FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

A Distributed Actor Library for HPC Applications

Bruno Miguel

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

A Distributed Actor Library for HPC Applications

Eine Verteilte Aktorbibliothek für Höchstleistungsrechneranwendungen

Author:	Bruno Miguel
Supervisor:	Prof. Michael Bader, Ph.D.
Advisor:	Alexander Pöppl, M.Sc.
Submission Date:	15.11.2019

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

Munich, 15.11.2019 Bruno Miguel

Acknowledgments

My sincere gratitude to Alexander Pöppl for sharing his excellent expertise in high-performance application development. To the professors and colleagues who have inspired me at the academia. To my family and friends who supported me in the recent years living abroad.

Abstract

The diversity of high-performance computer systems and application requirements harden the generalizability of software development. For these systems, where performance is critical, abstractions costs are unbearable. But, on the other hand, handcrafting code for every single requirement becomes complex, time-consuming and costly. In this thesis, we present an actor programming model library for high-performance applications running in distributed and shared-memory systems. The library aims to decouple computation and communication logic within finite-state machines entities and to release the pitfalls of parallel designs. We explore well-known standards such as MPI and OpenMP to provide broad cross-compatibility and close to native performance. Its usability and performance are explored on complex structured and unstructured mathematical models, and its results compared with other well-known programming models for HPC, such as the BSP and PGAS. By proxying a tsunami-generated simulation model, we achieve notable scalability on both weak and strong scaling performance tests. Our library allows programmers to focus on algorithms and data-dependencies specifications rather than low-level parallelization constructs and system load-balancing.

Contents

1. Introduction

High-performance computing – *HPC* – applications try to accomplish three main objectives: to work with massive problem sizes, to provide fast response times and to compute multiple problems concurrently.

Software that requires significant compute power varies greatly. They can cover the whole spectrum of mathematical models, from the theory to the application field. Generally, these applications try to produce valuable results in a reasonable and economical amount of time. For instance, an astrophysics application may require to work with massive workload sizes; a natural disaster simulator may have important deadlines to fulfill; a cooperate application may be constrained by its infrastructure resources availability.

1.1. Supercomputers

Early computers tried to fulfill high-performance application requirements by amalgamating multiple processors sharing the same memory device. But as the number of processors grew exponentially (reaching millions of processors), shared memory architectures failed to scale. Scalable issues were raised mainly from memory accesses contention and memory coherency between different processors.

To cope with the massive number of processors, modern supercomputers are built upon a centralized distributed memory architecture. Their essential characteristic is to provide groups of shared-memory many-core processors – *nodes* – connected through high-quality network fabrics. Groups maintain their own private memory spaces, and remote memory accesses is only possible through specialized hardware, software packages abstractions or point-to-point communication.

Distributed-memory systems introduce new programming models, and induce the developer to partition and orchestrate computation and workload in order to provide the maximum execution performance.

1.2. High-Performance Application Development

Clusters are assembled from multiple hardware vendors and software packages. Public and private organizations tries to create standards in order to provide programmability transparency; otherwise, cross-platform development would be infeasible due to the high-costs of fulfilling multiple dependencies and of providing accordingly grammatical and semantic rules to develop such highly-coupled applications.

1.2.1. Distributed-Memory Systems

Distributed-memory systems denote multiprocessors architectures where each processor has its own memory addresses. The challenges of developing applications for such systems correlates with computation and communication orchestration. The former latency is likely to be orders of magnitude lower than the latter, and therefore, one must ensure that communication is asynchronous and not frequent.

There are different ways of accessing remote memory. In a lower level, they are related to active message exchanges, or direct memory accesses. Abstractions are created to either provide transparency between local and remote memory accesses, or to indicate their discrepancy.

An example communication model for distributed-memory systems is the use message-passing protocols, where communication between nodes is done through explicit message exchanges. These exchanges are either bidirectional, where all the participating nodes actively call communication operations; or unidirectional, where nodes are freely to access other nodes memory without their participation. In this thesis, we will work closely with MPI, a message-passing protocol standard implemented for a wide-range of architectures.

1.2.2. Shared-Memory Systems

Clusters not only contain numerous interconnected nodes, but the nodes themselves are also a sort of small-scale clusters. Modern processors can run tens or even hundreds of threads concurrently on top of multiple physical and logical cores. Furthermore, nodes can also provide specialized hardwares, such as graphic processing units.

Challenges to develop shared-memory parallel applications are related to threads orchestration, and local memory accesses coordination. Memory dependencies and workload among threads have to be well-aligned and distributed; otherwise, memory coherence and contention would entail great performance overheads.

There exists programming languages, libraries and a combination of both that can abstract the development of shared-memory applications. OpenMP is an example of the former. It provides an API standard for C/C++ and Fortran which defines compiler transformations and library constructs to support thread management and workload orchestration.

OpenMP is based on directives that translates user-code to parallel execution code blocks using the POSIX threading standard. An OpenMP application flow follows a fork/join parallel model: 1. The program starts with a single thread – *master thread* –, and a pre-allocated pool of worker threads. 2. Within specific points in the application, a team of threads is dispatched to execute certain code in parallel. 3. The team of threads finishes their execution, and control is given back to the main thread.

1.3. Objectives

In this thesis, we aim to enhance an existent high-performance programming library that can be used for applications running on shared and distributed-memory systems.

The library is based on the actor programming model, which encapsulates data and computation into entities distributed among the underlying available resources. It eases local and remote memory accesses efforts by providing seamless communication constructs that is optimized for low latency and high bandwidth. Furthermore, the library facilitates data flow and dependencies within the application by providing explicit data exchange constructs. It works with a wide variety of data types, which internally are handled to ensure the lowest memory footprint and overheads.

As we will see during the performance tests, our implementation works well for scientific applications running with structured and unstructured data distribution and parallel execution flow. Furthermore, given the well accepted standards utilized during the development, the library is not coupled to a specific system architecture or application model.

1.4. Outline

In chapter 2 and 3 we introduce the actor model and its related work. Our implementation is derived from an UPC++ version; therefore, on chapter 4, we discuss UPC++ as a programming model for distributed-memory systems. Chapter 5 discuss the internals of MPI, and then chapter 6 describes our implementation. Chapter 7 introduces a mathematical model which we will use to test the performance of our implementation against existent ones. Finally, 8 we discuss future work and final conclusions.

Appendix A.1 describes the cluster architecture used to profile and debug our implementation.

2. Literature

In this chapter, we present related work to the actor model. We describe its origins, related languages and libraries, and state-of-the-art implementations.

The first actor model was proposed by Hewitt [HBS73] to abstract concurrency in applications. Later, Agha [Agh85] provided its first formal computation model. Many programming languages came afterwards to employ the model or support some variation of it.

Erlang was the first to introduce the actor model in industrial applications [Arm96]. An interest in lightweight task based concurrency on multi-core processors further drove the model's development into other platforms. Event-driven environment systems – such the JVM – influenced the development of subsequent lightweight actor implementations.

An essential characteristic of an actor model is to isolate actors' internal states from each other. This is done by creating communication channels between actors, where a channel is only readable by the owning actor. This concept was formalized by Agha [Agh85], which had introduced a FIFO structure, known as a mailbox.

Multiple implementations of a mailbox were developed. It's core concepts falls into two categories. In the first, a mailbox receives messages and stores it. Actors are free to retrieve messages in FIFO order at any time. The second category is more strict and event-driven oriented. There is not a concept of storage. A signal is sent to the actor upon a respective message arrival. This signal is issued only once, and later retrieval of the message is not possible.

Erlang and the *C++ Actor Framework* (CAF) [Cha+13; CHS16] are examples of actor models implementing the first type of mailboxes. Akka [Nor+11], integrated into the Scala standard-library, implements the second.

CAF supports distributed applications with communication maintained by TCP sockets. The CAF architecture includes a run-time environment spawned per node. This environment is divided into three main cores: a message broker, a multithread scheduler and GPGPU wrapper.

The broker is used to intermediate communication between local and external actors. It is shared between local actors as a single transport channel. Multiplexers are used to provide mutual direct communication between actors. A single transport channel enables multiple actors to read-only access a common message without the costs of extra communication, copies or mutual exclusion requirements.

The scheduler organizes and fairly executes local actors concurrently through a pool of managed C++ standard threads.

The GPGPU wrapper over OpenCL allows integration of heterogeneous components and provides seamless communication between the host and accelerator devices. This provides the possibility to create actors from kernels and to integrate themselves with host actors.

Different from our implementation which we use a mailbox for each communication channel created between actors, an actor in CAF implements a single-reader-many-writer lock-free mailbox structure for parallel write access. Mailbox traversing and message matching for specific senders can be costly. Another drawback of the CAF mailbox concept comes from the possibility of arbitrary actors communicate to each other. Channels do not have to be explicit created between actors, resulting in a more unstructured and more unpredictable application.

We develop our library based on two subsequent *Asynchronous Partitioned Global Address Space* (APGAS) actor model implementations: the *actorX10* [Rol+16] programmed with X10, and a UPC++ [PBB19] version.

PGAS or APGAS are interesting models for distributed applications due to its implicit communication feature between ranks. Nodes provide public and private memory partitions. The public partition can be accessible globally (between ranks) through a concept of global pointers, while the private only internally.

The *actorX10* implementation benefits from X10 high-level PGAS abstractions. X10 enables implicit transfer of arbitrary objects and exposes global pointers to seamless manage local and remote actors. The language is implemented using one of two backends flavors: *managed*, which compiles into Java; and *native*, into C++. The first drawback of the *actorX10* implementation, is that the highest performance is only achieved over specific hardwares: clusters using the PAMI interconnect. MPI is used as a fallback solution [Tar+14]. Second, due to extra abstraction layers, native operation such as *vectorization* and integration with third-party libraries increases the programming burdensome.

UPC++ implementation seems to succeeded *actorX10*'s performance [PBB19]. It exploits the UPC++ thin-layer PGAS programming model. With lesser abstraction than it's antecessor *actorX10*, the library was developed with a finer-grained control over remote and local communication, higher processor optimization and better integration with external libraries and tools. The UPC++ uses the GASNet-EX network middle layer, which is compatible with Infiniband and Cray Aries interconnect fabrics [BH18]. A deeper overview of UPC++ is available on chapter 4.

3. Actor Model

The actor model is a conceptual model to deal with parallel and concurrent applications. It aims to abstract low-level communication and computation constructs. The model provides a secure context where programmers can confidently parallelize shared and distributed-memory applications without caveats of data race, data dependencies and scaling issues.

In a nutshell, the actor model resembles object-oriented programming models. Heterogeneous actors inherits from an abstract actor type, each which can contain its own computation logic, state and data-type definitions. Actors are closed entities, and accessing their state is only possible through message exchange. Actors have input and output ports, where the former receives messages while the latter sends messages. At a semantic level, an actor is a black-box finite state machine. It receives an input, checks its current state, realizes some action, transforms its state and then outputs some value. Actors are connected to each other and can be represented by a directed multi-graph, where vertices are actors and edges communication paths between themselves. The entire application workflow relies on actors exchanging messages with each other until all actors are in a final state.

A state-of-the-art actor model library should allow the programmer to abstractly express the computation and communication model at an actor level, while leaving the correctness and efficientness responsibility of communication and hardware resource requirements to itself. At execution time, the library should efficiently distribute actors across shared and distributed-memory processes or threads in order to maximize bandwidth and minimize communication latency. Furthermore, the library should ensure that specific hardwares are available for specific actors. In our first proposal, we discard different hardware specifications and focus only on the communication performance.

Our actor library follows the model introduced by Pöppl [PBB19], where formally, an actor's input port ip is connected to an actor's output port op through an unique channel. A channel $c_{n,t}$ is a FIFO queue with compile-time defined capacity n of tokens of type t. t is any primitive type or selected type from the standard library: *std::vector*, *std::list* or *std::array*.

An actor $a = (\mathrm{ID}, r, I, O, F, R) \in A$ is a tuple containing a unique name ID, a placement (e.g. a rank) r, the set of ip I, the set of op O, the set of functions F, and the

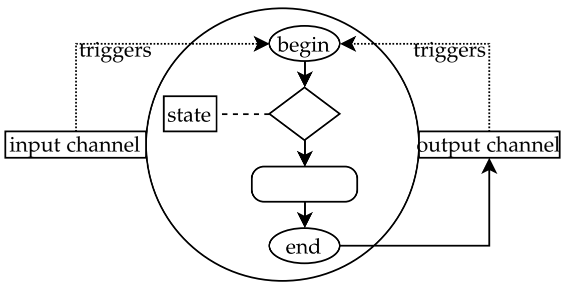


Figure 3.1.: An actor entity has its own private state, computation logic and input and output channels. It reassembles a state-machine that is triggered once one of its ports changes.

finite state machine R. F is a set of guard (accessor) and action (mutator) functions. Guards check for the actor's internal state, and for the state of their ports. Essentially, actors need to know the availability of its attached channels, and how to act based on its current state. An act occurs upon changes (triggers) onto the actor's ports: an ip receives a message or an op completes a send. The action is based on the actor's FSM $R = (Q, q, T)$. R has a set of states Q, a current state q and a set of transitions between states T triggered by tokens of type t.

Our current implementation constrains the number of actors and connections between actors to be specified at compile-time. Actors cannot instantiate other actors. Furthermore, actors migration is not supported in this current version. Actor migration and creation at run-time could be beneficial for load-balancing workload.

Figure 3.1 exemplify a typical actor workflow. A change in one of its ports triggers itself. The actor checks for its current state. If appropriate, it consumes a message from the input port, realize some computation and then outputs a message. As we will see in chapter 6 with respect to the internal implementation, messages can rest in the output channel for a while before arriving in the input port of another actor. This is due to the remote communication semantics.

Currently high-performance parallel applications relies on low-level communication constructs, such as explicit message exchange declarations under message-passing protocols, or remote procedure calls under remote direct memory access models. This requires a great precision from the programmer to ensure communication parameters are correct, communication buffers are disjoint, and the overall data distribution and path are clear and protected. Not only this scenario increases the difficult to create correct applications, but also decreases time-efficiency by replicating, debugging and profiling code. Furthermore, working with low-level constructs likely decreases the possibility of reusability, since they are targeted to specific models.

Actors are self-containing entities; state can only be accessed or mutated through explicit message exchange; and computation and communication model is upfront declared and disjointed. The model decreases the programmer's responsibility to protect against data race, even if the application resides in shared memory systems. It alleviates common performance problems such as cache false sharing and mutual exclusion constructs. It increases reliability by managing memory, and providing fault-tolerance. Furthermore, applications becomes easier to write. It becomes transparent to communicate locally or remotely with maximum efficiency. The abstract communication model ensure that scaling performance is only dependable on the given algorithm, and not on the communication layer.

