all local buffers related to the communication requests/operations in the queue are guaranteed to be valid. However, since `gaspi_wait` is a blocking operation which also prevents any other rank from accessing the particular queue that is being flushed, it should not be used very often, as otherwise communication/computation overlap cannot be achieved.

4.5. Asynchronous Communication

The basic asynchronous read and write requests are `gaspi_read` and `gaspi_write`. The read operation requires the following parameters:

- a local segment ID where to read the data into, and the offset in the local segment where the read data will start.
- the remote rank where the data resides.
- a remote segment ID to read the data from, and the offset at which the data starts.
- the number of bytes to read.

- the queue ID where the request will be posted.
- a maximum time to wait until the request completes (exceeding which, the operation returns `GASPI_TIMEOUT`).

The write operation takes identical parameters, only the flow of data is from the local to the remote segment. Therefore, to write data to a remote segment, the process first copies the data into a local segment, and then sends a write request to copy the data over, from segment to segment.

4.6. Notifications

Since there is no way of verifying completion of remote communication requests at a local level, GPI introduces the concept of *notifications*. After posting a request, a process may post a notification using `gaspi_notify` to a queue. Every segment has a number of notification IDs attached to it, and the process may update some notification value of a remote segment. The remote segment can enter a blocking routine using `gaspi_notify_waitsome`, continuously checking the notification IDs of a segment until it sees that one of the IDs have been changed to a nonzero value.

Even though ordering of requests on a queue is not guaranteed, it is assured that all local requests posted to a queue will be placed before the notification is placed, *i.e.*, when the notification is received, all previous local operations are complete.

Figure 4.1 shows the notification system in action. Process rank 2 is expecting some incoming value from process rank 1, which may be needed in some future computation. However, it has no way of knowing whether the data has been sent or not. The rank can wait for a notification from the sending process, which will verify the integrity of the data. Eventually, after writing the data, process 1 sends a notification using `gaspi_notify` to process 2. Process 2 can block execution till the notification is received, and upon receiving the notification, may send one back. Once both notifications are sent and received, both processes know that the data has been successfully send and received. The data may become available before the notification reaches process 2, but checking the notification is how process 2 can be sure of the previous request's completion.

4.7. Synchronous Communication

In order to communicate synchronously, GPI provides `gaspi_passive_send` for writing and `gaspi_passive_receive` for reading. The sending thread must call the send function, and the receiving thread must call the receiving function. If one is called before the other, one blocks its local thread till the other is called. These functions read and write data from segments, and do not use the queue.

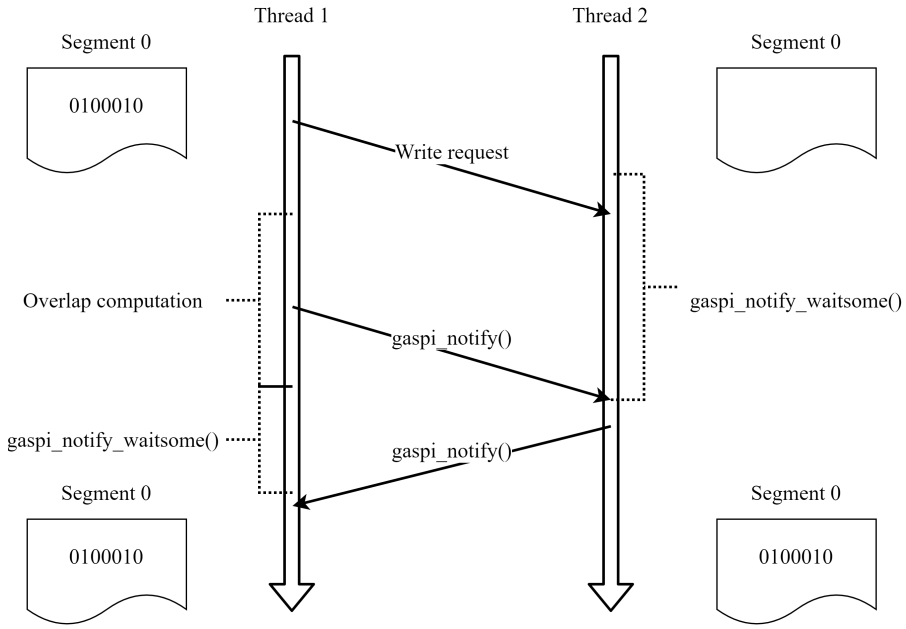


Figure 4.1.: GPI asynchronous communication mechanism.

4.8. Interoperation with MPI

To facilitate slow migration of existing MPI code and infrastructure to GPI, GPI allows programs to be embedded in MPI code. All GPI code can be run within MPI processes, which means that the user can use MPI to spawn processes and distribute tasks, while the actual code runs using GPI communicators. The supported program layout is shown in Code Snippet 4.1:

Code Snippet 4.1.: Overlapping MPI with GPI

```
1  main()
2  {
3      MPI_Init();
4      // start and finish all MPI communication; no GPI
5      MPI_Barrier();
6
7      gaspi_proc_init();
8      while(!done)
9      {
10         //start and finish all GPI communication; no MPI
11         gaspi_barrier();
12
13         //start and finish all MPI communication; no GPI;
```

```
14          MPI_Barrier();
15      }
16      gaspi_proc_term();
17
18      //can still do MPI work but no GPI
19      MPI_Finalize();
20  }
```

This also allows execution with `mpiexec` in case the system does not allow the GPI library's own `gaspi_run`.

4.9. Advantages of GPI

With the features of both the actor library and GPI in mind, we can now state our arguments for why GPI is uniquely suited for an actor library implementation.

- Large segments of remotely and asynchronously accessible memory allow actors to freely read from and write to each other, without worrying about handshakes or further overhead while communicating. Metadata and commands can similarly be exchanged between graphs residing on different processes.
- Having multiple queues and multiple segments allow for effective separation of concerns while communicating different types of data or metadata. For example, one segment may be set up to send data offsets, and one may be set up to handle actor triggers.
- Since segments are accessed by offset, actors can maintain distinct data areas in the same segment without worrying about collision. This ensures that actors never have to check for intended recipient of data and hence overhead is avoided.
- Interoperability with MPI allows for higher portability on a larger selection of hardware.

Keeping these advantages in mind, we can now look to implementing the Actor-GPI library.

5. Implementation

Keeping the details of the GASPI API (and consequently the GPI library) and the expectations of an actor library in mind, in this chapter we discuss the details of the implementation of the actor library using GPI.

5.1. The Components of the Library

In chapter 3, we have already outlined the components to be expected in the library. Here we discuss the implementation-specific details.

5.1.1. Actor

The virtual actor class, in this implementation, requires a unique global ID to differentiate itself from the other actors in other ranks. This ID can be explicitly added or automatically generated. To help maintain uniqueness, the automatic ID generation uses the formula presented in Code Snippet 5.1, where *procNo* refers to the rank where the actor was initialized, and *actNo* refers to a serial number attached to the actor. Therefore, ensuring that serial numbers are unique in a rank is enough to ensure unique global actor IDs.

Code Snippet 5.1.: Code for calculating actor IDs

```
// functions to encode and decode global actor IDs
uint64_t Actor::encodeGlobID(uint64_t procNo, uint64_t actNo) //static
{
        return (procNo << 20) | actNo;
}
std::pair<uint64_t,uint64_t> Actor::decodeGlobID(uint64_t inpGlobId) //static
{
        uint64_t procNo = inpGlobId >> 20;
        uint64_t actNo = inpGlobId & ((1 << 20) - 1);
        return (std::make_pair(procNo, actNo));
}
```

5.1.2. InPort and OutPort

InPort and OutPort objects have a label (unique within the actor's scope) and a reference to a Channel object. The Port objects have no explicit functionality of their own; they merely call the relevant functions of the Channel object. They serve as a convenient wrapper to the Channel objects.

5.1.3. Channel

Channels connecting two local rank actors (actors declared in the local rank) and connecting a local and a remote actor (actors declared in remote ranks) have wildly different functionalities, hence in the implementation we have two different classes, *LocalChannel* and *RemoteChannel*.

LocalChannel implements a simple std::queue of the specified type, storing data and passing it on as requested.

RemoteChannel uses GPI communication functions to send and receive data from remote ranks.

The specifics of the communication methods are discussed in the relevant sections in this chapter.

5.2. Overview of the Library

The actor library, at minimum, expects an actor implementation (a class inheriting from the actor class, implementing the *run()* method) and a file with a *main()* method, which will do the following:

- Initialize the actor graph at every rank.
- Create actors at ranks as required.
- Add local actors to local graph.
- Call the *syncActors()* function to allow the graphs to exchange actor IDs.
- Define connections between actors using their global IDs at every rank using the *connectPorts()* function.
- Call the *finalizeInitialization()* function to update all actors and channels after all connections are made, to initialize required segments.
- Run the graph, with the *run()* function.

An overview of the process can be seen in Figure 5.1.

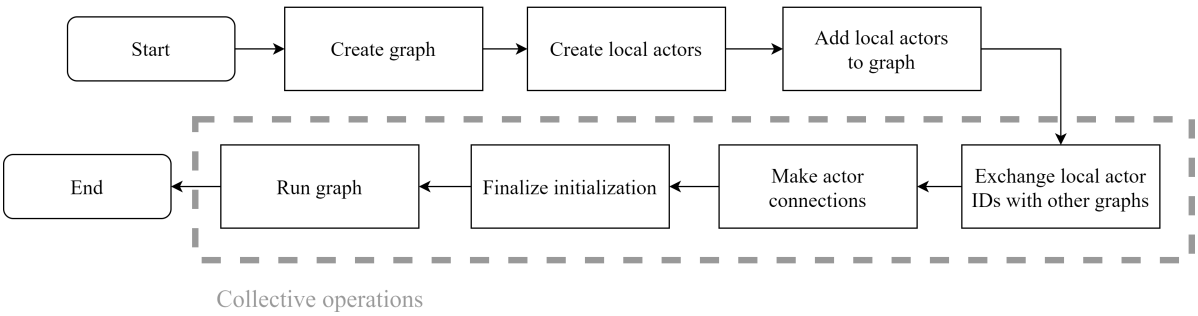


Figure 5.1.: Overview of a run of the library.

5.3. The Initialization Process

The following subsections go through the initialization process step by step, which includes all the steps in Figure 5.1 prior to the "Run graph" step.

5.3.1. Local Object Initialization

Each rank of the program requires its own instantiation of the actor graph. The actor graph at each rank stores references to all local actors. The local actor objects, once created, need to be added to the local actor graph with the *addActor()* function.

5.3.2. Exchanging Actor IDs with *syncActors()*

Once all the local actors are created and added to the local graph, the member function *syncActors()* of the graph is called, which does the following:

- Creates temporary segments.
- Sends its own actors' global IDs.
- Reads actor IDs from other graphs from other ranks.
- Deletes temporary segments.

At the end of this step, every graph of every rank has all the IDs of the actors created in various ranks. This allows every graph to have full information about interacting ranks when actors are connected in the next step of the process.

5.3.3. Connecting Actors

Now in every rank, the full graph is built by connecting actors to each other with directional, labeled edges. The *connectPorts()* function is called as shown in Code Snippet 5.2. The function takes 4 arguments and 2 template parameters. The arguments denote the two connecting actors represented by their global IDs (connected in order of the flow of data), along with

labels for the new ports they'll be connected by. Here, *LABEL1* denotes the output port (outport), and *LABEL2* is the corresponding input port (inport). The two template parameters signify the type of data that will be passed in the connection, and the maximum number of data packets to allow in the channel (the queue size).

Code Snippet 5.2.: Linking actors with ports

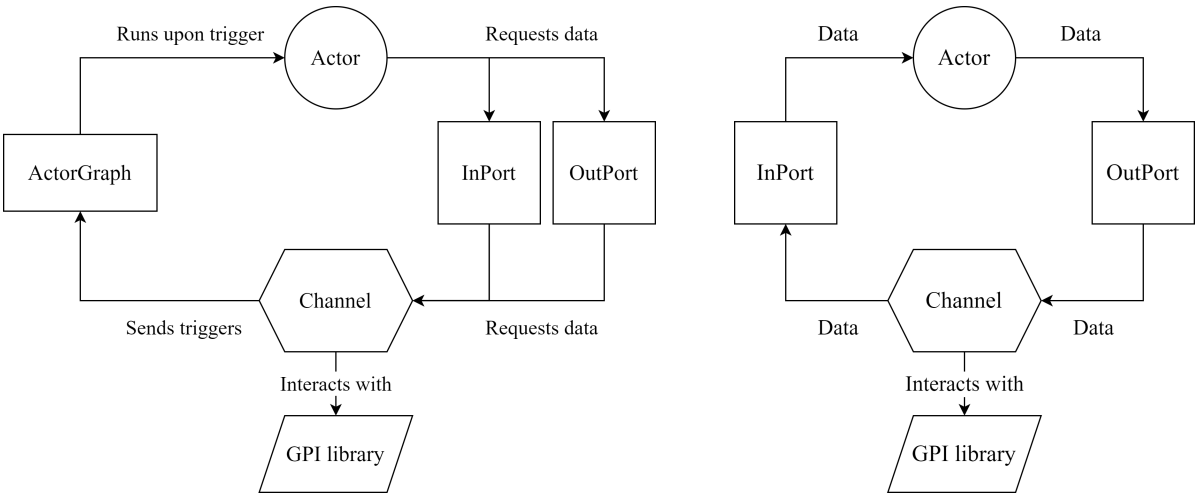
```
actorGraph.connectPorts<double, 1>
    (Actor::encodeGlobID(1,0),"LABEL1",
    Actor::encodeGlobID(2,0), "LABEL2");
```

Every successful connection, as discussed in chapter 3, results in two actors being connected, by means of an inport, an outport and a channel. Hence, when the function is called, it takes the following steps. Note that a local actor, in future contexts, is an actor which was declared in an actor graph's local rank, and all other actors are remote actors, for which the graph has the IDs but not the reference.

- The two actor IDs and port labels are concatenated to form a channel ID. This is stored in a list of all channels, as a record of all the connections made.
- Next, the actor IDs are scrutinized. The lists of local and remote actors are used to match the IDs.
  - If both actors are local, an inport and outport object are assigned to the receiver and the sender actor respectively. Then a local channel object is created, and the reference is passed to both the port objects,
  - If one actor is local while the connecting actor is remote, an inport or outport object is created, depending on whether the local actor is a sender or a receiver. Then a remote channel object is created, and the actor, port and channel are linked. This channel needs to be linked to the corresponding channel in the remote rank where the remote actor is defined. This linking step is handled by the future *finishInitialization()* function.
  - If both actors are remote actors, no further steps are taken.
- If the previous steps are successful, the connection is complete.

5.4. Interaction of Components

After the graph is initialized, the components react with each other as summarized in Figure 5.2a and the data flow occurs as summarized in Figure 5.2b. Here, a brief overview is presented.



(a) Control flow of components.

(b) Data flow between components

Figure 5.2.: How the actor library components interact with each other.

- The graph is responsible for manually triggering actors which have received triggers, and running all triggered actors. These triggers are received from channels. Additionally, actors may trigger themselves, and the user may specify a starting number of triggers.
- The actor is run by the graph when it has triggers left and has not concluded its work. It only interacts with its inports and outports, sending and receiving data.
- Port objects only call relevant methods in the linked channel object, and return data to the actor as required.
- Channel objects interact with ports, sending and receiving data from the parent actor. Local channel objects are shared between ports, and remote channel objects interact with the GPI functions to send their data to remote ranks.
- As stated by the actor library requirements, the graph takes no part in the data exchange, and only runs actors based on triggers.

5.5. Finalizing Initialization

After connections are made, two steps still need to be taken before the graph can be run. The communication segments need to be set up, and channels on different ranks have to be linked up.

5.5.1. Communicating Segments

As shown in Figure 5.2, there are multiple communication avenues between the components of the library. Each of these avenues require a separate GPI segment, to prevent overlap. Here,

to explain the purposes of the segments better, all the segments are classified into one of two types: **data segments** and **command segments**. Every rank has to have all of these segments available.

The implementation uses three data segments, *i.e.*, segments that communicate data and metadata between ranks. These segments are used exclusively by channels to read from and write to. They work as follows:

Datasize segment stores the total size of the data being transferred in bytes. The remote channel is expected to read from this segment first to allocate space on the remote rank.

Databank segment stores the actual data to be transferred. The entire size of the data bank is user-defined. The data bank is divided into blocks, and channels can reserve some number of consecutive blocks depending on the size of data to be sent. The block size is also user defined. For example, if the block size is 1 kB and a channel needs to send 3.5 kB of data, the channel reserves four consecutive blocks from its local databank. This space will be cleared and the reservation removed after the data is read by the remote rank.

Offset segment contains the starting offset at which the channel has reserved databank blocks. The expectation is that the remote channel will read from this segment for an offset, and then read directly from that offset from the databank segment, for some number of blocks, which can be calculated by reading the value from the datasize segment.

In addition, we have two command segments, segments which also store data but the data is used to perform some function at the remote rank. These segments are written into by channels and read from by actor graph objects. The graph object then performs some actions depending on which segment the data comes from. Brief descriptions follow.

Clear segment is written to by a remote channel after *reading* data. It contains the offset that was previously read from the offset segment. Seeing that this offset's data has become stale, the graph then clears the databank and marks it empty.

Trigger segment is written to by a remote channel after *writing* data. It contains the global actor ID of the local actor the channel is connected to. The graph, upon seeing the ID, triggers the corresponding actor.

In addition to these, every rank has a **cache segment**. This segment is important as data can only be read from a segment into another segment. The cache segment is used multiple times while communicating, to store data temporarily before being written to another segment.

5.5.2. Lookup Table

How do the channels reserve multiple blocks of the databank segment without collision or overlap, and how does the graph know to delete multiple blocks' worth of reservation when it is sent a clear command? This is managed by a static, public array at every rank which

has elements equaling the number of data blocks at every rank. Whenever a channel needs to reserve a block, it looks through the lookup table for an empty block (or a number of consecutive empty blocks). The lookup table is updated every time a block is reserved or cleared. While finalizing initialization, the lookup table is reset.

5.5.3. Linking Channels

Since all the channels will be writing to shared segment spaces, it is imperative to ensure that channels are assigned areas to write to prevent space collisions and mismatch of data. Furthermore, remote channels need to know where to read from without continued prompting from local channels, since otherwise the communication overhead will be too large. The idea thus is for a channel to write to the same offset of all the segments (barring the databank segment, which is reserved independently as shown before) every time, so that two channels never compete for the same offset in a segment.

The way this implementation makes sure those requirements are fulfilled is by assigning a unique integer ID to every channel. Since every rank has the full list of connections, and since each channel has a unique ID from concatenating the connecting actor IDs and port labels, these strings can be sorted to easily get a list of integer IDs for each channel. The result of the sort is identical on every rank; hence this process can be done independently at all ranks.

After sorting the string IDs, the channel list is iterated through. Channels that join two local actors (in any rank) are ignored. A running total of the number of incoming channels and outgoing channels is maintained. Every channel is assigned two integers: a data offset, which is the sorted index of the channel (when sorted among all outgoing channels) at the source rank, and a trigger offset, the sorted index of the channel (when sorted among all incoming channels) at the destination rank. These IDs are guaranteed to be unique and the same for the same channel at different ranks, hence if a channel in the local rank writes to its data offset, the corresponding linked channel at the remote rank can be assured to find the same data at the data offset, even though this offset was independently calculated at both ranks. The operation is shown in Code Snippet 5.3.

Code Snippet 5.3.: Channel offset calculation

```
// channel list is assumed to be sorted here
// noOfOutgoingChannels is an array of length = number of ranks
// at any index we can find the number of outgoing channels from that rank
// same of noOfIncomingChannels
for(int i = 0; i < remoteChannelList.size(); i++)
{
    uint64_t srcRank = Actor::decodeGlobID(remoteChannelList[i]->srcID).first;
    uint64_t dstRank = Actor::decodeGlobID(remoteChannelList[i]->dstID).first;
    if(srcRank == dstRank) //skip local connections
            continue;
    dataOffset = noOfOutgoingChannels[srcRank];
```

```
12      triggerOffset = noOfIncomingChannels[dstRank];
13
14      noOfOutgoingChannels[srcRank]++;
15      noOfIncomingChannels[dstRank]++;
16
17      remoteChannelList[i]->fixedDataOffset = dataOffset;
18      remoteChannelList[i]->fixedTriggerOffset = triggerOffset;
19  }
```

An example graph is shown in Figure 5.3 with the lines between actors denoting the channels, with their IDs written on the arrow. Inter-rank actor channels are removed for brevity. Table 5.1 shows the generated offsets for each channel. For example, when channel 2130 is sorted with all channels that leave from rank 2, it's the third channel, hence it gets data offset 3. Similarly, when it is sorted with all channels that enter rank 3, it's the second channel, hence it gets trigger offset 2.

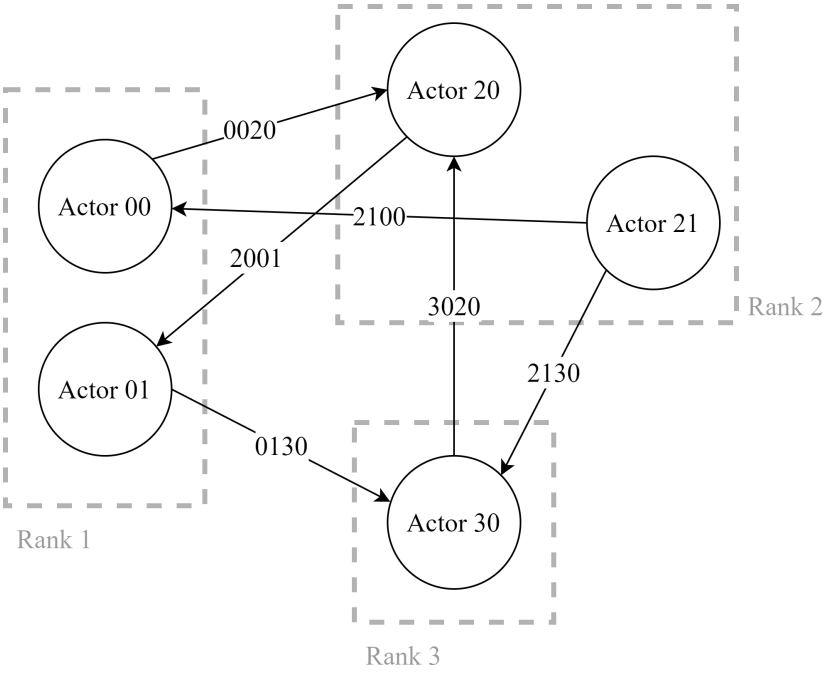


Figure 5.3.: Example graph structure to explain channel offsets.

Because sorting and performing these calculations results in identical tables at reach rank, the two instances of channel 2130 (one at rank 2, one at rank 3) never need to communicate to establish a link. Channel 2130 at rank 2 can write to the offset segment at its data offset, and channel 2130 at rank 3 can read the data from the same offset, without any setup required.

Table 5.1.: Offsets generated for channels in the example.

Channel ID	Data offset	Trigger offset
0020	1	1
0130	2	1
2001	1	1
2100	2	2
2130	3	2
3020	1	2

5.5.4. Creating Segments

As the last step of finishing initialization, the 6 segments at every rank are created. The sizes of the offset segment, datasize segment and clear segment are at least the size of the number of outgoing channels times the queue sizes of the connections, again to ensure that channels have a space in the segment without overlap. The size of the trigger segment is at least the size of the number of incoming channels times the queue sizes of the connections, for similar reasons. The databank size is the number of data blocks per rank times the block size, and the cache segment is at least the size of one datablock.

After the segments are created, every channel is given pointers to the start of the segments. All segments are reset to a default value, and initialization is complete.

5.6. Sending Data

With the setup complete, and with all channels having their offsets, communication becomes very easy to execute. If two actors on the same rank need to communicate, the channel, upon receiving some data from the outport, simply adds it to an STL queue data structure.