In chapter 6 we discuss in-depth our current implementation using two-sided and one-sided MPI operations; in chapter 7 we present a mathematical computation model to describe water flow dynamics; and then we realize performance tests over this mathematical model using our implementation, a previous UPC++ implementation [PBB19] we have inherit from, and a classical bulk-synchronous parallelism (BSP) implementation model.

3.1. Example Application

In this section, we illustrate the usability of our actor library by creating and discussing a simple parallel producer-consumer application model. For an overview of the actor library execution flow, see figures 6.2 and 6.3. The following example application is hard-coded in figures 3.5, 3.6 and 3.7, and also summarized below:

1. *Application Initialization*
   The application creates two instances – ranks – of itself. The set of all instantiated ranks is called *world*. Ranks can be hosted within a common node, or in different nodes. Each rank creates an inherited Actor object, either a producer – *Chef* –, or a consumer – *Delivery*, according to its rank id.

2. *Actor Graph Initialization*
   Each rank adds its actor to a local *ActorGraph* object. ActorGraph holds information of all the actors within the world; and therefore, synchronization is required to ensure that local actors are replicated globally.

3. *Data Dependencies*
   A communication channel between Chef and Delivery is created, as shown on figure 3.2. Chef creates an output port named *outPortName*, while Delivery creates an input port *inPortName*. The connection statement *ActorGraph::connectPorts* is called by all ranks, and therefore, explicit synchronization is not necessary.

ActorGraph::connectPorts ensures that proper channel communication constructs are initialized according to the locality of the respective actors. In this example, both actors resides in different ranks, and therefore, the channel operations is modeled using MPI. If they would reside within the same rank, then shared-memory would be utilized for ports communication.

4. *Execution*
   ActorGraph::run starts the application execution. It calls each local actor's *void act()* method as long as the actor is in an active state. The calls are triggered by changes in the actor's port. In this example, every time a message is arrived at *inPortName*, *Delivery::act()* is called. And every time a send operation is completed at *outPortName*, *Chef::act()* is called. Note that in order to begin the application flow, Chef starts with an extra trigger.

   See figure 3.3 for Chef's execution model. Chef initializes a local *completedOrders* integer to 0. *completedOrders* is incremented on every *Chef::act()* call, and then sent to Delivery until its value reaches *maxNumberOrders*. When *maxNumberOrders* is reached, Chef sends a *endToken* and then becomes inactive.

   See figure 3.4 for Delivery's execution model. Delivery reads its input port. As long as the receiving numbers are not equal to *endToken*, Delivery remains in an active state. Otherwise, it becomes inactive.

5. *Finalization*
   When all local actors are in an inactive state, the rank waits in a *MPI_Barrier* for the world. When this barrier is released, the application finalizes. Figure 3.8 shows the overall execution output.

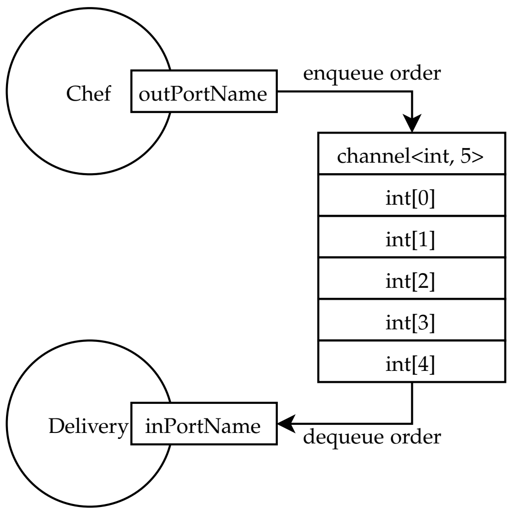


Figure 3.2.: Channel acts as a communication middle-layer between two actors. In this example, the channel has capacity of 5 integers; the Chef actor enqueues numbers to the channel, while Delivery dequeues from it. The process of checking the state of the channel, and mutating it, is done using input and output ports.

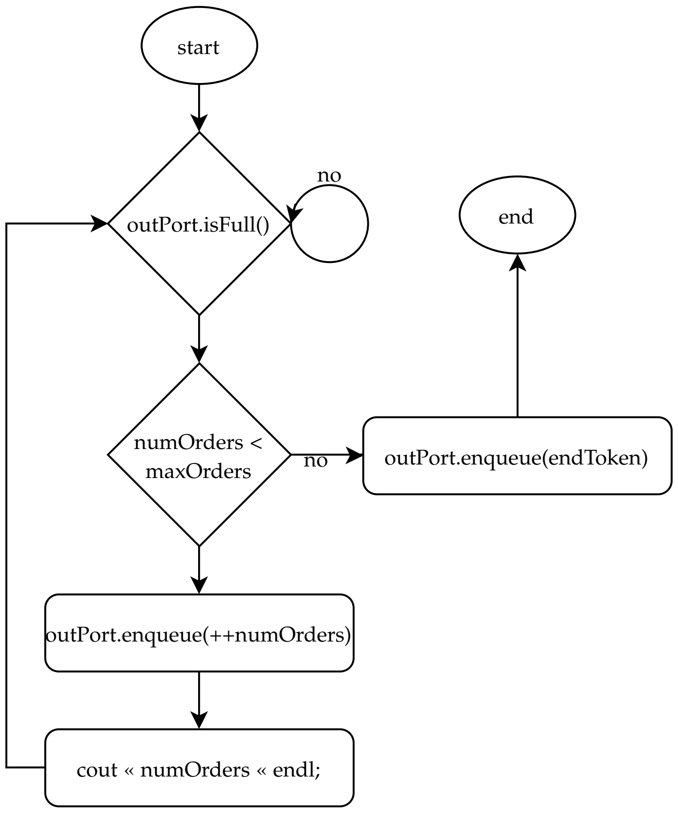


Figure 3.3.: Chef's workflow. Chef constantly produces numbers until they reach a maximum value. These numbers are sent to the actor Delivery until the limit is reached. Upon reaching the limit, Chef finishes its execution and sends a token in order to inform about its end of execution.

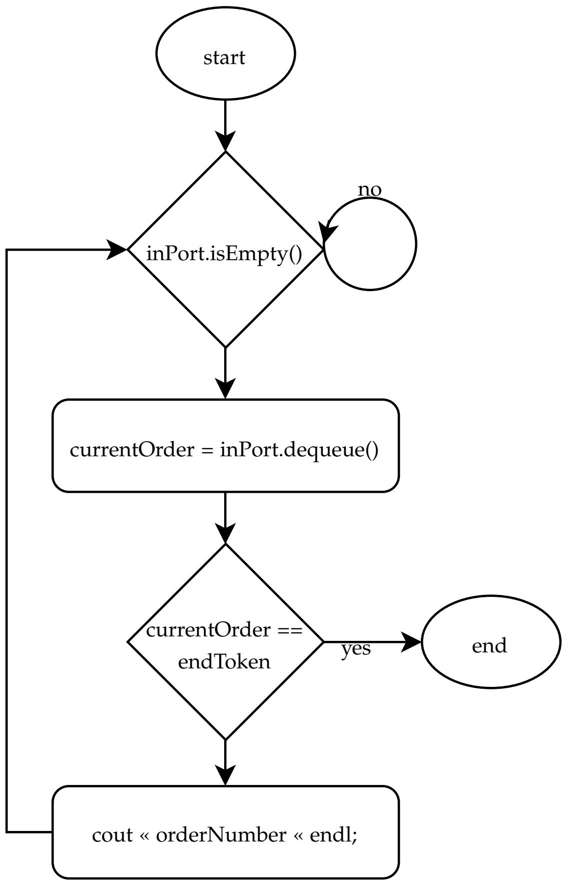


Figure 3.4.: The actor Delivery constantly checks the state of its input port; and upon receiving a new message, Delivery reads from it. If the receiving integer is not the final token, Delivery processes it; otherwise, Delivery finishes its execution.

```
int main(int argc, char **argv) {
  mpi::init();

  if (mpi::world() != 2) {
    std::cout << "This␣example␣requires␣2␣ranks!" << std::endl;
    return 0;
  }

  constexpr auto endToken = −1;
  constexpr auto maxNumberOfOrders = 10;

  std::unique_ptr<Actor> myActor;

  if (mpi::me() == 0)
    myActor = std::make_unique<Chef>(std::string("Chef"), endToken,
                                     maxNumberOfOrders);
  else
    myActor = std::make_unique<Delivery>(std::string("Delivery"), endToken);

  ActorGraph ag;

  ag.addActor(myActor.get());
  ag.synchronizeActors();

  ag.connectPorts<int, 5>(ag.getActor("Chef"), "outPortName",
                          ag.getActor("Delivery"), "inPortName");

  ag.run();

  mpi::finalize();
}
```

Figure 3.5.: Example application workflow using the actor library. It follows a SPMD programming model, where the application code is instantiated as many times as requested. MPI SPMD model injects information such as the number of instances and a unique instance identifier. In this example, we create two actors, one per instance. Each actor is added to an actor graph locally, which is then synchronized globally. Actor Chef is connected to Delivery, creating a communication channel between then. The order of connection matters, in this example, Chef's output port writes messages to Delivery's input port. Ports are used to abstract interaction within a channel. Figure 6.2 depicts the forementioned flow.

```
class Chef : public Actor {
public:
  Chef(std::string name, int endToken, int maxNumberOrders)
        : Actor(std::move(name)), endToken(endToken),
          maxNumberOrders(maxNumberOrders), completedOrders(0), op(nullptr) {}

  void act() {
    if (!op)
      op = getOutPort<int, 5>("outPortName");

    if (op−>freeCapacity() > 0) {
      if (completedOrders == maxNumberOrders) {
        op−>write(endToken);
        finish();
      } else {
        op−>write(++completedOrders);
        std::cout << "Order␣number␣" << completedOrders
                  << "␣is␣ready␣to␣be␣delivered." << std::endl;
      }
    }
  }

private:
  int endToken;
  int maxNumberOrders;
  int completedOrders;

  OutPort<int, 5> ∗op;
};
```

Figure 3.6.: Actor Chef's source code. Chef inherits from the abstract class Actor. The method *void act()* is overriding in order to provide the actor's finite-state machine logic. The private variables composes the actor's private state. Each time Chef's act() is called, it checks the output port for empty capacity. If the output port has free space, Chef writes a new integer to the actor Delivery. The integer is either an increasing number, or a token that represents a final state. Upon sending a final state token, Chef finishes its execution permanently, and informs Delivery about its new state.

```
class Delivery : public Actor {
public:
  Delivery(std::string name, int endToken)
      : Actor(std::move(name)), endToken(endToken), ip(nullptr) {}

  void act() {
    if (!ip)
      ip = getInPort<int, 5>("inPortName");

    if (ip->hasElement() > 0) {
      auto orderNumber = ip->read();
      if (orderNumber == endToken)
        finish();
      else
        std::cout << "To deliver order number " << orderNumber << "."
                  << std::endl;
    }
  }

private:
  int endToken;
  InPort<int, 5> *ip;
};
```

Figure 3.7.: Actor Delivery's source code. Similarly to Chef, Delivery inherits from Actor and overrides *void act()* method. Delivery checks for the availability of messages in its input port. If there are messages, Delivery reads it. If the value is a token representing a final value state, Delivery finishes its execution; otherwise, Delivery processes the value. In this case, just output a string.

```
<>:~/actor> mpiexec −n 2 build/orders_example
Order number 1 is ready to be delivered.
Order number 2 is ready to be delivered.
To deliver order number 1.
Order number 3 is ready to be delivered.
Order number 4 is ready to be delivered.
Order number 5 is ready to be delivered.
To deliver order number 2.
To deliver order number 3.
Order number 6 is ready to be delivered.
To deliver order number 4.
Order number 7 is ready to be delivered.
To deliver order number 5.
Order number 8 is ready to be delivered.
To deliver order number 6.
Order number 9 is ready to be delivered.
To deliver order number 7.
Order number 10 is ready to be delivered.
To deliver order number 8.
To deliver order number 9.
To deliver order number 10.
```

Figure 3.8.: Example application's execution output messages. Ranks run in parallel, and therefore, the order of messages should not be taken in consideration. The actor library provides a proper abstraction in order to alleviate the developer the necessity of connecting MPI ranks manually and pairing MPI communication operations. The library distribute ranks among different nodes or within shared nodes seamlessly. Communication between ranks within the same node is done through shared-memory, while communication between different nodes is done through explicit messaging-passing MPI operations.

4. UPC++

UPC++ is a library that supports *Partitioned Global Address Space* PGAS programming model in C++. It is implemented in a compiler-free approach where C++ templates and UPC++ run-time libraries are used to support PGAS semantics, static type checking and program analysis on shared and distributed-memory systems. UPC++ follows a SPMD execution model, where each independent execution unit is called a *thread*.

The basic semantics of UPC++ lies on dividing a thread address space into private and shared segments. The private is only accessible locally, while the shared by any other thread. UPC++ allows scalars and arrays data types to be shared in the form of *distributed objects*. Scalar objects lives on unique threads, while arrays objects may be divided and distributed among other threads. Shared memory is accessible through *global pointers*. These pointers encapsulates the thread id and the target's local memory address. Equivalent to MPI, it is implementation dependent whether arbitrary stack or regular allocated memory can be used as a shared memory space.

UPC++ remote operations are executed using one-sided *Remote Memory Access* (RMA): only one thread actively participate in the communication. The memory consistency model for these remote operations is relaxed. Different threads accessing a remote memory address can occur in any order while respecting data dependencies. The exception is for the same thread accessing the same memory address multiple times, where program's order is preserved. In contrast to UPC++, MPI one-sided communication ordering within an epoch applies only for accumulative operations. Subsequent remote operations occurring within the same epoch to a common target window is undefined.

UPC++ operations are asynchronous by default; their status are stored in *unordered event queues* maintained by each process' run-time environment. To advance the state of these operations, progress is required. UPC++ does not have a separate thread to ensure progress; instead, the user is responsible to make active calls to the library. There are two levels of progress: *user* and *internal*. User level progress advances the run-time's internal state and emit signals to the host application based on local events. These local events are usually callbacks from asynchronous operations issued by the user, or remotely injected remote procedure calls. Internal level progress only advances the run-time's internal state, and does not affect the local application. Dispatching remote procedure calls or progressing remote memory accesses are examples of the latter progress level.

Remote procedure calls (RPC) is a PGAS feature that does not exist nativity in MPI, but it is supported by UPC++. Its functionality is based on active messages exchange and on unordered event queues. When a node wishes to invoke a function into another node, a pointer to the respective function and its respective arguments are serialized into a continuous buffer. This buffer is then sent to the target node, which unpacks the data and enqueues it into the event queue. Return values in RPC is supported.