If a channel needs to send data to its linked channel in another rank, it follows a series of steps explained here, and summarized in Figure 5.4.

- The channel receives the data from the connected outport, and calculates how many blocks it needs to reserve to send the data.
- The channel then uses the static lookup table to reserve that many blocks of data.
- Then the offset obtained from the lookup table is taken, and written into the local rank's offset segment, at the current queue number plus data offset of channel.
- The data is written into the reserved blocks, of the local rank's databank segment.
- Size of data in bytes is written to the local rank's datasize segment, at the channel's data offset plus current queue number.

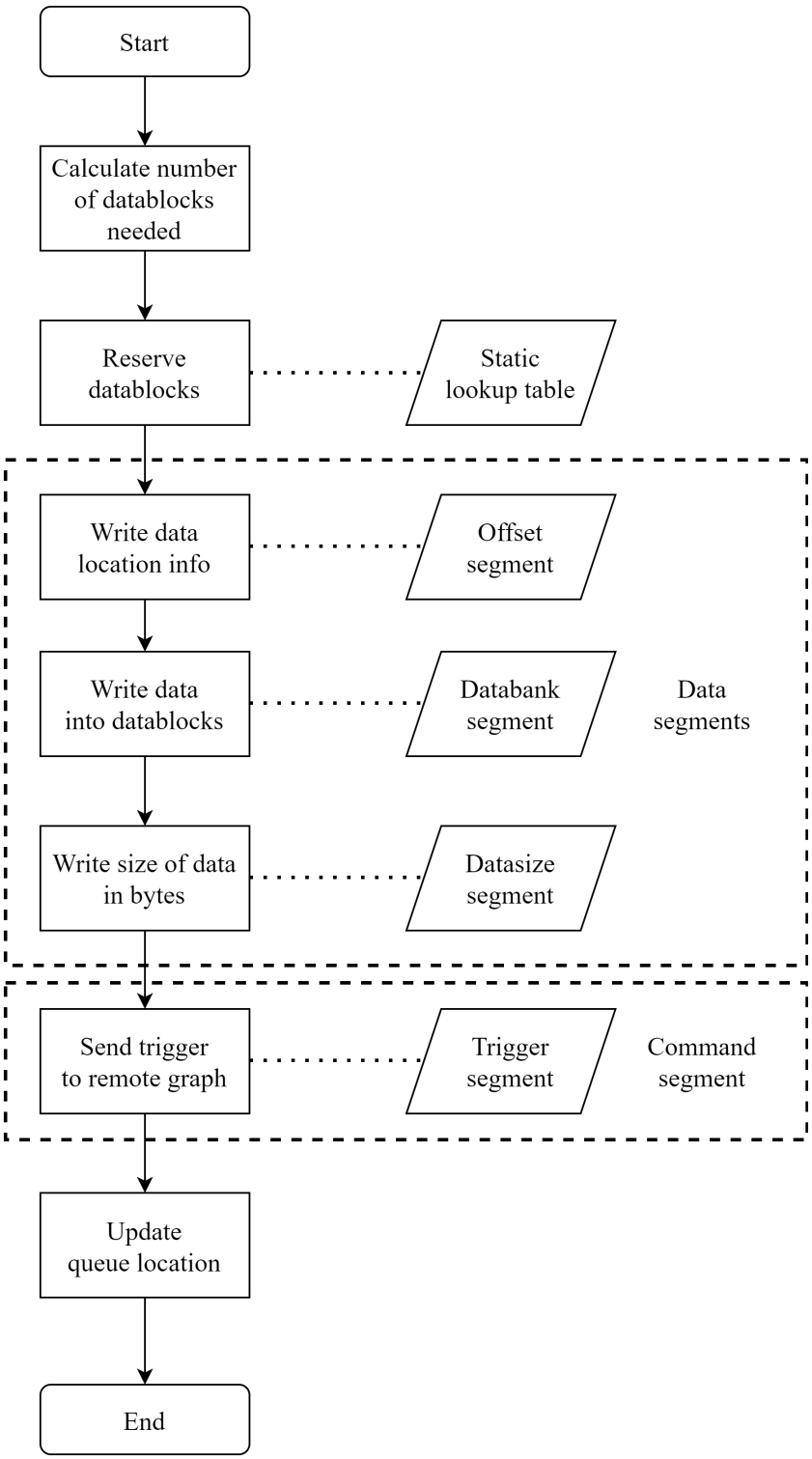


Figure 5.4.: Sending data to remote rank.

- Finally, the channel sends a trigger into the trigger segment of the remote rank, and increments its queue counter. To write the trigger, it has to use the cache segment to store the global ID of the remote actor before copying it to remote segment. The cache is cleared afterwards.

Channels in a local-local actor connection write a trigger into an STL queue maintained by the graph, exclusively to hold such triggers, separate from the trigger segment.

5.7. Receiving data

Receiving is a very similar process to sending. If the two actors are on the same rank, the channel upon receiving a data read request from the inport, pops the data from the STL queue data structure and sends it forward.

When a channel needs to receive data from its linked channel in another rank, the following procedure is followed, and summarized in Figure 5.5.

- The channel, having received a read request from the connected inport, first reads from the offset segment at the remote rank (where its linked channel is defined). It reads the data stored at the data offset, plus the current queue location. The data is read into the cache segment.
- Having the databank offset, it now reads from the remote datasize bank, at the data offset plus queue location, to find the total size of the data. This also goes into the cache segment. It calculates how many read requests it has to process, since one request can read a whole block.
- The channel then executes successive read requests from the remote databank segment. Every request is read into the cache segment, and the cache is emptied into a local STL container before being used for reading again.
- After the required number of bytes are read, the channel then writes a clear command to the clear segment at the remote rank. Again, the cache segment is used. The databank offset (obtained from the offset segment at step 1) is written to the cache and copied into the remote segment.
- Finally, the queue counter is incremented and the channel sends the data to the requesting inport.

5.8. The Run Cycle

Finally, in this section, we talk about the run cycle of the actor graph. The run cycle is initiated in the *main()* method of the application after the setup is finished.

- Firstly, the graph checks whether any local actors have work left to do. When an actor has run sufficient times and its operations are over, it sets a flag denoting so. The graph checks for this flag. If all actors are finished, the run cycle is over and the loop is broken.

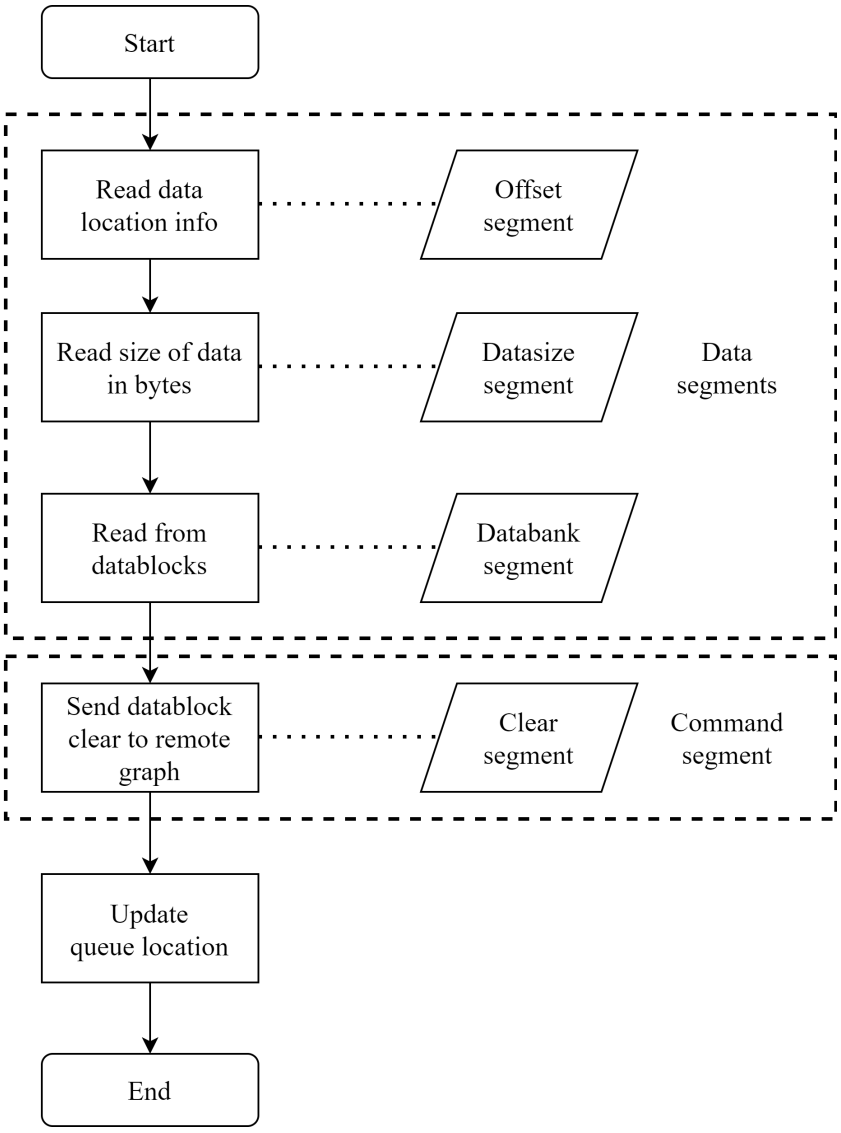


Figure 5.5.: Receiving data from remote rank.

- Otherwise, the graph first iterates through the trigger segment and local trigger queue, triggering all the actors that need to be run.
- Then the graph iterates through the local actors, running all actors with nonzero number of triggers. It waits for an actor to finish working before running the next actor.
- Finally, the actor accesses the clear segment, and iterates through it. When it finds an offset to be cleared, it removes the reservation at that offset from the lookup table. After this, the run cycle starts again from the beginning.

An overview of the cycle can be seen in Figure 5.6.

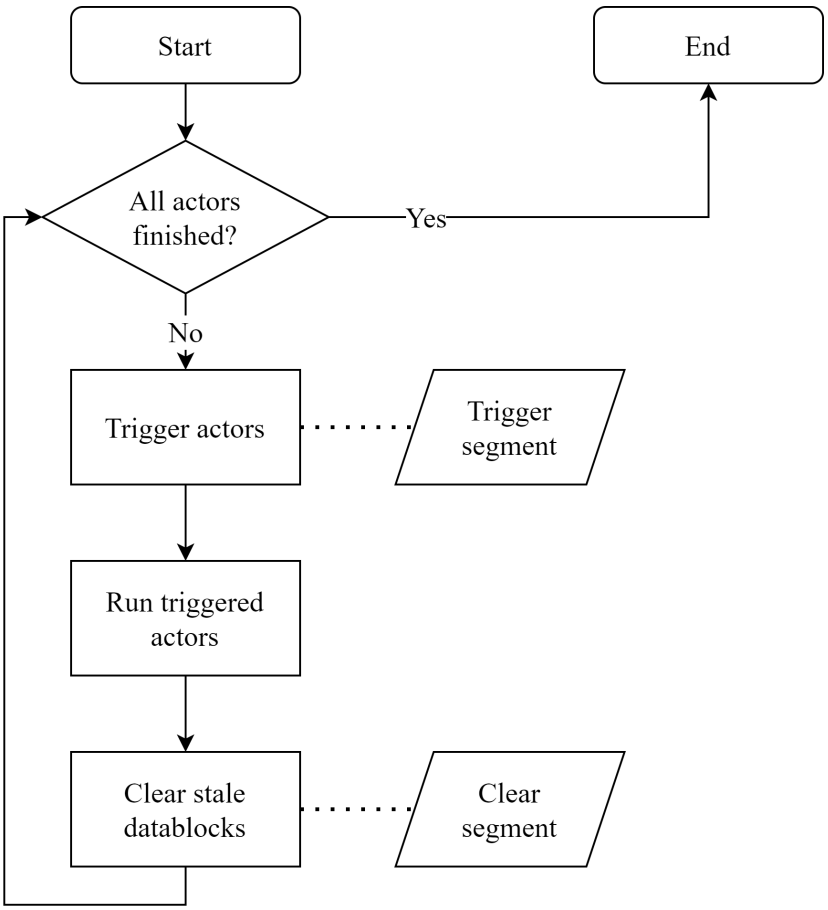


Figure 5.6.: The run cycle.

6. Shallow Water Equations

In this chapter, we use the Actor-GPI library to simulate two-dimensional shallow water equations. We use the preexisting actor implementation *Pond* implemented by Pöppl *et al.*[11], and replace the underlying Actor-UPC++ library with Actor-GPI. We compare performance of both versions and discuss the results obtained.

6.1. Background

The shallow water equations describe the behavior of water in a 2D domain. Here, the equations are solved assuming finite water volume, with varying water height h and elevation of sea floor b. The biggest assumption made in the model is that there is negligible vertical flow of water, necessitating the "shallow water" nomenclature. Also, for phenomena such as tsunamis, the disturbances of the ocean floor affect the whole water column, so vertical flow can again be neglected. With these assumptions in mind, the behavior of the water body with respect to time can be expressed through the following system of equations [22][23] :

$$\frac{\partial h}{\partial t} + \frac{\partial (v_x h)}{\partial x} + \frac{\partial (v_y h)}{\partial y} = 0 \tag{6.1}$$

$$\frac{\partial (h v_x)}{\partial t} + \frac{\partial (h v_x^2)}{\partial x} + \frac{\partial (h v_y v_x)}{\partial y} + \frac{1}{2} g \frac{\partial (h^2)}{\partial x} = -gh \frac{\partial b}{\partial x} \tag{6.2}$$

$$\frac{\partial (h v_y)}{\partial t} + \frac{\partial (h v_x v_y)}{\partial x} + \frac{\partial (h v_y^2)}{\partial y} + \frac{1}{2} g \frac{\partial (h^2)}{\partial y} = -gh \frac{\partial b}{\partial y} \tag{6.3}$$

where h denotes the water height, v_x and v_y the velocities in the respective coordinate directions, b the elevation of the sea floor and g the gravitational constant.

Equation 6.1 can be derived from the conservation of mass in a control volume. Since h is directly proportional to the mass of water (with constant volume and density) the terms hv_x and hv_y are proportional to the momentum of the water in the respective directions. Hence Equation 6.2 and Equation 6.3 can be derived from conserving momentum. Both these equations include the effects of gravity exerting a hydrostatic pressure, and the effect of an uneven ocean floor.

6.2. Adapting SWE in Pond

Breuer and Bader [24][23] developed a software package (titled SWE) to approximate the 2D SWE using a finite volume method. Firstly, the momenta along each direction are defined as

$hu(t,x,y) = h(t,x,y)v_x(t,x,y)$ and $hv(t,x,y) = h(t,x,y)v_y(t,x,y)$. Then, the simulation area is discretized into cells of dimensions Δx and Δy along the Cartesian grid, and each cell is given indices i, j. The assumption is that for a given cell, the four functions (water height $h_{i,j}$, momenta $hu_{i,j}$ and $hv_{i,j}$ and sea floor height $b_{i,j}$) are constant.

Now, given a cell (i, j), its vector of conserved values at some time step t^n is denoted by $Q^n_{i,j} = [h_{i,j}, hu_{i,j}, hv_{i,j}]$. Since the values at any grid point only depend on the value of its neighboring cells, we only need to calculate the *numerical fluxes* on each of the 4 edges (which approximates the transfer of momentum/mass) to update the values of the cell. This is done iteratively at each time step.

Hence the values of the vector $Q^{n+1}_{i,j}$, which are the conserved values of the cell at time step t^{n+1}, are calculated with the help of Equation 6.4. Δt denotes the time step value. The terms $A^{\pm}\Delta Q^n_{i\mp\frac{1}{2},j}$ correspond to the solution of the Riemann problem at the left and the right edges of the cell respectively, and similarly the terms $B^{\pm}\Delta Q^n_{i,j\mp\frac{1}{2}}$ are the solutions of the Riemann problem at the bottom and top edges of the cell. These terms are also shown diagrammatically in Figure 6.1 and Figure 6.2.

$$Q^{n+1}_{i,j} = Q^n_{i,j} - \frac{\Delta t}{\Delta x}(A^+\Delta Q^n_{i-\frac{1}{2},j} + A^-\Delta Q^n_{i+\frac{1}{2},j}) - \frac{\Delta t}{\Delta y}(B^+\Delta Q^n_{i,j-\frac{1}{2}} + B^-\Delta Q^n_{i,j+\frac{1}{2}}) \tag{6.4}$$

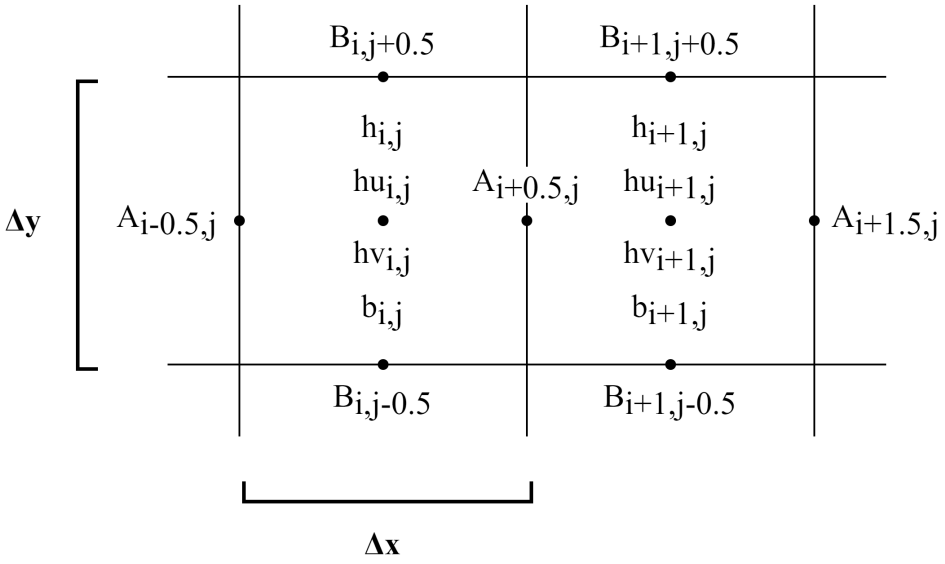


Figure 6.1.: Grid of domain cells with associated variables.

The implementation of SWE grouped together many cells in *blocks*, which facilitated concurrent programming; each block could be placed on a parallel thread and blocks would communicate values of boundary cells with neighboring blocks. Each block is therefore surrounded by extra cells, so called *ghost cells* for easy exchange with the boundary cells of another block. This is shown in Figure 6.3.

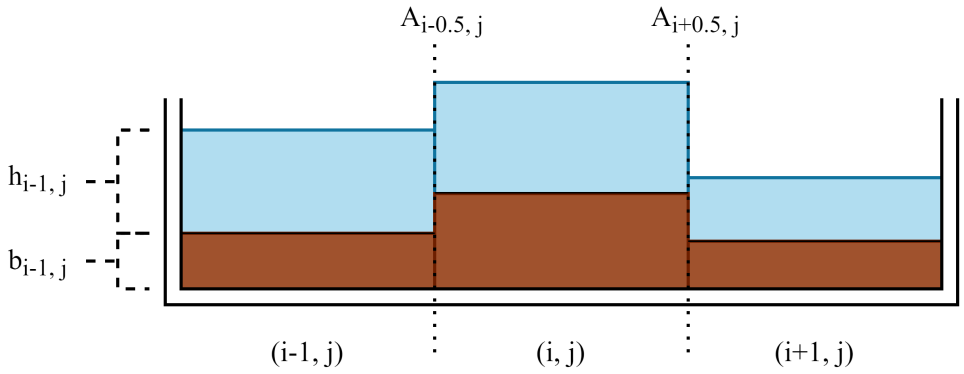


Figure 6.2.: Slice view of grid cells, showing sea floor height and water height.

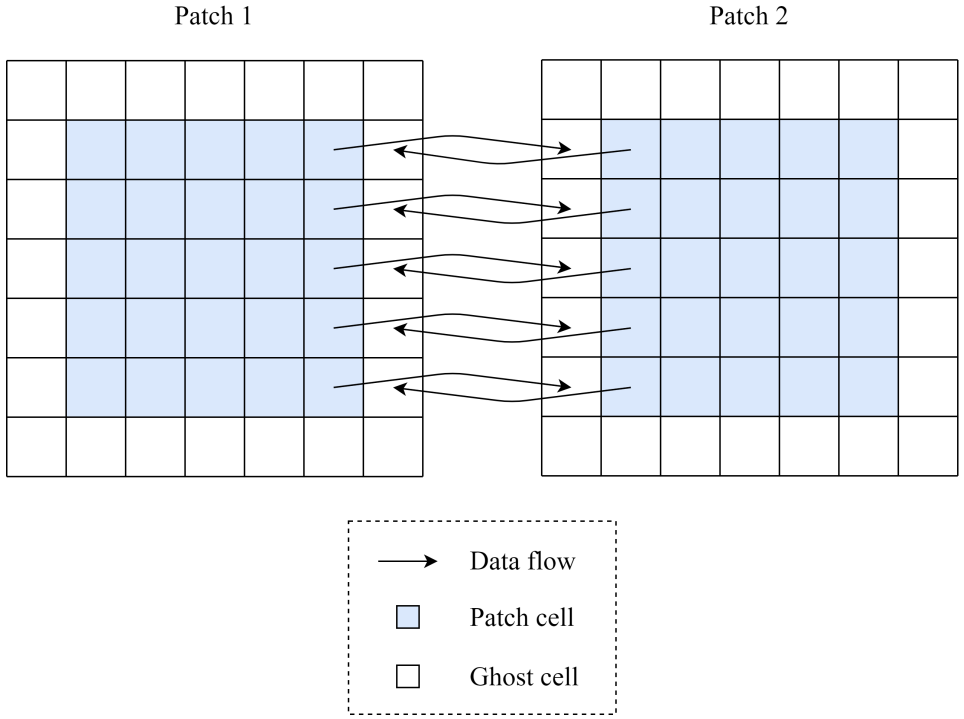


Figure 6.3.: Neighboring blocks made of cells, exchanging data through ghost cells.

This was adapted to run on an actor graph in the work by Pöppl *et al.* [25] who named it the Pond application in their work. Here, the simulation area is divided into sub-areas called *patches*, which each consist of many cells. Functionally, they are the same as the blocks defined earlier. Since the domain is broken up into blocks that communicate with their neighbors, the layout can be easily modeled with an actor graph, where each actor is connected to its neighboring actors in a Cartesian grid structure.

Hence, having divided the domain into patches, each patch can be assigned to an individual actor, which can then use the actor library's communication channels to exchange boundary cells with its adjoining patches. The actor graph is partitioned using the METIS library [26] to ensure an optimal distribution of actors across nodes to reduce inter-node communication. The domain size, patch size and number of ranks are passed to the program at startup, and the program initializes the actors and the graph accordingly.

An example distribution is shown in Figure 6.4, where 6 actors are each given a patch with 4 cells. Three actors are placed on one node, while three are placed on another.