Remote communication in UPC++ is more relaxed than in MPI. In principle, it allows non-stride RMA transfers, different data types, and it implicit alleviates the necessity of creating synchronization epochs on memory windows in order to issue operations. UPC++ uses GASNet as its communication layer to handle shared and distributed memory transfers. GASNet is a communication library that supports PGAS run-time systems requirements. It exposes Remote Direct Memory Access (RDMA) capabilities of the network hardware and it is highly compatible with modern and popular network fabrics: Mellanox's Infiniband, Cray's Aries, Intel's Omni-Path and IBM's PAMI, to name a few [BH18].

5. MPI

Message Passing Interface (MPI) is a message passing programming model standard. MPI employs a Single Program Multiple Data (SPMD) execution model, which is a technique to achieve parallelism on distributed memory architectures.

In a SPMD execution model, applications are divided into multiple processes. These processes receives different inputs and produces different outputs, which are later gathered into the final output. Processes runs the same code base, and the data set is typically discretized and distributed among then. Processes do not share memory space, and therefore, communication is done through explicit message exchange.

MPI 3.1 – which at the time of this paper release – is the latest on production specification, it defines over 400 methods. These methods are related to the operations of handling data, point-to-point, collective, and one-sided communication, processes topology and communicators, and I/O.

In this chapter, we examine properties of the MPI standard that influence the efficiency of high-performance applications. Some of these properties are not defined in the standard itself, and thus, are implementation dependent.

5.1. Point-to-Point Communication

Point-to-point communication is the basic data exchange operation in MPI. Two processes are actively involved in this type of communication, the sender and the receiver. Both processes must explicitly call an appropriate method to execute the operation. The sender must issue a send call, and the receiver a receive call. The order these calls are issued, the size of the data and the MPI implementation will influence the efficiency and the performance of the communication.

Data to be transmitted is associate with the concept of an envelop. An envelop is an implementation dependent struct that typically consist of addressing and data information to be decoded by the receiver process. The information may include the data length, the source process identification number, the communicator and the identification tag.

The path the data takes and the amount of synchronization involved in the communication is high correlated to the efficiency of it. The path and the synchronization are influenced by the usage or not usage of intermediate buffers. Intermediate buffers correlates with the memory footprint, the blocking or non-blocking behavior, and the amount of synchronization of the communication operation.

5.1.1. Blocking and Non-blocking Communication

MPI defines the following send modes:

- buffered send *MPI_Bsend/Ibsend*
- synchronous send *MPI_Ssend/Issend*
- ready send *MPI_Rsend/Irsend*
- regular send *MPI_Send/ISend*

A buffered send copies the data to be transmitted into a intermediate buffer at the sender side, and then finishes.

A synchronous send completes only when the receiver has started the matching receiving call.

A ready send assumes that the receive call has been posted, allowing some implementations to do optimization, such as to reduce the amount of synchronization required. However, its completion does not indicate that the receiver call has been posted, only that the sending buffer can be reused.

It is implementation dependent whether a synchronous or ready send uses a intermediate buffer, or uses Remote Direct Memory Access techniques to reduce the communication latency.

A regular send method is dynamic and may use any sending mode technique described above. An important factor that may influence the choice for the sending mode is the workload.

There is only a single receive method, which is independent and can be called against any variance of the sending methods.

Non-blocking variances for the receiving and for all the sending methods are also available. Call to a non-blocking method returns immediately; however, it does not indicate its completion. There exist additional methods to check the status (*MPI_Test*) or to wait (*MPI_Wait*) for on-going operations.

Figure 5.1, 5.2 and 5.3 compares point-to-point communication methods discussed. The graphs evaluate the bandwidth for small, medium and large workloads respectively. The tests were executed using the Intel's MPI 2017 implementation, running on a cluster (more details on A.1) consisting of 28-way Intel Xeon E5-2697 nodes, connected by FDR14 Infiniband. Blocking communications are represented by continuous lines, while non-blocking by dashed lines. The color and marker for the blocking

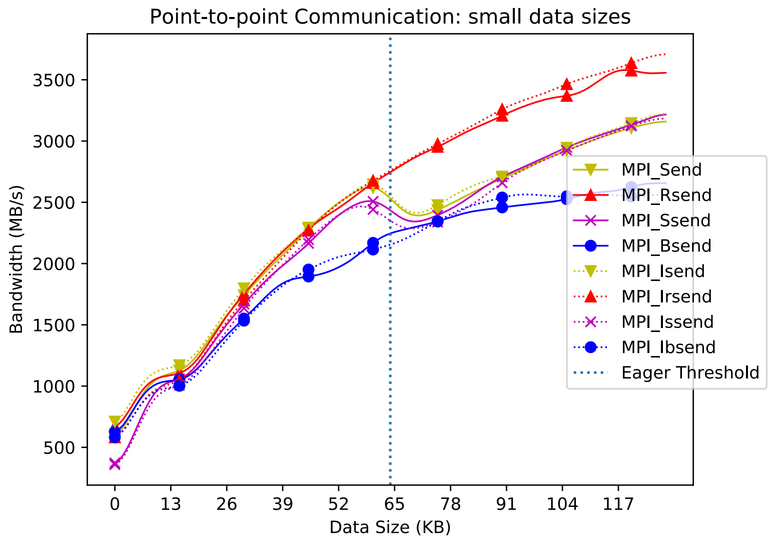


Figure 5.1.: Send performance for small workloads.

and non-blocking methods of the same type are preserved. The threshold where the implementation changes the message protocol (discussed in 5.1.3) from Eager to Rendezvous protocol is represented by a dashed-line labeled 'Eager Threshold'. One can adjust the threshold on the Intel implementation by setting the environment variable *I_MPI_EAGER_THRESHOLD* [Car18].

The tests were setup similarly. In all the cases, the receiver matches all the send requests with a non-blocking receiving method (*MPI_Irecv*), and then waits for all (*MPI_Waitall*). In the case the sender performs a blocking communication, the sender iterate over a loop. In each iteration, the sender calls the respective blocking send method (regular, synchronous, ready synchronous, buffered) with a workload of increasing size. In the non-blocking case, the sender calls a non-blocking send method and immediately waits for it (*MPI_Wait*). It is expected that the blocking and non-blocking version of a communication operation perform similarly. To test the non-blocking performance over the blocking, we have setup a different test described in (5.4).

In all test cases (small, medium and large workloads), we see that buffered sends (*MPI_Ibsend*, *MPI_Bsend*) are the slowest. It is worthwhile to note, that not only performance is degraded due to extra copies, but also the memory footprint is grown proportionally. Depending on the communication pattern and bandwidth, one can experience memory exhaustion and crashes if outgoing bandwidth is not bounded by the allocated buffer.

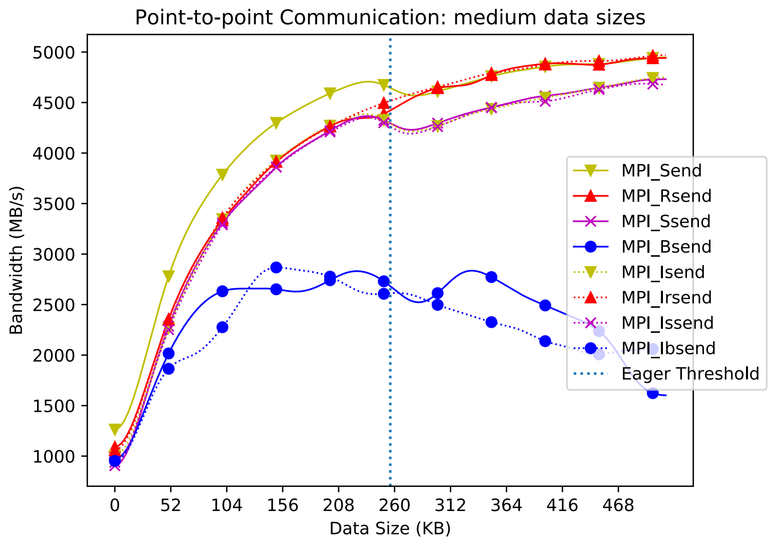


Figure 5.2.: Send performance for medium workloads.

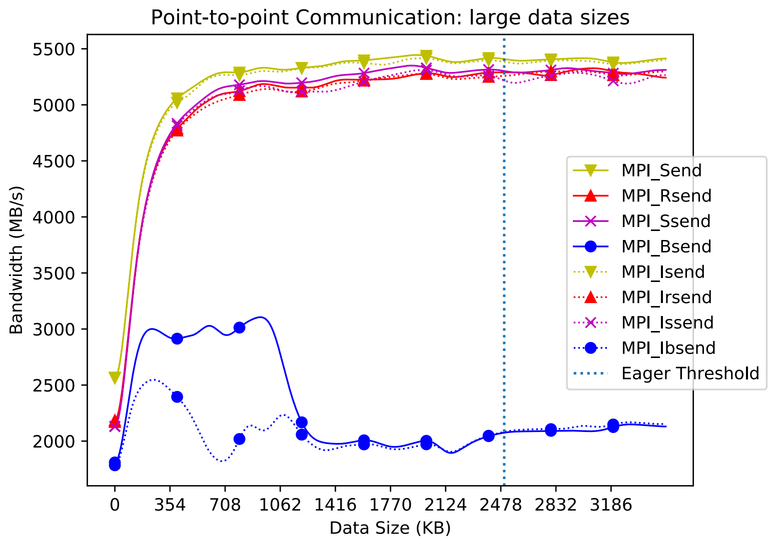


Figure 5.3.: Send performance for large workloads.

For small and medium workloads, it becomes clear that ready sends (*MPI_Rsend*, *MPI_Irsend*) performs better than their sibling synchronous sends (*MPI_Ssend*, *MPI_Issend*). One can also note that they scale better than regular sends (*MPI_Send*, *MPI_ISend*) upon protocol change.

For large workloads, all send methods, besides the buffered ones, balances out within the bandwidth limit.

5.1.2. Message Buffering

MPI processes have three abstract concepts of internal queue buffers: a send buffer, a receive buffer and an unexpected message buffer. These buffers and their existences is implementation dependent, and if existent, may promote non-blocking behavior.

Send and receive buffers are related to the send and receive operations, and resides at the sender and at the receiver side, respectively. MPI may use the send and the receive buffer as intermediate buffers during the communication process. To assure the existence of a send buffer, the user can call MPI methods (*MPI_Buffer_attach/detach*) to manage the existence and length of it, and then use the buffered send mode to communicate with other processes.

The unexpected message buffer is allocated at the receiver side. A process may use its buffer to receive messages addressed to itself without having previously posted any receive call. Further receive calls would first probe the unexpected receive buffer, and then return immediately if a matched message would be found.

5.1.3. Message Passing Protocols

Message passing protocols describes how a message is transmitted from the sender to the receiver buffer. The MPI standard does not specify different protocols, however, implementations such as IBM's and Intel's do provide two different ones: Rendezvous, and Eager.

Under the Eager protocol, the sender assumes the receiver is ready to receive the message. The communication is initiated and the message arrives either at the receiver buffer, or at the unexpected message buffer. A regular send method would therefore not block.

Communication using the Rendezvous protocol on the another hand, avoids the usage of intermediate buffers at the cost of an extra synchronization step. First, there exists a handshake between the sender and the receiver, and only then, the message is transmitted. It is implementation dependent whether the communication will involve intermediate buffers, or if RDMA (Remote Direct Memory Access) techniques is utilized.

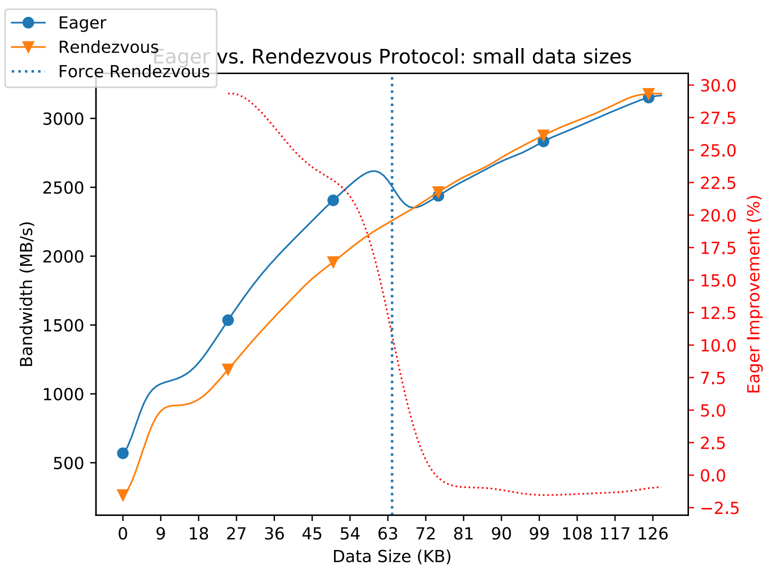


Figure 5.4.: Bandwidth and message protocol performance test for small workloads.

Intel's and IBM's MPI implementation uses the Eager protocol as long as the workload is below a specific threshold, and then it switches to the Rendezvous. The user can configure the threshold by setting specific environment variables; *MP_EAGER_LIMIT* [IBM] and *I_MPI_EAGER_THRESHOLD* [Car18] for IBM and Intel respectively.

We have executed performance (bandwidth) tests to evaluate the Eager and the Rendezvous protocols. The tests were executed using the Intel's MPI 2017 implementation, running on a cluster (more details on A.1) consisting of 28-way Intel Xeon E5-2697 nodes, connected by FDR14 Infiniband.

We setup the tests as follow: the sender and the receiver iterate over a loop calling a blocking regular send (*MPI_Send*) and a receive (*MPI_Recv*) method, respectively. On each iteration, the workload is increased. We manage the Eager threshold to see how the bandwidth is affected.

Figures 5.4, 5.5 and 5.6 shows the results for small, medium and large workloads respectively. We note 30 percent bandwidth increase for small and medium workload for the Eager over the Rendezvous protocol.

While the bandwidth itself is a important factor to decide upon different protocol implementation and utilization, it is worthwhile noting that other factors such as memory footprint and algorithmic synchronization steps differs between the different protocols.