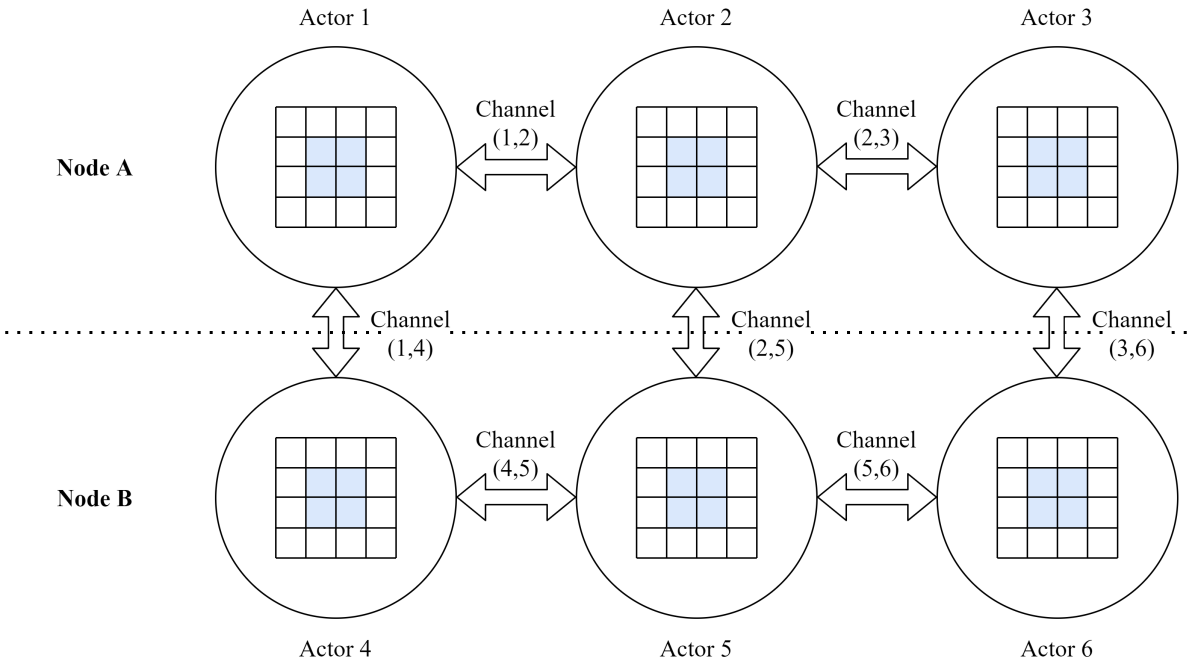


Figure 6.4.: Graph of actors, each with one patch assigned.

The simulation runs in time steps. For each time step:

1. The boundary cell values are set, either by communicating with a neighboring patch or by internal calculations, depending on the scenario.
2. The fluxes for all cells within a patch are calculated.
3. The conserved values for each cell are updated.

6.3. Experimental Setup

To test the performance of Actor-GPI, the Pond application was adapted to use GPI and a variety of tests were run using both Actor-UPC++ and Actor-GPI. Each test used a certain number of nodes, ranging from 1 to 32. The tests were run on *Linux-Cluster CoolMUC-2*, maintained by the *Leibniz-Rechenzentrum (LRZ)*, containing 384 28-way *Intel Xeon E5-2697 v3 ("Haswell")* processors. However, this environment provided limited support for running GPI-only programs, and there were technical issues caused by launching applications with a large number of ranks. The LRZ was made aware of these issues, and workarounds were found for most cases with the LRZ's help.

Hence the Actor-GPI library in this setup had to use the hybrid MPI-GPI model mentioned in section 4.8. Furthermore, the execution had to be limited to 10 ranks per node (one processor per rank) to avoid errors initializing and registering ranks to groups. These concessions were reflected in the Actor-UPC++ tests to ensure that fair comparisons are made. OpenMP was not used in any of the tests.

With the given conditions (1 to 32 nodes, 10 ranks per node) strong and weak scaling tests were performed.

- For strong scaling tests, the domain size was kept constant (16384 $\times$ 16384 cells) as the number of nodes increased, to test the performance of the application with similar problem size but doubling resources. The patch size was reduced to ensure that the number of actors increase. The test particulars are given in Table 6.1.
- For weak scaling tests, the patch size was kept constant (256 $\times$ 256 cells) and the domain size was doubled as the number of nodes increased, to see if the library performs as expected if the domain increases with the resources available. As the domain size increases, the cell size decreases and hence more time steps are needed to reach the stipulated end time. The details are given in Table 6.2.

Table 6.1.: Parameters for the strong scaling tests. Actor numbers are automatically calculated.

Nodes	Ranks	Domain size X	Domain size Y	Patch size	Number of Actors
1	10	16384	16384	1024	256
2	20	16384	16384	1024	256
4	40	16384	16384	512	1024
8	80	16384	16384	512	1024
16	160	16384	16384	256	4096
32	320	16384	16384	256	4096

The tests were run 5 times each, and the results were averaged.

Table 6.2.: Parameters for the weak scaling tests. Actor numbers are automatically calculated.

Nodes	Ranks	Domain size X	Domain size Y	Patch size	Number of Actors
1	10	2048	4096	256	128
2	20	4096	4096	256	256
4	40	4096	8192	256	512
8	80	8192	8192	256	1024
16	160	8192	16384	256	2048
32	320	16384	16384	256	4096

6.4. Results

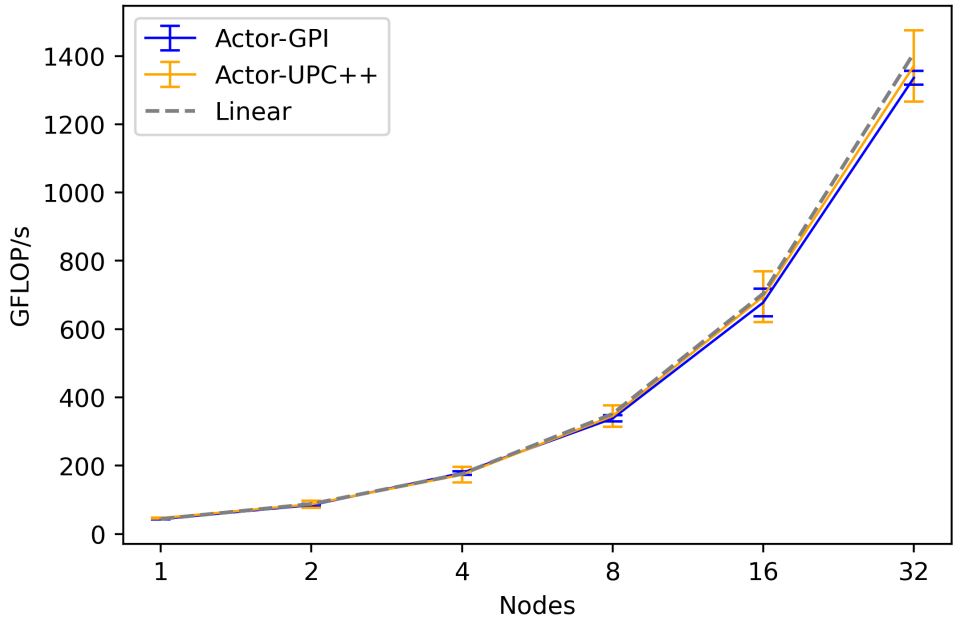
The average results for the strong and weak scaling tests are given in Table 6.3 and Table 6.4 respectively. Speedup is calculated by taking the performance on one node as the baseline. A graphical comparison, with error bars depicting the range of values, are given in Figure 6.5, Figure 6.6 and Figure 6.7.

As we can see, Actor-GPI, despite performing poorer on average than Actor-UPC++, scales somewhat linearly with the problem size and provides comparable results. GPI does not provide dynamic space allocation for global memory segments, and hence this implementation of the actor library used the concept of utilizing large data blocks that have to be reserved and cleared by communicating channels. Each data push or pull required several calls to the GPI library functions, to set up data areas and communicate metadata. Hence communication is expected to take more time for the Actor-GPI library. However, this manual setup may contribute in more consistent performance, as we can see smaller run-to-run variability for the GPI library when compared to the UPC++ library.

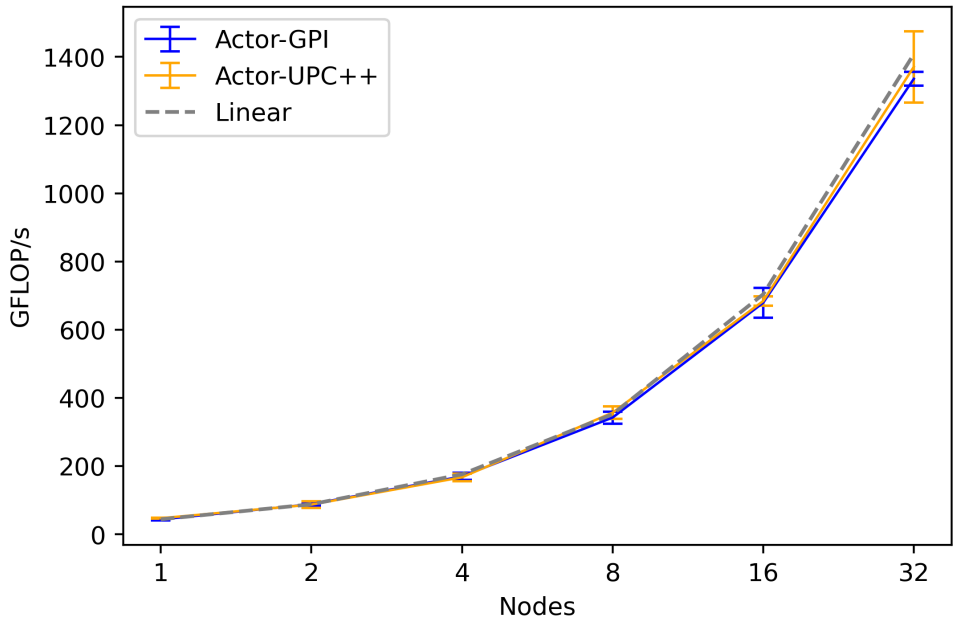
The speedup behavior is consistent with expectations, which shows that the library scales well with the problem size and utilizes extra resources effectively. GPI is built to exploit RDMA and InfiniBand technology, which the CoolMUC-2 cluster provides native support for. This may explain the consistent speedup we see as the number of nodes are increased.

Table 6.3.: Average results for the strong scaling tests.

		GigaFLOP/s		Time (in seconds)		Speedup	
Nodes	Ranks	GPI	UPC	GPI	UPC	GPI	UPC
1	10	44.03	46.52	7318.26	6926.54	1	1
2	20	84.72	87.17	3804.35	3702.27	1.92	1.87
4	40	178.75	173.67	1802.49	1858.55	4.06	3.73
8	80	338.39	345.48	952.14	933.61	7.69	7.43
16	160	677.8	695.12	475.51	464.3	15.4	14.94
32	320	1335.72	1370.81	241.2	235.18	30.34	29.47

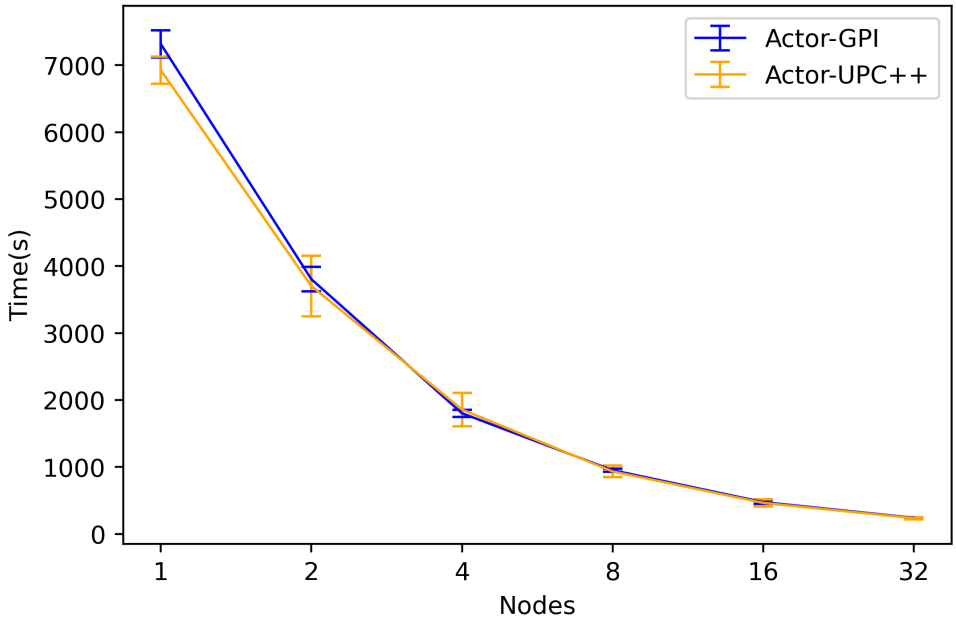


(a) Strong scaling results.