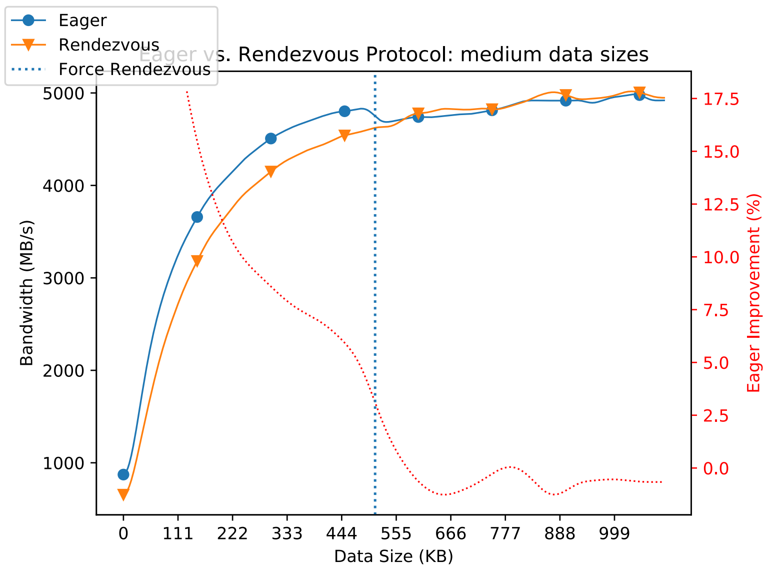


Figure 5.5.: Bandwidth and message protocol performance test for medium workloads.

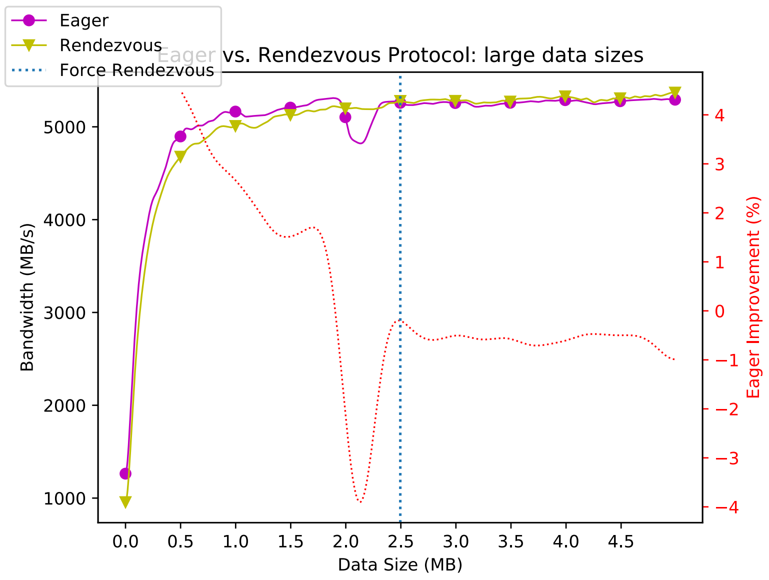


Figure 5.6.: Bandwidth and message protocol performance test for large workloads.

5.2. One-Sided Communication

One-sided communication, also know as Remote Memory Access (RMA), decouples communication to synchronization. A process exposes part of its main memory (*windows*) and during specific time intervals (*epochs*), other processes may modify or read memory addresses within the window. The process that originates the transfer is called *origin*, while the targeted process is called *target*.

High-performance interconnect providers, such as Infiniband and Ethernet (using RoCE), allows MPI vendors to implement one-sided communication using Remote Direct Memory Access (RDMA). RDMA bypasses the operating system and accesses the main memory directly using special network ports. The result is zero CPU utilization at the target side to participate in the communication, and less synchronisation and intermediate buffers utilization.

Aside from the potential of lower messaging passing latencies using RDMA, one-sided communication is free of communication matching, intermediate buffers usages, and synchronization protocols. For applications where communication patterns is irregular (contrary to classic BSP models), one-sided communication allows the programmer to express algorithms more freely, without unnecessary broadcasting of communication parameters and synchronization primitives usages.

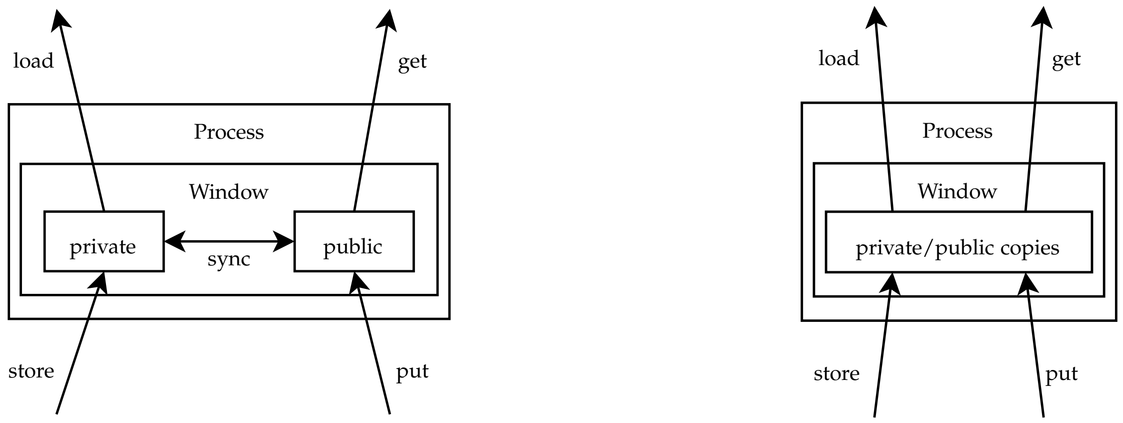
One-sided communication arises few concerns, mainly: when a origin process is allowed to access a local (origin and target are the same) or remote target window; when the modified data is available for the target process; and data race and coherence for non-overlapping (consecutive addresses) or overlapping memory accesses.

MPI defines three synchronization models (fence, start-port-wait-complete, lock-/unlock) to establish epochs and two memory models (unified, separate) to establish coherence.

5.2.1. Windows and Memory Models

A window is a continuous RAM block exposed by a process in order to enable RDMA from other processes. Depending on the vendor's implementation, the user can provided previously allocated memory to use as a window, or more recommended for performance and compatibility, allocate memory using specialized MPI operations (*MPI_Alloc_mem, MPI_Win_allocate*). Specialized MPI allocation operations may contains information about the allocated memory segment, and memory boundary alignments in order to boost access performance and decrease memory footprint from translation tables.

A window is logically separated into two different copies: a public and a private copy. Local accesses (loads and stores) interacts with the private copy; while public



(a) Separate Model.

(b) Unified Model.

Figure 5.7.: MPI Remote Memory Access Memory Models.

accesses (*MPI_Put, MPI_Get, MPI_Accumulate*) with the public copy. As represented by figure 5.7

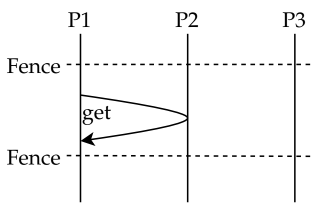
To support different implementations and architectures, two memory models (unified, separate) is implemented. These models defines coherence and synchronization logic between the two copies. In the separate model, coherence is managed at software level. It is the developer's responsibility to call appropriate MPI synchronization operations to ensure proper private and public copy synchronization. For the unified model, coherence is supported at hardware level. Local or remote updates to the window is automatically synchronized without explicit synchronization operations.

Memory models also defines how concurrent overlapping or non-overlapping memory access within a window behaves. Usually, concurrent accesses (reads or writes) to overlapping or non-overlapping memory under the separate model is constrained. In the unified model, concurrent accesses are valid, but the result is undefined if updates (writes) are performed to the same location.

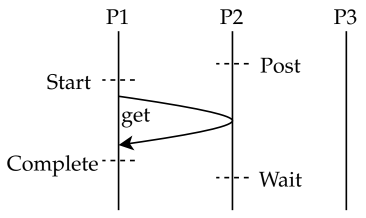
Fortran developers need to be aware of further coherence scenarios, that independently of the memory model. A successfully modified access can be overwritten by a variable from the register, while a read access may return an unsynchronized register variable (more on Registers and Compiler Optimizations section in [For15]).

5.2.2. Synchronization

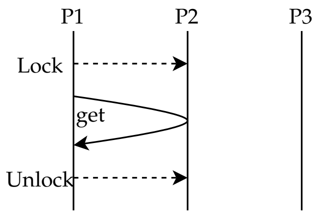
One-sided communication from one process to another has to be done through specific time intervals, called epochs. The start and the end of an epoch is defined by synchronization operations.



(a) Active Target.



(b) Generalized Active Target.



(c) Passive Target.

Figure 5.8.: MPI Remote Memory Access Synchronization Modes.

MPI defines two synchronization modes: active and passive. In the active mode, both the origin and the target processes are involved. Synchronization blocks until all the involved processes are synchronized. In the passive mode, the target process is not required to participate in the synchronization. The origin process locks the target window and then communicate to it, without the target participation.

There are two flavors of target synchronization mode: fence and generalized. In the fence mode, figure 5.8a, all processes from a collective group has to synchronize, even if a process does not communicate or it is not the communication target. In the generalized mode, figure 5.8b, only a subset of processed participate in the synchronization. A target process exposes its window to a subset of processes, which can then communicate to it.

Passive target synchronization, figure 5.8c, does not require the participation of the target process. An origin process requests a lock on the target window, performs the communication and then unlocks it.

Typically, one-sided operations are non-blocking; the developer issue operations within an epoch, which its execution is postponed to the end of an epoch. Order of operations is not preserved. For example, the developer should not assume that a *MPI_Get* issued before a *MPI_Put* within the same epoch will happen in the predefined order. The exception lies on accumulative operations, where ordering is preserved.

5.3. Progress

Non-blocking methods, in principle, allows applications to perform computation while communication is outstanding. In practice, however, asynchronous behavior is only achieved through user intervention and careful architectural design.

Networking hardware may impose buffer limitations at the outgoing port. Message protocols may require additional synchronization steps. Message buffering may introduce extra copying and probing. Either way, a communication operation may delay its execution, or divide itself into multiple steps. Due to the intrinsic nature of non-blocking methods to avoid iteration and recursion, and due to the default single-threaded MPI specification, additional calls to the library is required to progress remaining or delayed steps.

Progression is not only relevant for non-blocking communication operations, but also for the overall functioning of the library as a whole, which needs to constantly interact with the kernel and often direct with the hardware to realize external dependent operations, as sending and receiving messages.

There exists two concepts used for progressing the internal state of a MPI application: interrupt and polling. Polling is a top-down approach where the software regularly queries and trigger communication events. Interrupt, oppositely, relies on the network hardware providing interrupt signals upon message events in order to asynchronously trigger software methods. Cray [Pri+12] and IBM systems [Kum+08; KSK13] are examples of systems that supports interrupt-based approach.

From a software architectural perspective, there exist two traditional polling approaches to cope with internal progression: and algorithmic and a threaded.

In the algorithmic approach, the user would emplace tests (*MPI_Test*) operations alongside the computational code in order to regularly progress.

A threaded approach relies on using extra threads to progress, and it is even supported in some implementations, such as in the Intel's, by setting the *I_MPI_ASYNC_PROGRESS* environment flag [Int19]. It does however, introduce two possible drawbacks: core over-provisioning and internal thread safety guarantee overheads.

Thread safety guarantee would be required if there exists multiple threads in the application, and more than one thread calls MPI methods. First, the MPI standard does not require implementation to provide multithread support; second, multithread support entails performance overhead due to the introduction of locks and lock-free atomic algorithm within the library.

Addressing the necessity of thread safety guarantee by ensuring only one thread calls the library is feasible; however, it would still introduce complexities and overheads in order to design an application where multiple threads delegates communication requests to the progress thread.

Core over-provisioning on the another hand is less exposed for optimization. Assuming each MPI process needs an extra thread to progress, the application would double its number of threads requirement. Not only the number of threads poses a possible performance drawback, but also load balancing work and communication across the communication and the computation threads becomes challenging and application specific.

On the actor library implementation discussed in this paper, we delegate MPI operations to a single thread using lock-free algorithms. We iterate to progress and complete communication operations, and upon state changes on outstanding messages or new arrivals, we dispatch OpenMP tasks that acts upon the respective actor.

5.4. Hybrid Programming and Multithreaded MPI

The availability of many cores and specialized processing units within a node, together with the necessity to provide distributed load balance and high-computing performance, drives some applications to be built upon hybrid programming models.

Hybrid models provides two levels of parallelism: coarse- and fine-grained. While there exists no concrete definition with respect to the boundaries between these two levels, they essentially deal with the size of the workload to be distributed among parallel tasks. The former tries to distribute considerable sized workloads, where the communication itself is not strictly bounded by the network latency, and thus, can be justified by a following considerably busy computation. The latter is usually applied to shared memory environments, where communication is likely to be cheaper and data can be further distributed among lightweight parallel processing tasks (eg. shared-memory programming on GPUs).

A typical hybrid programming model on modern high-performance applications – where a system may consist of multiple interconnected many-core nodes – is to orchestrate parallelism across multiple nodes (possible multiple NUMA domains) using processes (eg. MPI), and then to further distributed workload within a node by threading (eg. OpenMP).

Processes provides high scalable coarse-grained parallelism. First, parallelism is at a process scale. Memory is private, which decreases share-memory latencies typically related to data race and false sharing. Second, scalability is usually better at a process scale, where adding more nodes may yield better performance than to adding more cores to a chip.

Fine-grained parallelism aims to increase CPU utilization by distributing the workload even further. Algorithms that somewhat have structured computational patterns and low data dependencies benefits the most. At this scale, parallelism can occur at different hardware levels, such as diving workload among the cores in the chip, utilizing specialized processing units (eg. GPUs), accessing data and exploiting the cache efficiently, utilizing vector processing units and hiding pipeline stalls with hyper-threading. Note that the last may not be in full control of the developer.

When combining coarse- and fine-grained parallelism, further performance queries may arise. Specifically, how to use MPI on a multithreaded application.

Although the MPI standard does not require implementations to provide thread safety, some such as Intel's and IBM's may define provide four:

1. *MPI_THREAD_SINGLE*

Only one thread may call MPI.

2. *MPI_THREAD_FUNNELED*
   Multithreaded environment where only one thread may call MPI.

3. *MPI_THREAD_SERIALIZED*
   Multithreaded environment where multiple threads may call MPI; however, at distinct times.

4. *MPI_THREAD_MULTIPLE*
   Multithreaded environment where multiple threads may call MPI, at any time.

To evaluate performance overheads when developing multithreaded MPI applications (*MPI_THREAD_MULTIPLE*), we perform a series of tests introduced by [TG09]. The tests where executed on 28-way Intel Xeon E5-2697 nodes connected by FDR14 Infiniband (more information on A.1). To remove outliers, we repeatedly executed the tests on about 30 combinations of node pairing and took the median results. The MPI implementation is from Intel 2017. The following tests are performed:

1. Thread safety overhead
   What are the overheads of thread safety.

2. Cumulative bandwidth
   How bandwidth varies between process to process, and thread to thread communication.

3. Cumulative latency
   How latency varies between process to process, and thread to thread communication.

4. Message rate
   How many messages per second can be sent between processes and threads.

5. Asynchronous progress
   How MPI non-blocking communication behaves.

5.4.1. Thread Safety Overhead

This test measure the overhead from thread safety guarantees. We launch two MPI processes distributed over two distinct nodes and then measure the latency for message exchange (ping-pong) between then.

We compare the ping-pong latency over two scenarios. On the first scenario, the application initializes MPI with no safety requirement (*MPI_THREAD_SINGLE*); and on

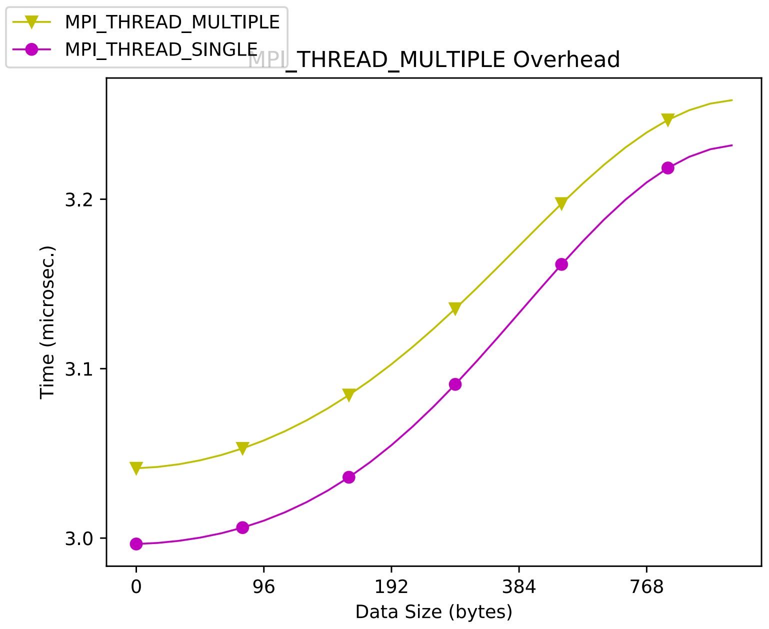


Figure 5.9.: Hybrid MPI Thread safety overhead.

the second, the application requires thread safety (*MPI_THREAD_MULTIPLE*). Apart from the initialization method, all the remaining setup and application remains the same.

Figure 5.9 shows the results. We note a negligible overhead of less than 0.05 μs.

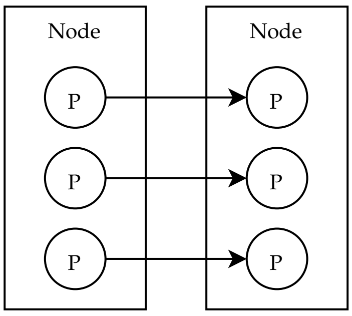
5.4.2. Concurrent Bandwidth

The concurrent bandwidth test measures the cumulative bandwidth of a node.

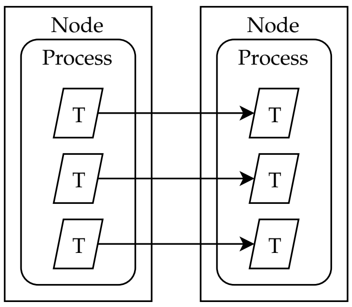
Figure 5.10 abstracts the communication scenarios. The tests runs over two nodes. The first scenario distribute paired processes between distinct nodes: one process is the sender, while the other is the receiver. The second scenario uses threads within a process instead of processes. A MPI process is launch on each node. The processes launches a group of threads that communicates with the other process' threads.

Each node on the targeted cluster has 28 cores. We perform the tests between 1 to 1 process/thread, 7-7, 14-14 and 28-28.

Figure 5.11 shows the results. The individual bandwidth of each process/thread within a node is summed to derive the total cumulative bandwidth. We note a significant bandwidth decline (up to about 40%) when communication is done through threads, instead of processes. The decline persists even when no locking contention



(a) Process to process.



(b) Thread to thread.

Figure 5.10.: Hybrid MPI process to process, and thread to thread communication.

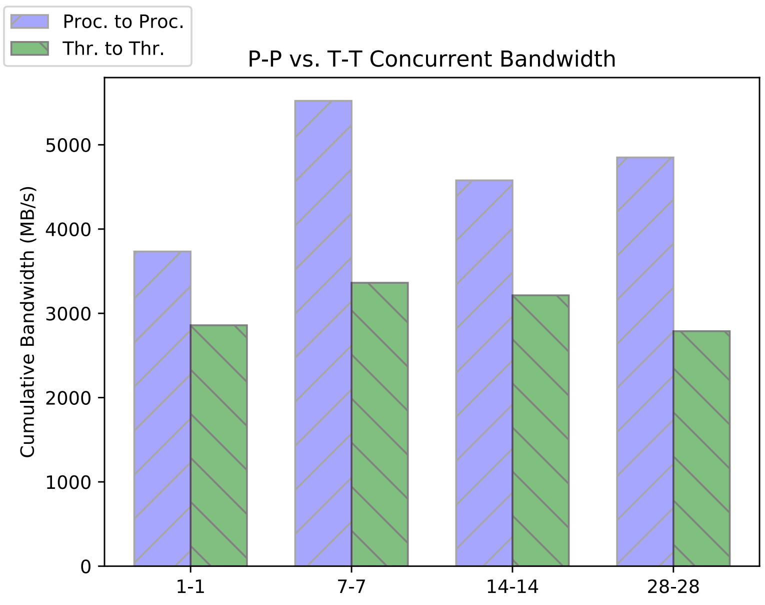


Figure 5.11.: Hybrid MPI Concurrent bandwidth.

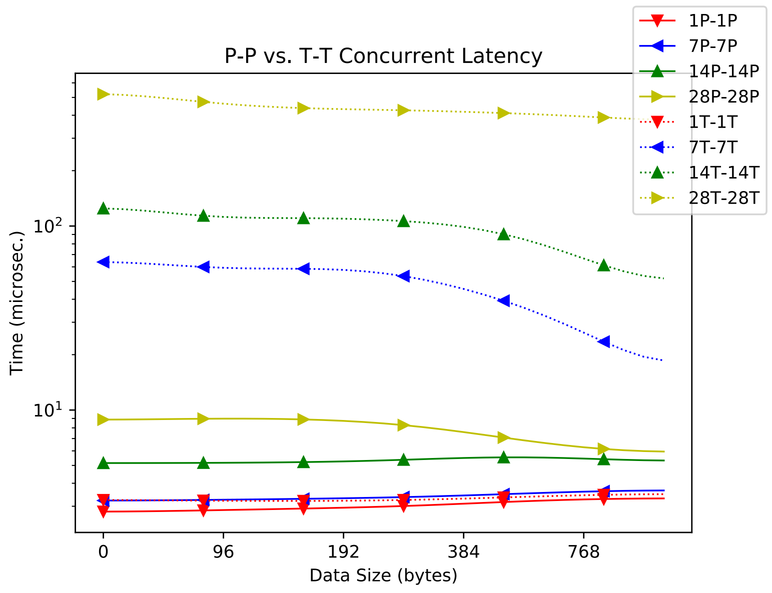


Figure 5.12.: Hybrid MPI Concurrent latency.

exists (1-1 threads). Ideally, we would expected to see that, for at least in the 1-1 configuration, the processes and threaded scenarios would perform similarly.

5.4.3. Concurrent Latency

This test is similar to the concurrent bandwidth described above. It shares the same communication paradigm (processes to processes, threads to threads). The difference is on the size of the messages. We focus on the latency for short individual messages, instead of the overall node bandwidth.

Figure 5.12 shows the result. Differently from the bandwidth test, we note that for the 1-1 configuration, there exists not a significant thread safety overhead. However, by increasing the number of threads and thus the contention, we see a latency increase of almost two orders of magnitude.

5.4.4. Message Rate

This test measures how many zero byte messages a node can send per second. The setup is similar to the concurrent bandwidth and latency test above; however the size of the messages is zero, and the receiver process only replies a send operation upon every 256 completed sends, differently from regular ping-pong exchange. We sum the

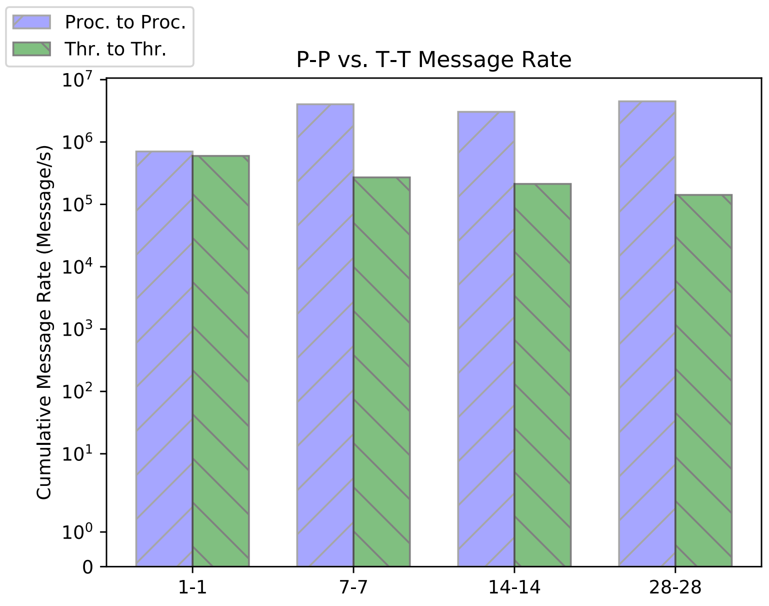


Figure 5.13.: Hybrid MPI Message rate (message/s per node).

message rate of each process/thread in order to have the cumulative rate per node.

The results are shown in figure 5.13. We note that the performance for the 1-1 process/thread configuration is similar. However, by increasing the number of threads, the message rate is at least one order of magnitude lower.

5.4.5. Asynchronous Progress

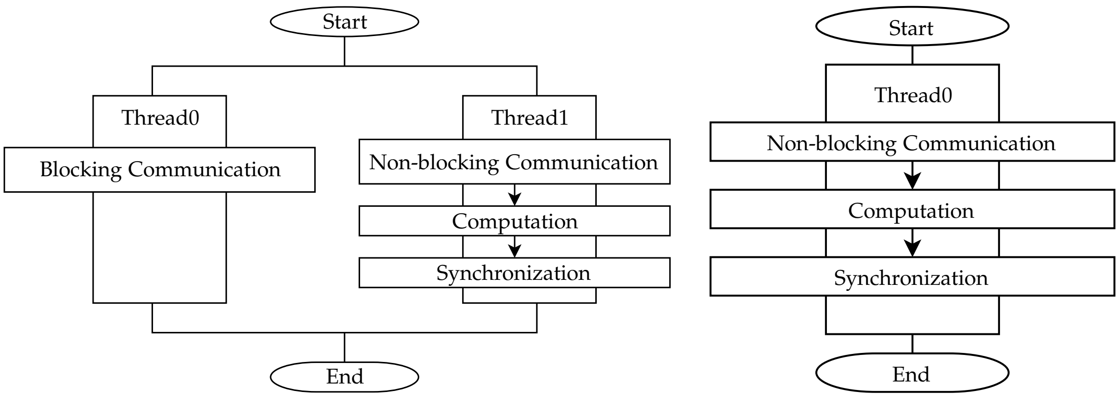
This test measures the asynchronous progress for non-blocking communication.

As discussed in section 5.3, it could be the case that non-blocking MPI communication may progress only if the user interacts with the implementation's progress engine indirectly.

The test setups a classic bulk-synchronous parallelism model. The workload is divided among MPI processes using a 2D grid representation. Each process computes a grid cell, and exchange the borders with its four nearest neighbors (top, right, bottom, left). More specifically, we perform three sequential steps: 1. communication, 2. computation, and then 3. synchronization, [repeat].

A non-overlap and an overlap scenario are evaluated. Figure 5.14 abstracts these scenarios.

In the non-overlap scenario, the process calls, for each neighbor, a non-blocking send (*MPI_Isend*) and a non-blocking receive (*MPI_Irecv*); do the grid computation and then



(a) Overlap scenario. (b) Non-overlap scenario

Figure 5.14.: Hybrid MPI Overlap and non-overlap scenarios for asynchronous progress test.

waits (*MPI_Waitall*) for the non-blocking operations to complete.

The overlap scenario executes exactly all the same steps. However, before doing the communication step, the process launches a single thread that calls a blocking receive operation (*MPI_Recv*). The blocking method essentially spins the progress engine and progress the non-blocking communication.

We execute the tests across two nodes. Increasing the number of processes per node from 7 to 28 (maximum number of cores).

Figure 5.15 shows the results. We do not see noticeable difference in performance between the two scenarios. We note that in the overlap scenario, a whole thread or core is busy progressing communication, and this may introduce performance concerns that do not justify a non-clear performance improvement. More research has to be done to fully understand the progression of non-blocking communication.

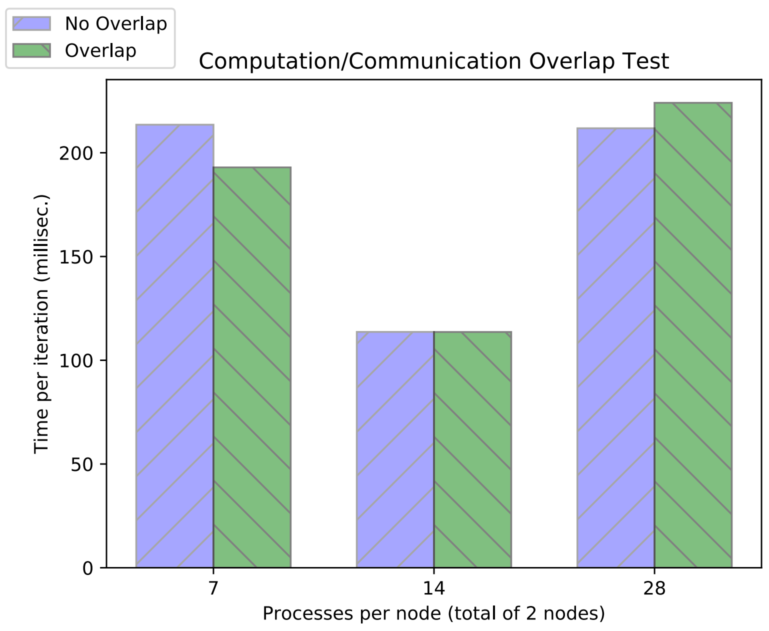


Figure 5.15.: Bulk-synchronous parallelism time per iteration based on overlap/non-overlap asynchronous progress.