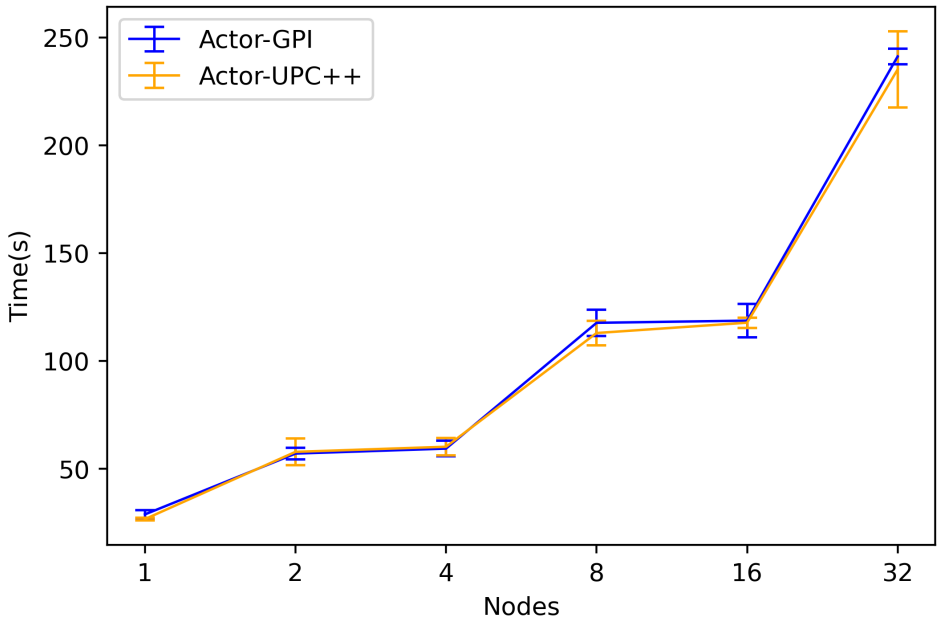


(b) Weak scaling results.

Figure 6.5.: Results of tests - GFLOP/s versus number of nodes.

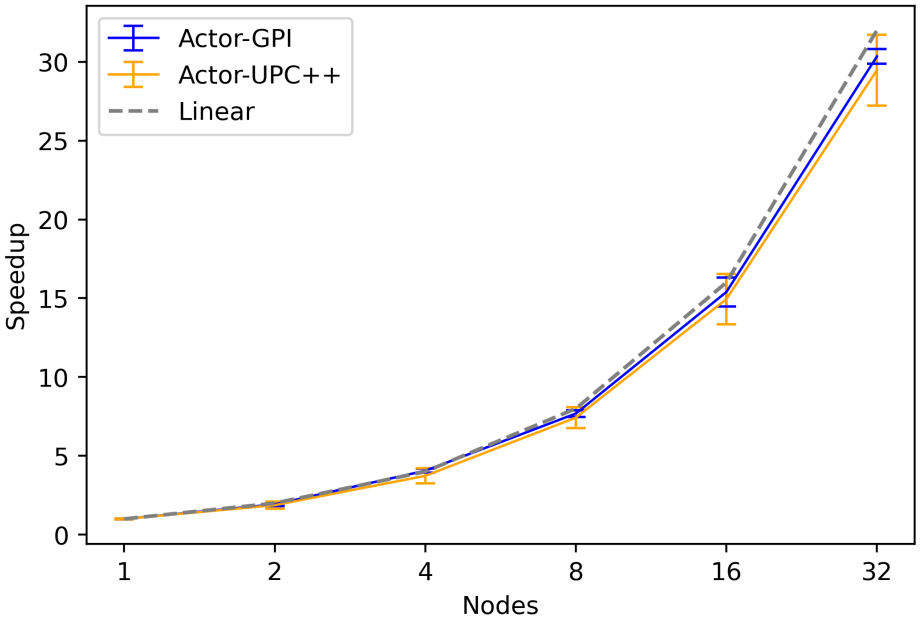


(a) Strong scaling results.

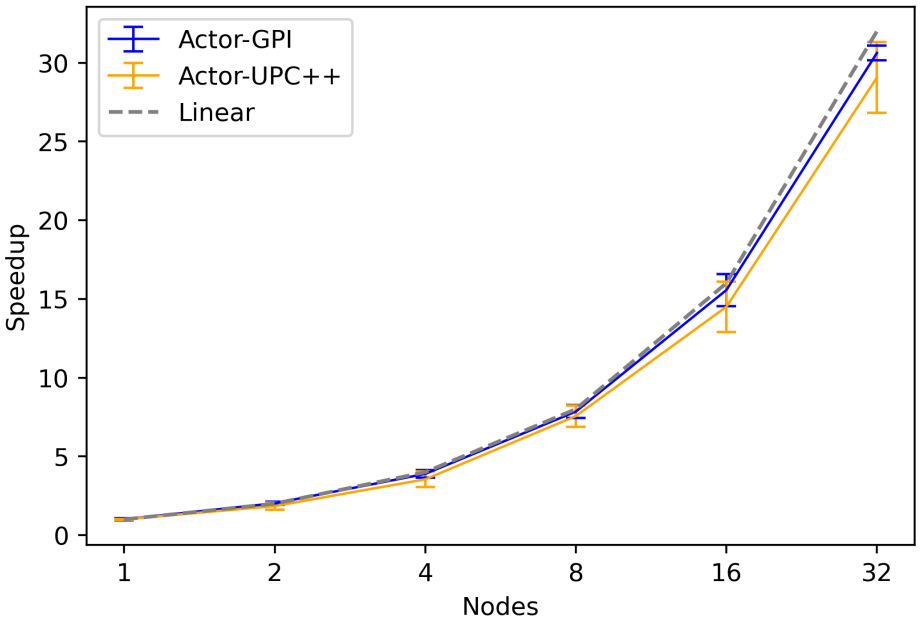


(b) Weak scaling results.

Figure 6.6.: Results of tests - time in seconds versus number of nodes.



(a) Strong scaling results.



(b) Weak scaling results.

Figure 6.7.: Results of tests - speedup versus number of nodes.

Table 6.4.: Average results for the weak scaling tests.

		GigaFLOP/s		Time (in seconds)		Speedup	
Nodes	Ranks	GPI	UPC	GPI	UPC	GPI	UPC
1	10	43.62	47.18	28.89	26.69	1	1
2	20	88.26	87.18	57.09	57.88	2.02	1.85
4	40	169.74	167.28	59.39	60.25	3.89	3.55
8	80	342.35	356.66	117.7	112.98	7.85	7.56
16	160	678.87	684.08	118.72	117.77	15.56	14.5
32	320	1335.72	1370.81	241.2	235.18	30.62	29.06

6.5. Computation and Communication

Since an important goal of PGAS models is to overlap computation and communication by using asynchronous routines, we take the opportunity here to investigate the Pond application's computation and communication times, and see whether it manages to effectively "hide away" the communication in the total runtime.

As stated before, the Pond application breaks down the computation domain into patches, and each patch is assigned an actor. These actors are placed on various processes and the whole graph is run to calculate the patch values. Continuing from section 6.2, here we can go more in depth about the particulars of the implementation.

- Upon receiving the simulation area and other parameters, the program loads the requisite scenario to simulate and creates patches and actors.
- The actors are connected to form an actor graph, and the METIS library [26] is used to partition this graph into connected subgraphs, each to be placed on different ranks. The idea is to partition the graph in such a way that actors are distributed evenly while communication time is minimized.
- The graph is run on each rank. Each triggered actor is run one by one.
- Upon being run, the `SimulationActor` object:
  1. Checks whether this is the first time it is run. If this is the case, then initial patch boundary data is sent to neighbors.
  2. If this is not the first run, the actor checks whether all of its input ports have data, and all of its output ports have space to send data. If this is not true, the actor exits.
  3. If it does have incoming data and the ability to send more data, the actor reads in from the input ports, computes numeric fluxes along the edges of its cells, updates the internal cells and the boundaries of the patch, and sends the boundary data to the connected actors.

4. The run time is incremented by the time step. If the run time exceeds the total time given, the actor terminates itself,

- Actors are run when there are triggers, and all the graphs terminate when the actors determine that they have run for enough time steps.

Due to the lack of shared-memory parallelization, each rank only runs one actor at a time. Each actor has to strictly separate communication and computation (since all computation in a patch necessitates reading in the whole ghost layer first). To test the computation-communication overlap in the whole graph, an additional test was done separating computation and communication. This was done by running the program with the relevant sections deleted. This was done for both Actor-GPI and UPC++. The computations are identical given the problem size, hence the results obtained for the computation tests are assumed to be equal for both libraries. The results are shown in Table 6.5, and are compared with the values for the usual runs of the libraries, with both computation and communication.

Table 6.5.: Separate computation and communication tests. All results are time in seconds.

Test parameters			Results (GPI)			Results (UPC++)		
Scaling	Nodes	Comp.	Comm.	Total	Normal	Comm.	Total	Normal
Strong	1	6327.48	18.7	6346.18	7318.26	22.14	6349.62	6926.54
Strong	2	3347.25	9.6	3356.85	3804.35	11.05	3358.3	3702.27
Strong	4	1649.22	9.73	1658.95	1802.49	8.96	1658.18	1858.55
Weak	1	23.71	0.63	24.34	28.89	0.28	23.99	26.69
Weak	2	55.73	1.11	56.84	57.09	0.56	56.29	57.88
Weak	4	47.18	1.52	48.7	59.39	0.71	47.89	60.25

As we can see, in every case the total time for individual computation and communication is lower than that achieved by running the program normally. For both GPI and UPC++, we have significant overhead. This indicates that the hard synchronization (needed to ensure that the boundary cells received by patches are of the correct time step) slows down the program considerably. As actors have to wait to be triggered by all of their neighbors in order to run, some ranks invariably have idle time as their neighboring ranks spend more time in computation. Furthermore, the partitioned graph does not give an equal number of actors to all graphs but instead gives rounded numbers, and this disparity causes more idle time in ranks with fewer actors, which can cause extra slowdown as shown in cases with fewer ranks.

To mitigate this time loss, the library can implement shared-memory parallelization using OpenMP. Having multiple actors run on the same rank simultaneously can help overlap multiple communication calls with computation from other actors from the same ranks, which helps ensure that processes have minimal downtime. Future work can include some sort of *actor migration* strategy [27], moving actors from activity hotspots and assigning them to ranks with more downtime. Better graph partitioning algorithms can also be explored.

6.6. Tracing Function Runtime

To further check the communication overhead, Intel VTune Amplifier was used to profile the program. A program run on 32 nodes, with 320 ranks, on a domain size of 16384 × 16384, with a patch size of 256 was analyzed. The outputs are shown in Figure 6.8 and Figure 6.9, which show the execution times of the most expensive functions of the library, and the execution times of all the inter-node communication functions of the library respectively. The column *CPU time: Self* denotes the total time the CPU spends executing the function, excluding the callees of the function. *CPU time: Total* shows the total percentage of time the CPU spends in the function, including the time spend executing the callees. The functions are sorted in descending order of *CPU time:self*.

Function	CPU Time: Total	CPU Time: Self
SWE_WaveAccumulationBlock::computeNumericalFluxes	66.6%	1883.420s
pgaspi_barrier	10.8%	304.419s
ActorGraph::clearDataAreas	6.3%	177.102s
SWE_WaveAccumulationBlock::updateUnknowns	5.2%	146.602s
ActorGraph::run	85.5%	137.947s
_gaspi_allreduce.constprop.7	0.8%	22.222s

Figure 6.8.: Most resource intensive functions.

Function	CPU Time: Total	CPU Time: Self
RemoteChannel<std::vector<float, std::allocator<float>>, (int)32>::pushData	0.1%	0.128s
RemoteChannel<std::vector<float, std::allocator<float>>, (int)32>::pullData	0.4%	1.468s
RemoteChannel<std::vector<float, std::allocator<float>>, (int)32>::isAvailableToPush	0.4%	10.914s
RemoteChannel<std::vector<float, std::allocator<float>>, (int)32>::isAvailableToPull	0.7%	1.352s

Figure 6.9.: Actor library communication functions.