6. Implementation

The actor programming model abstracts parallelism by confining units of computation within entities. These entities are interconnected using message passing communication protocols. An entity (actor) behaves similarly to a state machine. It has an internal state, an algorithm, and an input and an output channel (possible multiple channels). A typical life cycle of an actor would be to receive an input through a message, perform some computation and then send the output to another actor via its output channel.

Actors are independent from each other. Direct access to an actor state is not permitted. The input channel is used to receive messages, while the output channel (which is the input channel of another actor) to send messages. The algorithm within an actor can behave differently depending on its current state; therefore, repeatable iterations with the same input may result in different outputs.

A group of actors can be represented by a directed multi-graph, see figure 6.1. Each actor is a vertex, and connections (communication) between actors, edges. As in the directed multi-graph, connections between actors it is not necessary to be symmetric, or in other words, channels does not have to be reciprocal. Furthermore, actors can have an arbitrary number of input and output channels, that could be equally numbered or not.

Our current implementation is based on a single program multiple data (SPMD) model, typically used in MPI applications (chapter 5). A single application is written

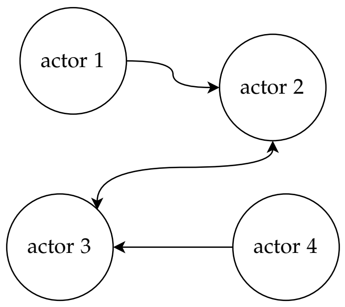


Figure 6.1.: Actor graph where each vertex represents an actor, and edges, communication. The pointing arrows on the edges depicts the direction of communication.

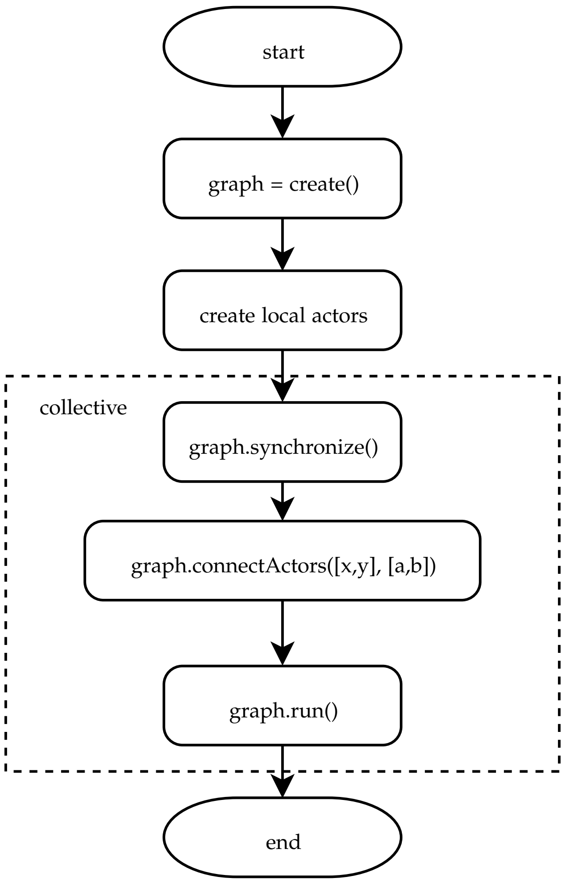


Figure 6.2.: The abstract actor model SPMD application flow.

and copies distributed across multiple nodes (computers). Each rank (application instance) is provided with a unique rank id, and a world (set of ranks) count. The application queries the rank ids and the world count in order to segregate the parallel logic for each of its instances. Multiple ranks can reside within a single node, or to be distributed across different ones.

The developer creates and connects different actors. The implementation distributes the actors between nodes and within nodes. The communication between actors within the same node is done through shared memory. For actors in different nodes, MPI is used.

Figure 6.2 demonstrates the typical actor model application flow. An actor graph and local actors are instantiated locally; followed by collective operations that has to be executed by all ranks. These operations synchronizes local actors across the world, create communication channels between actors and then run the application, respectively.

The application runs until the last actor is finished with its work. An actor remains idle until a change in one of its ports occurs: either a message sent has left its output channel, or a new message has arrived. Upon port's change, the actor executes its predefined algorithm in a parallel OpenMP task. Multiple actors within a rank can run in parallel; however, only a single task instance of the same actor is allowed to run concurrently.

Figure 6.3 depicts the application life cycle. Each rank iterates over a list of local actors until all actors are finished. First, progress within all actor's communication channels is enforced (chapter 5.3). If there has been any change in one of the actor's communication channel, either a successfully message dispatched or a new message arrived, then a new OpenMP task is dispatched to execute the actor's algorithm. If an actor changes its own internal state to terminal one, then it is removed from the iteration list.

When actors communicates with each other, three types of information is necessary: the data itself, a notification when a new message has arrived, and a notification when a message has been read and thus a new free space in the channel is available. An origin actor cannot send a message if the target channel is full; the target actor has to inform the origin when a space is freed. And the target actor has to be aware when there a new message arrives; the origin actor has to inform the target when a new message has been sent.

We implement the actor model using two MPI communication models. One using point-to-point operations, and the other with one-sided communication. The overall architecture and communication scheme are the same for both implementations, the only difference is in the communication operation types and the required synchronization steps.

First, on 6.1, we analyze how intra- and inter-node communication scheme works: how we handle shared-memory data-races and multi-thread MPI interaction. And then on sections 6.2 and 6.3 we describe how data is transferred and synchronization maintained using two-sided and one-sided communication operations, respectively.

6.1. Communication Scheme

The current implementation supports distribution of actors between different nodes and also within a single node. Communication within the same node is done through shared memory, and for actors between different nodes, through MPI. Two concerns arises: data-race in shared memory, and multi thread MPI calls.

For actors communicating through shared memory, data-race is a concern. We

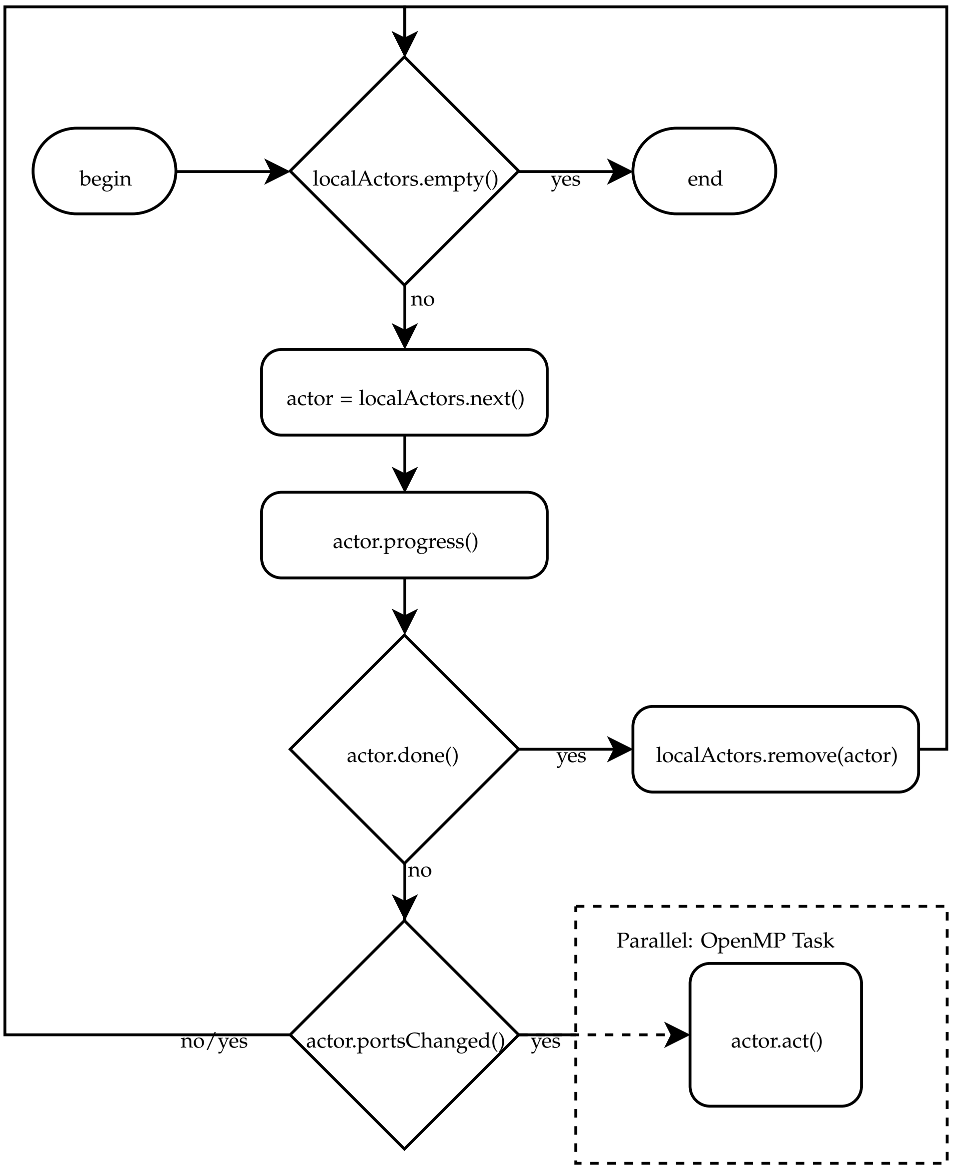


Figure 6.3.: The application running model. It runs until the last remaining actor is finished with work. MPI progress and port change checking is done in the main thread, while work dispatched on OpenMP tasks.

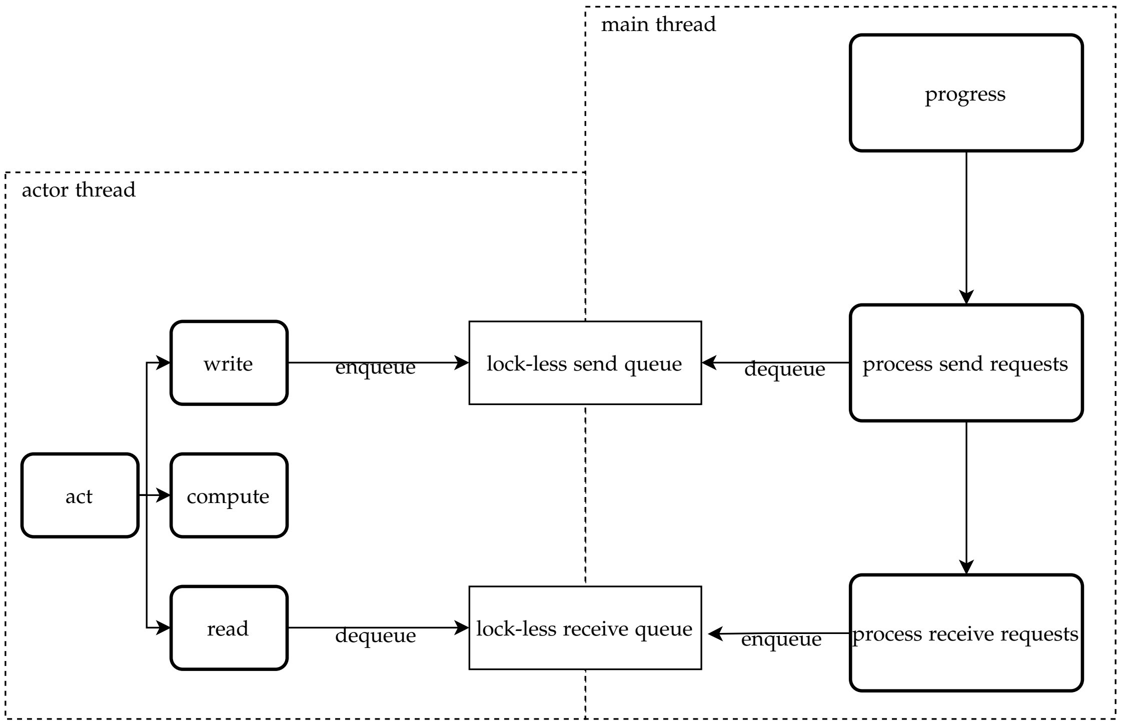


Figure 6.4.: Communication schema where MPI operations are funneled to the main thread.

implement lock-less input and output channels to protect against it. For communication between nodes, lock-less channel structures is unnecessary, and we discard atomic instructions to alleviate CPU time.

The next concern is with respect to intra-node actors calling MPI operations concurrently. On chapter 5.4 we analyze performance implications for using MPI from multiple threads; and conclude that while there is not a major latency increase or single operation overheads, the bandwidth is significant deprecated. Therefore, we funnel MPI calls to a single thread.

The implementation avoids multiple actors calling MPI operations by separating the communication layer to the logic layer. See figure 6.4. An actor communicates by interacting with lock-less queues responsible for funneling communication requests to the main thread. The main thread is then responsible to handle these requests, either by sending or receiving messages, and to progress (chapter 5.3) the internals.

Another factor that favors funneled MPI operations is to save CPU time. Consider an alternative solution, where we devote a thread for each actor, and each actor is allowed to call MPI concurrently. It is likely that actors will be busy waiting for channels

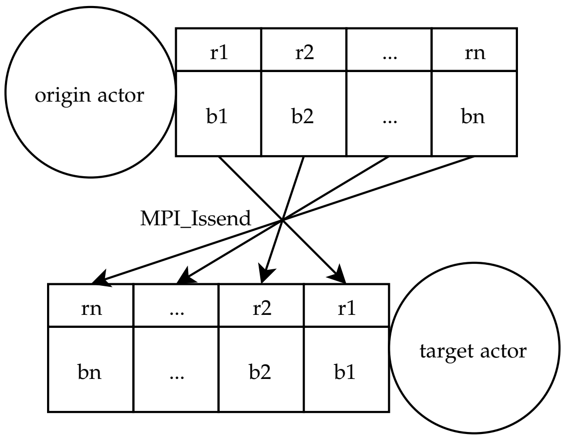


Figure 6.5.: Point-to-point inter-node actors channel communication scheme. The circles represents actors. The rectangle broken into two parts. The communication channel spaces at the bottom, and the respective MPI communication request handler at the top.

changes and to ensure MPI progress. Cheap operations that could possible wastes a whole core. In contrast, the current funneled solution uses only a single thread for MPI asynchronous progress and to check for channel updates. The remaining system threads are saved for fine-grained parallelism within actor's computation steps.

6.2. Point-To-Point Communication

The point-to-point implementation relies on two-sided MPI operations to transfer data and to synchronize the channels statuses. We use MPI_Issend (chapter 5.1.1) which is a non-blocking synchronous send operation. Non-blocking because it returns immediately. Synchronous because its completion status indicates that the receiver has received the message.

Synchronization is necessary for the origin actor to inform the target when a new message is available; and for target to inform the origin the availability of its communication channel.

The communication schema is depict on figure 6.5. Both the origin and the target actors have an extra array of the size equals to the communication channel number of spaces. Each entry in the array is related to a channel space.

The target actor opens receive requests (MPI_Irecv) for each freed buffer space; these buffers is then used by MPI to receive the communication data. The actor periodically checks for the statuses of the receive requests. A request completion implies that a new message is available at the respective request's buffer. And when a message has been read, thus a communication buffer space has been freed, a new receive request is created.

The origin actor uses MPI_Issend to transmit messages. When an origin actor wishes to send a message to the target, it checks the availability of the target buffer. If it has free space, the origin data is copied to a output buffer space and a new MPI_Issend request is created. The origin can query the amount of outstanding send requests in order to check for the availability of the target buffer. If a send requests has been completed, it means that the target space has been freed, and a new request can be send.

6.3. One-Sided Communication

This second implementation of the communication layer relies on one-sided MPI operations. One-sided communication reduces communication latencies by exploiting RMA characteristics (chapter 5.2). However, the extra synchronization steps may hide the extra gains in performance.

We use MPI_Put to transfer data from the origin to the target channel, and MPI_Send containing zero-byte sized messages for synchronization (notification signals). The communication schema is depict on figure 6.6.

For each of its input channels, a target actor allocates memory using MPI_Alloc_mem and exposes it via MPI_Win_create – see section 5.2.1. Two helper variables are maintained for each channel. One tracks the number of available messages (x: available messages), and the other tracks the last read channel space index plus one (y: next available). x is increased by notification signals sent by the origin actor. To read a message, the target actor checks x. If it is greater than zero, then the target actor: reads the input channel at location y; circularly increases y; decreases x; and then notifies the origin actor.

The origin actor does not allocate special memory (MPI_Alloc_mem) for its output channels. It does, however, maintains regular allocated output channels in order to save outstanding sending data. Similarly to the target's channels, the origin's channels maintains two helper variables. One tracks the target's channel free space count (a: target free space count), while the other tracks the target's next free space index (b: target buff next displacement). a is increased by notification signals sent by the target actor. To write a message, the origin actor checks a. If it is greater than zero: decreases a; puts the message at the target's buffer space index b; circularly increases b; and then notifies the target actor.

Note that MPI_Put operations issued by an origin target lies within a passive synchro-

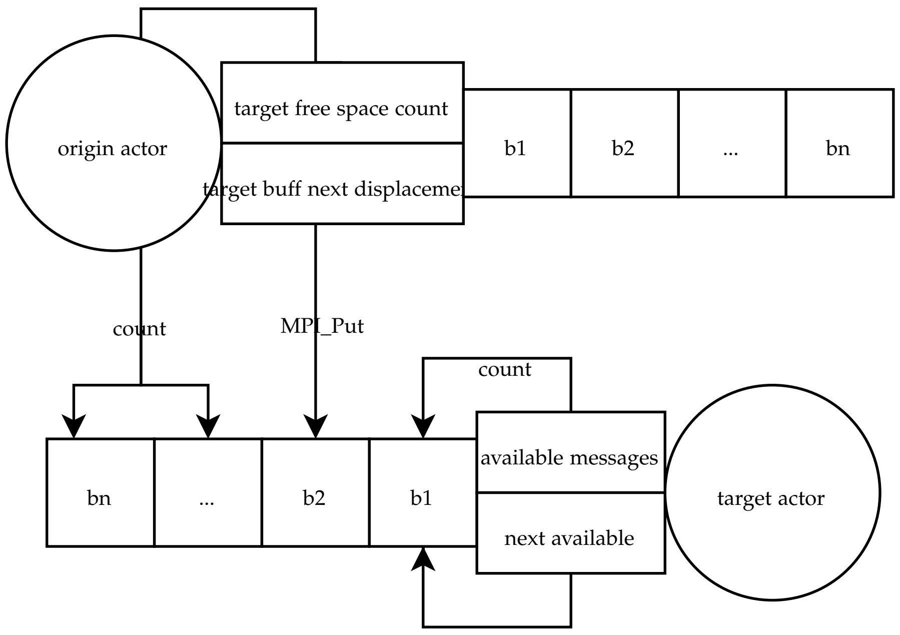


Figure 6.6.: One-sided inter-node actors channel communication scheme.

nization epoch (chapter 5.2.2). MPI_Put can occur at any time. The target participation in the communication is not required under the passive synchronization model.

6.4. Low-level Constructs and Semantics

In this section we describe in depth performance and usability properties of our library. We discuss how actors are managed in a remote and distributed memory context, and how we handle different data types and manage memory in the communication channels.

6.4.1. Actors

Every rank holds a *std::map<std::string, Actor*>* mapping remote and local actors to its object representation. The abstract *Actor* class defines a *std::string* name, *int* rank id, *std::map<std::string, Port*>* list of *Port*s, and a pure virtual *void act()* method[1]. *void act()* is the execution entry-point of an actor – see figure 6.3. An *Actor* inherits *RemoteActor* and user-defined *Actor* classes – see figure 6.7. *RemoteActor* implements a *throw* in *void*

[1]Classes that contains pure virtual methods cannot be instantiated.

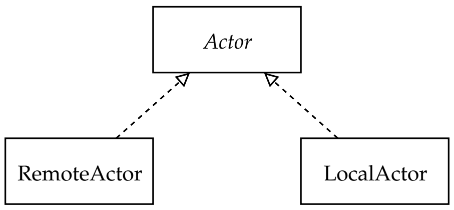


Figure 6.7.: Actor inheritance scheme according to their relative rank placement.

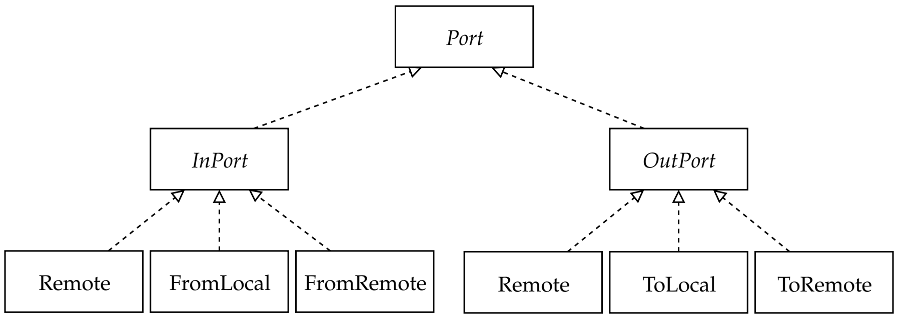


Figure 6.8.: Port inheritance scheme according to their relative rank placement.

act(). This enforce that a rank do not execute remote actors. Inherited user-defined *Actor* classes should implement *void act()* with their desired logic.

Communication is done between *pairing ports*: an input and an output port. The abstract *Port* class defines a *std::string* name, *enum* type, and a pure virtual *void progress()* method. *void progress()* funnels remote communication requests from the actor's thread to the main's one – see figure 6.1. From *Port*, classes *InputPort* and *OutputPort* are inherited. They specify pure virtual methods for reading and writing data, respectively, as well as other accessors. *InputPort* inherits *InputPortFromLocal* and *InputPortFromRemote* to receive from local and remote ports respectively. Similarly, *OutputPort* inherits *OutputPortToLocal* and *OutputPortToRemote* to send to local and remote ports – see figure 6.8. When an actor progress – see figure 6.3 –, *void progress()* is called for each of its ports. If a port communicates remotely, then an appropriate communication logic is implemented. Otherwise, *void progress()* is no-op.

Actors are either local or remote from a rank perspective. Communication between local actors is done through method calls: pairing ports hold references to each-other. Communication between local and remote actors differs between the two-sided and one-sided implementation. The former requires rank ids and tag values to be used in MPI two-sided operations. Pairing ports uses their hashed names as tag values and their hosting actor rank ids. In the one-sided implementation, tag values are unnecessary. When ports are instantiated, a *MPI_Win* object is collectively created for each pairing port – see section 5.2.1 on RMA windows. This object is then used as a communication parameter for MPI one-sided operations.

Actors, ports and their respective inheritance representations are instantiated locally in each rank during the graph synchronization and actor connection collective calls – see figure 6.2 for the application flow. In the current implementation, the number and distribution of actors are statically defined. The user instantiate actors at compile-time. During program execution, neither the library or existent actors can instantiate new actors. Actor distribution is internally done by graph partitioning libraries during the application initialization. Actors cannot migrate between ranks.

6.4.2. Channels

Channels $c_{n,t}$ have compile-time capacity c and data type t defined. t must be a built-in type or *std::array* and *std::vector* from the standard library. Both the one-sided and two-sided implementation uses asynchronous communication operations, and therefore, it has to ensure that *outstanding data* is unmodified until the send request is completed. For communication between local actors, this is trivial: communication operations are atomic and happens through shared memory. But for communication between local and remote actors, the channel has to hold a copy of the data – an *outstanding copy*. We choose to pre-allocate a continuous buffer – *outstanding buffer* – during ports initialization in order to avoid costs of memory operations during algorithmic run-time. These buffers has size of $size = c * sizeof(t)$ bytes[2]. There are performance and design implications for using different types of t. In the following subsections 6.4.2 and 6.4.3, we explain how the two-sided and one-sided implementation utilizes outstanding buffers to ensure outstanding data is immutable, and how they try to avoid costs of unnecessary copies. We denote built-in and *std::array* types as *copyable types*, and *std::vector* as *moveable types*.

Two-sided Implementation

When an origin local actor sends a message to a remote actor, a send request is created. The request idles in a lock-less FIFO structure until being dequeued by the main thread during actor progression – see figure 6.3. The structure is an abstraction on top of the outstanding buffer; it ensures mutual exclusion between the actor's thread and the main one's.

The main sending concern is how to transfer ownership of the data from the send call to the outstanding buffer. The library goal is to increase performance by avoiding copies, ensure correctness by not breaking data dependencies and to not leak memory.

[2]sizeof() returns the size in bytes of the given type or object.

For copyable types, a copy to the outstanding buffer is inevitable; the buffer cannot hold a reference to a stack variable and do not take ownership of user-allocated memory. The interface ensures that ownership of heap memory is not transferred to the library by exposing only constant references as parameter types. For moveable types, copies may be alleviated by using C++ 11 move semantics whenever it is suitable; thus, the sending structure ownership can be transferred from the caller to the outstanding buffer. Because moveable types from the standard-library handles its allocated memory, it does not expose risks of memory leaks.

The receiving process is similar to the sending one with respect to decoupling communication between threads and managing the outstanding buffer. *InPortFromRemote* provides a lock-less structure wrapper on top of its outstanding buffer – see figure 6.5. The buffer is associated with an array of receive requests. *void InPortFromRemote::progress()* ensures that these requests are always awaiting to receive data. If a request finishes, a pointer to the respective buffer space is inserted in a queue – *completed queue* – which is later consumed by read calls. Upon consuming these buffer spaces, their pairing receive requests are re-dispatched.

Copies from the receiving outstanding buffer to the user space is inevitable, since references to the buffer would entail data corruption upon receiving new data. The main concern lies on the return types from read calls. When a user connects actors, he or she specifies the data type and the channel capacity as template parameters. These parameters types are then used to wrap accessor methods – see code snippet 6.9. Copyable types are fixed in size at declaration-time and thus has enough buffer space to hold copies from the outstanding buffer. Moveable types, however, has to be dynamic resized by specific methods. Resizing a buffer could be costly, since it may entail memory allocation. However, the main concern is how to specify the required receiving length in the resize method. A dynamic and internal solution would use *MPI_Probe* before issuing a receive request. *MPI_Probe* would inspect the unexpected receive queue for messages and if encountered, could check for their sizes. However, this operation is costly and could entail in extra synchronization steps for large data sizes under the MPI rendezvous protocol. Our solution is to require an extra parameter binding during actors connection which specifies the communication length – see code snipped 6.10.

6.4.3. One-sided Implementation

One-sided operations require memory to be explicit exposed in a RMA context. The exposure operation is collective, which implies synchronization. During our implementation, we have tended to favor local CPU utilization over communication constructs, since the latter's latency tends to be orders of magnitude higher than the former's. For

```
using namespace std;

ActorGraph ag;
// initialize and synchronize actors

ag.connectPorts<array<int, 10>, /*channelSize*/>(actor1, "outPortName", a2, "inPortName");

// below is executed from the actor's void act() method
array<int, 10> read(){
  array<int, 10> data;
  // fill data from from completed queue's buffer
  return data;
}
```

Figure 6.9.: Template specification during ports creation is used to deduct types during accessor and mutator method calls from an actor's communication port.

```
using namespace std;
ActorGraph ag;

// initialize and synchronize actors

ag.connectPorts<vector<int>, /*channelSize*/>(actor1, "outPortName", a2, "inPortName", 10);
// port::communicationLength = 10

// below is executed from the actor's void act() method
vector<int> read(){
  vector<int> data;
  data.resize(this->communicationLength);
  // fill data from from completed queue's buffer
  return data;
}
```

Figure 6.10.: Moveable types requires extra connection parameters and may entail dynamic allocations during read accesses.

this reason, dynamic addresses on the outstanding buffer is not possible, which results in the impossibility of using move semantics – copies are enforced for all data types.

There are significant differences in the outstanding buffer's wrapper as it does not manages pairing send or receive requests. Instead of using requests as direct communication layers, *void InPortFromRemote::progress()* and *void OutPortToRemote::progress()* one-sided variation handles specific two-sided byte-sized operations – *signals*. They receive and send signals, and specifically for output ports, also handle remote writes (*MPI_Put*).

Input ports interpret incoming signals as arrival of messages: a circular index increases for each signal received. This index is used to create pointers to outstanding buffer spaces; these pointers are enqueued in the completed queue. When a message is read by the client, and thus the respective buffer space is freed, the input port sends a signal to its pairing port to inform that a new space is available.

Similar to its pairing input port's circular index, the output port holds a circular count to track the availability of free spaces in the target outstanding buffer. The port uses this index to correctly do remote memory writes (*MPI_Put*). Upon a successful write, a signal is sent to the input port to inform the message arrival.