As we can see in Figure 6.8, the most expensive functions are the mathematical functions calculating patch values. The `gaspi_barrier` function also takes sizable CPU time. This is because, due to the library being an MPI-GPI hybrid program, there are mandatory barriers required to properly set up the GPI ranks across nodes. This is a cost only paid in initial setup, and does not affect the simulation runtime.

Keeping these results in mind, we see in Figure 6.9 that the communication routines take hardly any time compared to the mathematical functions. Hence the communication routines do not create significant overhead, at least in the final strong scaling case among the given test cases (which would be expected to take the largest amount of time, since it is the case with the greatest number of actors and ranks, and hence the greatest number of inter-process communication requests). Communication can always be streamlined more in the future versions of the library, but the current implementation provides a good starting point with relatively inexpensive communication between nodes.

7. Summary

In this final chapter, we outline the work in the previous chapters. We restate the conclusions drawn, and suggest directions for future development of Actor-GPI library and the actor library in general.

7.1. Outline of Work

The actor library provides a helpful abstraction layer for parallel programming, where computation and communication are separated into *actors* and *channels*. Actors are black-box state machines that transition from state to state upon receiving or sending data through connected channels. The intended usage of the library is to implement custom actors by subclassing the `actor` class and defining its `act()` method. Then actors are connected with the help of channels to form an *actor graph*, which can then be executed in a parallel manner by partitioning the graph and distributing the actors to separate processors. Actors can send and receive data from connected actors, and can terminate themselves when their stopping criteria are met.

The purpose of the thesis is to provide an actor library implementation using GPI. GPI is an implementation of the GASPI standard, which in turn is a Partitioned Global Address Space (PGAS) API. GPI provides asynchronous one-sided communication and global shared memory among processes. Shared memory is implemented by marking large memory areas as *segments*, which can be indexed via offset. Any process can read to and from segments on any other process, using communication requests which are posted to *queues*. Synchronicity is maintained by using *notifications* if required. Embedding into MPI programs is also possible if running GPI is not supported in the required environment.

The actor library is thus implemented using GPI to facilitate inter-node communication between actors. Several segments are set up to handle data, metadata and commands. The actors use the data and metadata segments to communicate with connected actors, and the command segments to communicate with the remote actor graph the connected actor is a part of. Each rank maintains a large data area, which actors can reserve blocks from. The location of a reserved block is sent to the intended recipient actor of the data, who can then read the data directly without any intervention. Then the actor can send triggers to other actors, or mark the data segment as stale so that it can be cleaned up.

For testing purposes, this library was used to solve shallow water equations to simulate wave propagation in tsunamis. The water area to simulate was broken up into patches, with the understanding that each patch's internal state can be calculated solely from the height and momentum of the water at its surrounding patch borders. Each patch was assigned to

an actor, and actors containing neighboring patches were connected. This actor graph was partitioned and distributed to the different processes, and executed on the *LRZ - CoolMUC-2* cluster. The results were collected and compared with the library this work was ported from, Actor-UPC++.

7.2. Results and Conclusions

In both the strong and weak scaling tests performed, the library shows expected levels of speedup with increasing resources. Overall performance is slower than that of Actor-UPC++, which can be explained by the requirement of multiple communication requests per data push due to the need of communicating metadata.

Upon running the communication and computation separately, we can see that the load distribution among the processes is uneven, causing some slowdown in tests with fewer nodes. The requirement of having an actor run only when it is triggered also means actors have to wait idly, which means sub-optimal performance.

However, the communication functions themselves take up very little CPU time, since all communication is asynchronous. Hence, they contribute to very little overhead, and with the right graph structure, can produce great results.

7.3. Future Work

As noted in section 6.5, the current version of the library lacks the ability to move actors to idle ranks dynamically, to better balance the load and optimize resource usage. If actor migration can be done without too much overhead, it will help greatly in further improving the application performance.

Also, due to the difficulties of partitioning segments for local actors, shared-memory parallelization was not implemented in the current iteration of the library. The main trouble lies with the cache segment, which can be used by the all the actors to send and receive data; it cannot be guaranteed that no data collision will happen when two actors use the same cache segment. Perhaps the segment can be partitioned like other segments are, or multiple cache segments associated with the various metadata segments can be created.

The current model of the actor library can also be improved. The actual low-level functions for sending and receiving data may be asynchronous, but the abstractions `sendData()` and `receiveData()` implemented in the channels are synchronous. Making them non-blocking can allow for more efficient overlap of computation and communication, leading to better performance.

Finally, the current library utilizes a lot of workarounds to run on the cluster environment with minimal issues. With a better environment, the library can be made to be purely GPI instead of the hybrid MPI-GPI implementation we have now. It will be interesting to compare and contrast the performances of the pure library with the hybrid one.

A. Code for the Primer Example

This section contains example code for the illustrative example in section 3.3. For simplicity's sake, the first actor *PrimeSourceActor* passes 5 predetermined values and does not generate them randomly.

Code Snippet A.1.: Code for PrimeSourceActor class

```
// PrimeSourceActor
class PrimeSourceActor: public Actor
{
public:
    void act();
    int noLeftToSend;
    std::vector<int> numList;
    PrimeSourceActor(uint64_t rank, uint64_t srno);
};

PrimeSourceActor::PrimeSourceActor(uint64_t rank, uint64_t srno)
    : Actor(rank, srno)
{
    noLeftToSend = 5; //number of digits to send
}

void PrimeSourceActor::act()
{
    if(finished) return; //failsafe

    if(numList.size() == 0)
    {
        numList.push_back(12);
        numList.push_back(500);
        numList.push_back(7);
        numList.push_back(1000);
        numList.push_back(50);
        this->triggers = 5; //since producer, needs manual triggers
    }

```

```
31      auto port = getOutPort<int, 3>("outSource");
32      if(port->freeCapacity() > 0)
33      {
34          std::cout << "source sending " <<
35              numList[numList.size() - noLeftToSend << std::endl;
36          port->write(numList[numList.size() - noLeftToSend]);
37          noLeftToSend--;
38      }
39      else
40      {
41          std::cout << "queue full for source, waiting... " << std::endl;
42          this->trigger(); //replace the consumed trigger
43      }
44
45      if(noLeftToSend == 0)
46          finished = true;
47
48  }
```

Code Snippet A.2.: Code for PrimeCalcActor class

```
1   // PrimeCalcActor
2   class PrimeCalcActor: public Actor
3   {
4   public:
5       void act();
6       int noListsSent;
7       std::vector<int> primes;
8       bool listPending;
9       PrimeCalcActor(uint64_t rank, uint64_t srno);
10  };
11
12  PrimeCalcActor::PrimeCalcActor(uint64_t rank, uint64_t srno)
13      : Actor(rank, srno)
14  {
15      noListsSent = 0;
16      listPending = false;
17  }
18
19  void PrimeCalcActor::act()
```

```
20 {
21     if(finished) return; //failsafe
22 
23     if(listPending)
24     {
25         std::cout << "list pending to be sent to print, resending..."
26             <<std::endl;
27     }
28     else
29     {
30         //must populate list
31         primes.clear();
32         auto port = getInPort<int, 3>("inCalc");
33         int data = 0;
34         if(port-> available())
35         {
36             data = port->read();
37             std::cout << "primer received "<< data << std::endl;
38         }
39         else
40         {
41             std::cout << "primer triggered without input data!" << std::endl;
42             return;
43         }
44         //at this point data is received
45         primes.push_back(2);
46         primes.push_back(3);
47         bool isPrime;
48         for(int i = 5; i < data; i++)
49         {
50             isPrime = true;
51             for(int j = i - 1; j > 1; j--)
52             {
53                 if(i%j == 0)
54                     isPrime = false;
55                 if(!isPrime)
56                     break;
57             }
58             if(isPrime)
59                 primes.push_back(i);
60         }
61         listPending = true;
```

```
62      }
63      //at this point we have a list to send in either case
64      auto printPort = getOutPort<std::vector<int>, 3>("outCalc");
65      if(printPort->freeCapacity() >= sizeof(int) * primes.size())
66      {
67          printPort->write(primes);
68          noListsSent++;
69          listPending = false;
70      }
71      else
72      {
73          std::cout<< "primer outqueue full, trying again soon" <<std::endl;
74          this->trigger();
75              //will go thru true branch of listpending with this trigger
76      }
77
78      if(noListsSent == 5)
79      {
80          this->triggers = 0;
81          finished = true;
82      }
83  }
```

Code Snippet A.3.: Code for PrimePrintActor class

```
1   // PrimePrintActor
2   class PrimePrintActor: public Actor
3   {
4   public:
5       void act();
6       int recdNos;
7       PrimePrintActor(uint64_t rank, uint64_t srno);
8   };
9
10  PrimePrintActor::PrimePrintActor(uint64_t rank, uint64_t srno)
11      : Actor(rank, srno)
12  {
13      recdNos = 0;
14  }
15
```

```
16 void PrimePrintActor::act()
17 {
18     if(finished) return;
19     auto port = getInPort<std::vector<int>, 3>("inPrinter");
20 
21     if(port-> available())
22     {
23         std::vector<int> data = port->read();
24         recdNos++;
25         std::cout << "sink received: ";
26         for(auto it = data.begin(); it != data.end(); ++it)
27             std::cout << *it << " ";
28         std::cout << std::endl;
29     }
30     else
31     {
32         std::cout << "print triggered without input data!" << std::endl;
33     }
34 
35     if(recdNos == 5)
36         finished = true;
37 
38 }
```

Code Snippet A.4.: Code for main method

```
1  // main method
2 
3  #ifndef ASSERT
4  #define ASSERT(ec) gpi_util::success_or_exit(__FILE__,__LINE__,ec)
5  #endif
6 
7  int main(int argc, char *argv[])
8  {
9      MPI_Init(&argc, &argv); //MPI-GPI hybrid code
10 
11     gaspi_rank_t rank, num;
12     gaspi_return_t ret;
13 
14     ASSERT( gaspi_proc_init(GASPI_BLOCK) );
```

```
15
16      ActorGraph ag;
17
18      ASSERT( gaspi_proc_rank(&rank) );
19      ASSERT( gaspi_proc_num(&num) );
20
21      Actor *localActor1;
22
23      if(rank == 0)
24      {
25          localActor1 = new PrimeSourceActor(rank, 0);
26      }
27      else if (rank == 1)
28      {
29          localActor1 = new PrimeCalcActor(rank, 0);
30      }
31      else
32      {
33          localActor1 = new PrimePrintActor(rank, 0);
34      }
35
36      ag.addActor(localActor1);
37
38      ASSERT (gaspi_barrier (GASPI_GROUP_ALL, GASPI_BLOCK));
39
40      ag.syncActors();
41      ag.printActors();
42
43      ag.connectPorts<int, 3>(Actor::encodeGlobID(0,0),"outSource",
44              Actor::encodeGlobID(1,0), "inCalc");
45      ag.connectPorts<std::vector<int>, 3>(Actor::encodeGlobID(1,0),"outCalc",
46              Actor::encodeGlobID(2,0), "inPrinter");
47
48      double rt = ag.run();
49      gaspi_printf("Runtime from rank %d: %lf\n",rank,rt);
50
51      ASSERT (gaspi_barrier (GASPI_GROUP_ALL, GASPI_BLOCK));
52
53      ASSERT( gaspi_proc_term(GASPI_BLOCK) );
54      MPI_Finalize();
55      return EXIT_SUCCESS;
56  }
```

Code Snippet A.5.: Helper code for GPI

```
// success_or_exit method
namespace gpi_util
{
    static void success_or_exit(const char* file,
        const int line, const gaspi_return_t ec)
    {
        if(ec!=GASPI_SUCCESS)
        {
            gaspi_printf("Assertions failed in %s[%i]:%d\n",file,line,ec);
            gaspi_string_t error = gaspi_error_str(ec);
            gaspi_printf("Error string: %s\n", error);
            exit(EXIT_FAILURE);
        }
    }
}
```

B. Guide to Running Actor-GPI

B.1. Hardware and Software Setup

B.2. Compiling and running Actor-GPI

Code Snippet B.1 contains the commands to compile Actor-GPI from source. The commands should run as-is on the CoolMUC-2 cluster. The package includes a rudimentary `FindGPI.cmake` file, which requires an environment variable `$GPI_2_BASE` to be set to the installation path of GPI. This is automatically set by the `module load` commands in the cluster environment. In other environments this should be manually set. The commands should be executed from the root of the repository.

Code Snippet B.1.: Bash commands for compiling Actor-GPI

```
#load required modules
module use -p \
    /lrz/sys/spack/.tmp/test/gpi2/modules/x86_avx2/linux-sles12-x86_64
module load gpi-2/1.4.0-gcc8-impi-rdma-core-24
module load cmake/3.16.5
module load metis/5.1.0-intel19-i32-r32
module load netcdf-hdf5-all/4.6_hdf5-1.10-intel19-impi
module load gcc/8

#compile the code
mkdir build && cd build
CXX="mpiCC" CC="mpicc" cmake ..
make -j 8
```

In Code Snippet B.2 we have a sample batch script corresponding to the largest test case in chapter 6, namely the case with 32 nodes, 10 ranks per node, a 16384×16384 field with a patch size of 256. Execution string should be identical even in non-SLURM environments.

Code Snippet B.2.: Job script for running Pond with GPI

```
#!/bin/bash
#SBATCH -J GPIstrong32
```

```
#SBATCH -o ./%x.%j.%N.out
#SBATCH -D ./
#SBATCH --get-user-env
#SBATCH --clusters=cm2
#SBATCH --partition=cm2_large
#SBATCH --qos=cm2_large
#SBATCH --nodes=32
#SBATCH --ntasks-per-node=10
#SBATCH --mail-type=all
#SBATCH --mail-user=<enter email>
#SBATCH --export=NONE
#SBATCH --time=03:00:00

module load slurm_setup

module use -p \
    /lrz/sys/spack/.tmp/test/gpi2/modules/x86_avx2/linux-sles12-x86_64
module load gpi-2/1.4.0-gcc8-impi-rdma-core-24
module load metis/5.1.0-intel19-i32-r32
module load netcdf-hdf5-all/4.6_hdf5-1.10-intel19-impi
module load gcc/8

cd <home directory>/actor-gpi-v2/build

export OMP_NUM_THREADS=1

gaspi_logger &
mpiexec -n 320 --perhost 10 ./pond -x 16384 -y 16384 -p 256 -e 1 -c 1 \
    --scenario 2 -o output/out
```

B.3. Compiling and running Actor-UPC++

For the sake of completion, the steps taken in this particular project to compile and execute Actor-UPC++ are given here. In Code Snippet B.3, `$UPCXX_DIR` points to the location of `FindUPCXX.cmake`. `$UPCXX_INSTALL` and `$PATH` should be set to the installation folder, and other variables are set according to Actor-UPC++ specifications. A sample job script for the same test case is given in Code Snippet B.4. The flag `-DACTORLIB_EXECUTION_STRATEGY` can be set to either `thread`, `rank` or `task`. In the current work, `rank` is chosen to be the execution strategy, since it would provide the fairest comparison to Actor-GPIs rank based execution.

More details about Actor-UPC++ can be found in the original work [11].

Code Snippet B.3.: Bash commands for compiling Actor-UPC++

```
#load required modules
module load cmake/3.16.5
module load metis/5.1.0-intel19-i32-r32
module load netcdf-hdf5-all/4.6_hdf5-1.10-intel19-impi
module load gcc/8

#set environmental variables
export PATH="<home directory>/upc/bin:$PATH"
export UPCXX_DIR="<home directory>/actor-upcxx/CMake-Aux/Modules/"
export UPCXX_INSTALL="<home directory>/upc/"
export UPCXX_GASNET_CONDUIT=ibv
export UPCXX_CODEMODE=03

#compile the code
mkdir build && cd build
CC="mpicc" cmake \
    -DACTORLIB_EXECUTION_STRATEGY=rank -DCMAKE_CXX_COMPILER=mpicxx ..
make -j 8
```

Code Snippet B.4.: Job script for running Pond with UPC++

```
#!/bin/bash
#SBATCH -J UPCstrong32
#SBATCH -o ./%x.%j.%N.out
#SBATCH -D ./
#SBATCH --get-user-env
#SBATCH --clusters=cm2
#SBATCH --partition=cm2_large
#SBATCH --qos=cm2_large
#SBATCH --nodes=32
#SBATCH --tasks-per-node=10
#SBATCH --mail-type=all
#SBATCH --mail-user=<enter email>
#SBATCH --export=NONE
#SBATCH --time=03:00:00

module load slurm_setup
```

```
17
18 module load metis/5.1.0-intel19-i32-r32
19 module load netcdf-hdf5-all/4.6_hdf5-1.10-intel19-impi
20 module load gcc/8
21
22 export PATH="<home directory>/upc/bin:$PATH"
23
24 cd <home directory>/actor-upcxx/build
25
26 export OMP_NUM_THREADS=1
27
28 upcxx-run -n 320 ./pond -x 16384 -y 16384 -p 256 -e 1 -c 1 \
29     --scenario 2 -o output/out
```

List of Figures

3.1. Two actors and a channel. 7
3.2. Actor graph of Primes example. 9

4.1. GPI asynchronous communication mechanism. 13

5.1. Overview of a run of the library. 17
5.2. Interaction between actor library components. 19
5.3. Example graph structure to explain channel offsets. 22
5.4. Sending data to remote rank. 24
5.5. Receiving data from remote rank. 26
5.6. The run cycle. 27

6.1. Grid of domain cells with associated variables. 29
6.2. Slice view of grid cells. 30
6.3. Neighboring blocks of cells. 30
6.4. Actor graph of patches. 31
6.5. Results of tests - GFLOP/s versus number of nodes. 34
6.6. Results of tests - time in seconds versus number of nodes. 35
6.7. Results of tests - speedup versus number of nodes. 36
6.8. Resource usage of the most resource intensive functions. 39
6.9. Resource usage of the library communication functions. 39

List of Tables

5.1. Example offsets generated for channels 23

6.1. Strong scaling test parameters 32
6.2. Weak scaling test parameters 33
6.3. Strong scaling test results 33
6.4. Weak scaling test results 37
6.5. Separate computation and communication tests. 38

Glossary

Finite State Machine A mathematical model defined by a collection of states an initial state, and allowable inputs that trigger transitions. The machine can be at one given state, and in response to some input, transitions to another state.

Remote Direct Memory Access A feature available in some cluster architectures that allow processors to directly read data from the internal memory of other processors, without involving the remote processor's operating system. The data is directly moved onto communication buses without copying into registers first.

InfiniBand A networking communications standard used in HPC originating in 1999.

Intel VTune Amplifier A profiling and performance evaluation tool for single- and multi-threaded programs running on Intel platforms.

Acronyms

APGAS Asynchronous Partitioned Global Address Space.

API Application Programming Interface.

CAF Coarray Fortran.

FIFO First In First Out.

FSM Finite State Machine.

FVM Fraunhofer Virtual Machine.

GASNet Global Address-Space Networking.

GASPI Global Address Space Programming Interface.

GPI Global Address Space Programming Interface.

GPU Graphics Processing Unit.

HPC High Performance Computing.

MPI Message Passing Interface.

OOP Object Oriented Programming.

OpenMP Open Multi-Processing.

PGAS Partitioned Global Address Space.

RDMA Remote Direct Memory Access.

UPC Unified Parallel C.

UPC++ Unified Parallel C++.

Bibliography

[1] C. Hewitt, P. Bishop, and R. Steiger. "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence". In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute. 1973, p. 235.

[2] I. Greif. "Semantics of communicating parallel processes." PhD thesis. Massachusetts Institute of Technology, 1975.

[3] H. Baker and C. Hewitt. "Laws for communicating parallel processes". In: (1977).

[4] W. D. Clinger. "Foundations of actor semantics". In: *AITR-633* (1981).

[5] G. A. Agha. *Actors: A model of concurrent computation in distributed systems.* Tech. rep. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[6] J. Armstrong. "Erlang—a Survey of the Language and its Industrial Applications". In: *Proc. INAP*. Vol. 96. 1996.

[7] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch. "Native Actors: A Scalable Software Platform for Distributed, Heterogeneous Environments". In: *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 87–96. ISBN: 9781450326025. DOI: 10.1145/2541329.2541336. URL: https://doi.org/10.1145/2541329.2541336.

[8] D. Charousset, R. Hiesgen, and T. C. Schmidt. "Revisiting actor programming in C++". In: *Computer Languages, Systems & Structures* 45 (2016), pp. 105–131. ISSN: 1477-8424. DOI: https://doi.org/10.1016/j.cl.2016.01.002. URL: http://www.sciencedirect.com/science/article/pii/S1477842416000038.

[9] J. Bonér, V. Klang, R. Kuhn, P. Nordwal, B. Antonsson, and E. Varga. "Akka scala documentation". In: *Tech. rep.* Typesafe Inc., 2014.

[10] S. M. Jodal. *Pykka documentation*. 2014.

[11] A. Pöppl, S. Baden, and M. Bader. "A UPC++ Actor Library and Its Evaluation On a Shallow Water Proxy Application". In: *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*. 2019, pp. 11–24. DOI: 10.1109/PAW-ATM49560.2019.00007.

[12] S. Roloff, A. Pöppl, T. Schwarzer, S. Wildermann, M. Bader, M. Glaß, F. Hannig, and J. Teich. "ActorX10: An Actor Library for X10". In: *Proceedings of the 6th ACM SIGPLAN Workshop on X10*. X10 2016. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 24–29. ISBN: 9781450343862. DOI: 10.1145/2931028.2931033. URL: https://doi.org/10.1145/2931028.2931033.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing". In: OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 519–538. ISBN: 1595930310. DOI: 10.1145/1094811.1094852. URL: https://doi.org/10.1145/1094811.1094852.

[14] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. "UPC++: a PGAS extension for C++". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 1105–1114.

[15] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. "FunState-an internal design representation for codesign". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9.4 (2001), pp. 524–544. DOI: 10.1109/92.931229.

[16] R. Machado and C. Lojewski. "The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model". In: *Computer Science-Research and Development* 23.3-4 (2009), pp. 125–132.

[17] D. Grünewald and C. Simmendinger. "The GASPI API specification and its implementation GPI 2.0". In: *7th International Conference on PGAS Programming Models*. Vol. 243. 2013, p. 52.

[18] J. Breitbart, M. Schmidtobreick, and V. Heuveline. "Evaluation of the Global Address Space Programming Interface (GASPI)". In: *2014 IEEE International Parallel Distributed Processing Symposium Workshops*. 2014, pp. 717–726. DOI: 10.1109/IPDPSW.2014.83.

[19] O. Lena. "GPI2 for GPUs: A PGAS framework for efficient communication in hybrid clusters". In: *Parallel Computing: Accelerating Computational Science and Engineering (CSE)* 25 (2014), p. 461.

[20] *Website for the GASPI Forum*. URL: http://www.gaspi.de/gaspi/ (visited on 12/26/2020).

[21] *Gaspi : Global Address Space Programming Interface; Specification of a PGAS API for communication*. Version 17.1. GASPI Forum. Feb. 2017.

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

[23] *SWE - A Simple Shallow Water Code*. 2013. URL: https://www5.in.tum.de/SWE/doxy/index.html (visited on 01/28/2021).

[24] A. Breuer and M. Bader. "Teaching Parallel Programming Models on a Shallow-Water Code". In: *2012 11th International Symposium on Parallel and Distributed Computing*. 2012, pp. 301–308. DOI: 10.1109/ISPDC.2012.48.

[25] A. Pöppl, M. Bader, T. Schwarzer, and M. Glaß. "SWE-X10: Simulating Shallow Water Waves with Lazy Activation of Patches Using Actorx10". In: *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*. 2016, pp. 32–39. DOI: 10.1109/ESPM2.2016.010.

[26] G. Karypis and V. Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs". In: *SIAM Journal on scientific Computing* 20.1 (1998), pp. 359–392.

[27] Y. K. Budanaz. “Dynamic Actor Migration for a Distributed Actor Library”. Bachelorarbeit. Technical University of Munich, Aug. 2020.