7. Shallow Water Equations

To test the performance of our implementation, we use it within *Pond* from Pöppl [PBB19]. Pond is a shallow water proxy application that uses an UPC++ actor library to model its computation and communication requirements. We replace the original underlying UPC++ actor library by ours and then execute weak and strong scaling tests.

7.1. Case Study

The shallow water equations *SWE* describe the behavior of a volume over time given an initial condition, such as sea floor elevation caused by an underwater tsunami – see figure 7.1. They are derived from depth-integrating the third-dimension of Navier–Stokes equations [LGB11], as the vertical flow is considerable smaller than the horizontal one. SWE is shown below, where x and y refer to the coordinates within the domain, t refers to time, h to the height of the water column, hu and hv to the momenta in two-dimensions, g to the gravitational constant, and $S(x, y, t)$ as a model source term (tsunami, flooding, atmospheric flows, ...).

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

Given the SWE equations, Pond, following the approach of [BB12; LGB11], does a Cartesian grid Finite Volume discretization in order to approximate the wave propagation form within each grid cell i, j, described by:

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

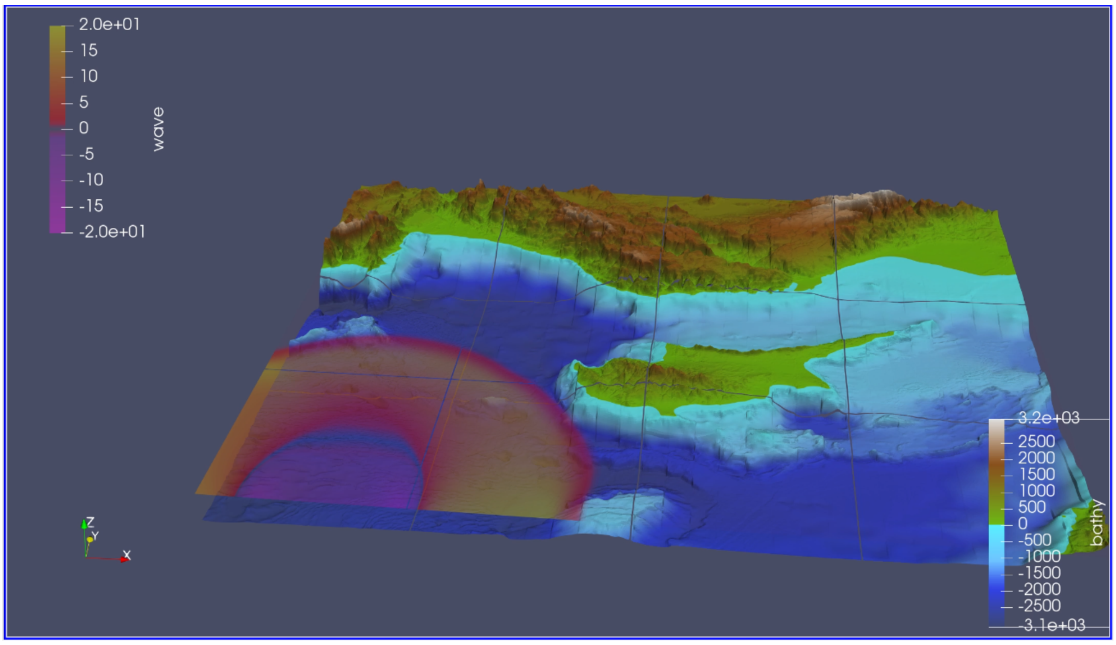
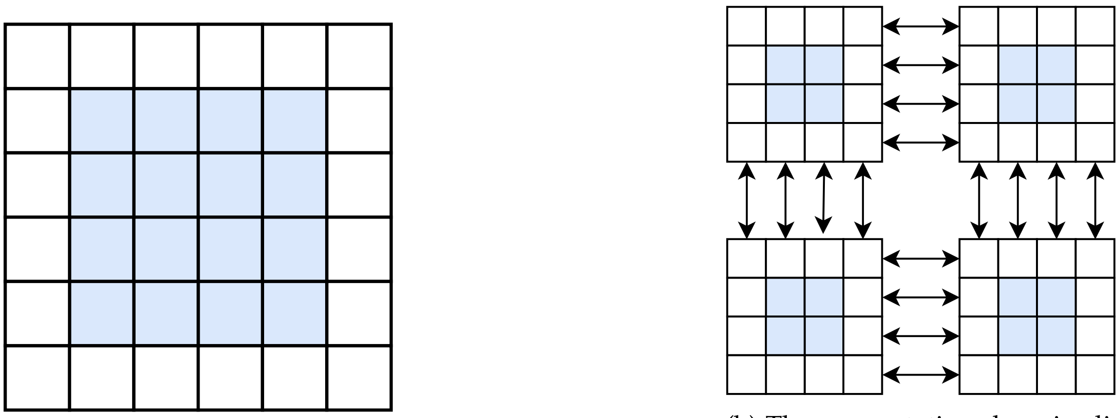


Figure 7.1.: The Shallow Water Equations 3D Visualization. An initial water volume elevation is triggered at the bottom-left corner of the domain. It causes a wave that is propagated throughout the entire computation domain, during a specified time interval. The domain is divided into squared patches, each which runs in parallel. Adjacent patches shares data dependencies. Data is communicated either through shared memory or through MPI operations, according to the location of adjacent patches.



(a) The computation domain.

(b) The computation domain divided into patches.

Figure 7.2.: The simulation domain is divided into patches. The domain and its respective patches include a surrounding extra cell-sized layer to represented boundary conditions. Boundaries are used to represent adjacent patches values, or, if located at the edge of the computation domain, to describe walls or undisturbed flows.

The equation iteratively calculates the unknowns $Q_{i,j}^{(n)} = [h_{i,j}^{(n)}, (hu)_{i,j}^{(n)}, (hv)_{i,j}^{(n)}, b_{i,j}^{(n)}]$ based on an initial state t_0, a time-step Δt and a cell size Δx and Δy. $\mathcal{A}^{\pm}\Delta Q_{i\pm\frac{1}{1},j}^{(n)}$ and $\mathcal{B}^{\pm}\Delta Q_{i,j\pm\frac{1}{1}}^{(n)}$ represents the solutions of the Riemann problems calculated by the HLLE Riemann solver [HLL97], which essentially describes the amount of water transported per time – *fluxes* –, across a cell's left/right and top/bottom boundaries, respectively.

7.2. Implementation

The simulation domain is represented by a Cartesian grid. Each cell carries a constant sea floor bottom height b, and the unknowns water height h and momenta hu and hv for the directions x and y respectively. The domain – figure 7.2a – is divided into sub-domains called *patches* – figure 7.2b. Surrounding the domain and each of its patches, there exists a cell-sized layer – *ghost layer* – to represent the behavior of flow across boundaries.

There are three types of boundaries conditions: 1. Undisturbed flows, where adjacent cells shares the same values. 2. Wall, where flows do not propagate. 3. Connect, where adjacent patches are connected and their respective boundary values are dependent to each other.

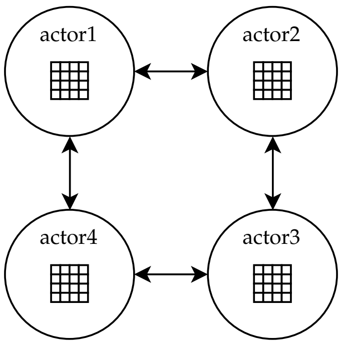


Figure 7.3.: Patches are encapsulated into actors. Each actor is responsible for the local computation of its patch, and to communicate boundary values to its logically adjacent actors.

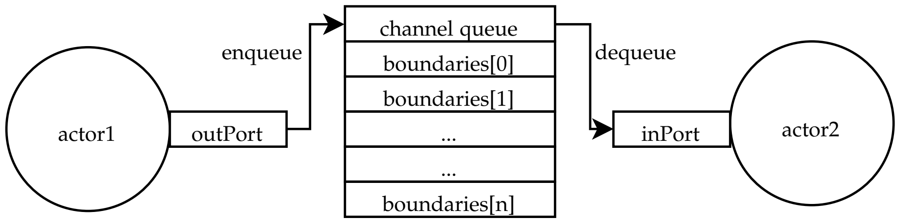


Figure 7.4.: Actors communicate fixed-sized continuous arrays of data, where each array carries the boundary values of its adjacent patch.

The simulation runs in time steps. For each time step: 1. Boundary cells have their values set according to the respective boundary condition, either locally computed, for the undisturbed or wall scenario; or based on their adjacent neighbor values, for the connect scenario. 2. Fluxes for all cells within a patch are calculated, and then 3. Unknowns for each cell is updated based on time-step Δt.

We share the Pond implementation which divides the simulation domain into quadratic patches, and then assign each patch to an actor – see figure 7.3. Actors sharing adjacent patches are connected to each other. Connections are delineated by channels holding fixed sized continuous arrays (eg. $std :: array$s or $std :: vector$s). Ports utilize these channels to exchange ghost layers between actors – see figure 7.4.

To run the application in a distributed and shared-memory system, we specify the domain size in number of grid cells, the size of patches and how many cores per patch should be reserved – see appendix A.2 for an batch example. Then, given the set of application constrains and the system's resources availability (nodes and cores), METIS [KK98] – a graph partitioning library – distributes actors (each carrying a patch) across a set of nodes. METIS ensures that the distribution is optimal in order to minimize remote

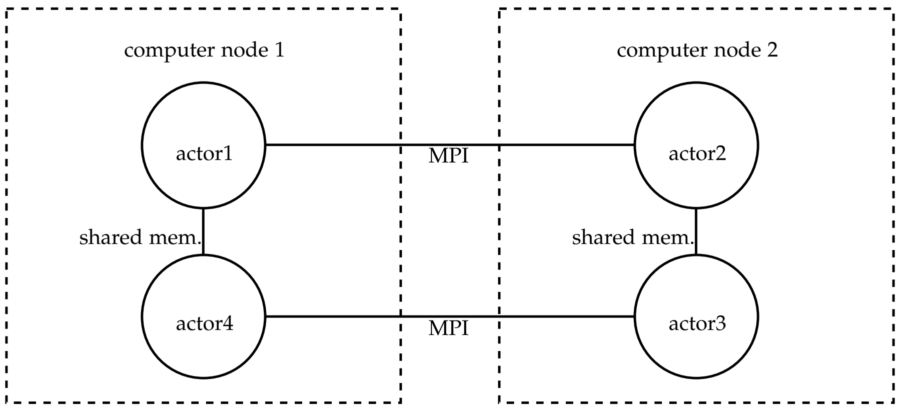


Figure 7.5.: During application initialization, actors are distributed across shared and distributed nodes. Actors within a common node communicate through shared-memory, while pairing actors from external nodes utilize MPI to exchange boundary values.

communication between adjacent actors. Actors sharing the same node communicates through shared memory, while communication between actors in different nodes utilize MPI operations – see figure 7.5.

The number of cores per actor is used for fine-grained parallelism, where fluxes computation are vectorized[1] and parallelized using OpenMP for-loop pragmas[2].

We note that the application is not perfectly load-balanced due to time dependencies during the fluxes calculation; thus, there may exists patches that are triggered frequently, not so frequently or even idle. This drawback can result in a underutilized system. Future work could be done to introduce dynamic actor instantiation or actor migration, in order to provide more computational resources for hot patches.

7.3. Performance Evaluation

To measure the performance of our library, we run four SWE implementations and compare the number of floating point operation per second for each of them. We do weak and strong scaling tests, where in the former, the size of the domain is increased

[1]Only supported on processors that provides SIMD instructions.

[2]OpenMP is a shared-memory parallelization standard for C/C++ and Fortran. OpenMP manages a thread pool in order to save their management costs. For parallel computation, OpenMP provides work-sharing constructs that transforms sequential code into parallelized ones, or handles independent units of work – *tasks*. For-loop pragmas defines work-sharing constructs where a loop's index set is divided into subsets, and then dispatched in parallel throughout the pre-allocated threads.

along with the resources (nodes and cores); while in the latter, the domain size is fixed while the resources are increased.

The local computation model in all implementations are the same. They are vectorized and parallelized using OpenMP abstractions. The main difference lies on the communication scheme. Basically, communication needs to exchange data, and to ensure that the communication partner is aware of the availability of the new data.

We evaluate the following SWE implementations:

1. *MPI Two-Sided*
   Our MPI two-sided implementation uses MPI non-blocking synchronous *MPI_Issend* operations. *MPI_Issend* guarantees that data exchange and synchronization happens upon single pairing sending and receiving calls. To further increase performance, some MPI implementations ensures that synchronous and ready operations transmit data using remote memory access techniques.

2. *MPI One-Sided*
   Our MPI one-sided implementation uses MPI remote memory access *MPI_Put* operation to transmit data, and two-sided *MPI_Send* operation to support synchronization. RMA explores hardware features to ensure the lowest communication latency and the avoid intermediate buffer utilization. A drawback of one-sided operations is the requirement of explicit synchronization operations to ensure the target is aware of its new memory state. By using byte-sized *MPI_Send* to provide synchronization, we ensure that these calls are non-blocking and are promptly sent to the destination with low latency. The non-blocking and the low latency behavior are promoted by ensuring that the target is always ready to receive acknowledge messages. In case a message is sent without a pairing receiving call, we rely on the underlying MPI implementation to buffer the acknowledgement message at the target's unexpected receive queue for later retrieval.

3. *Pond: UPC++*
   Pond is the parent library that we derive our implementation from. Pond's implementation is very similar to ours. It promotes actors performing local computation and abstracting communication using their ports. The main difference is on the underlying remote communication scheme. Pond uses UPC++ remote procedure calls – *RPC* – to exchange boundary layers and to perform synchronization. UPC++ combines both of our MPI implementations advantages: it supports communication and synchronization within single calls, and it transmits data using remote memory access techniques. GASNet-Ex – UPC++'s network layer – and UPC++ run-time environment ensure that RPCs are executed using RMA techniques, and that proper synchronization is accompanied along with it.

4. *Bulk-Synchronous Parallel: MPI*
   Both our implementations and Pond are derived from this MPI BSP model introduced on [BB12]. Patches are distributed among MPI ranks and synchronization happens at global scale. During run-time, 1. ranks exchange their patches' boundary values, then 2. they perform local computations and then 3. ranks are synchronized globally. *MPI_Sendrecv* is used to exchange data and *MPI_Barrier* to promote synchronization.

The tests were executed on *Linux-Cluster CoolMUC-2*, containing 384 28-way *Intel Xeon E5-2697 v3 ("Haswell")* processors – see appendix A.1 for more information. We have used 1, 2, 4, 8, 16 and 32 nodes with aggregated 28, 56, 112, 224, 448, and 896 cores, respectively. Each node instantiated 4 MPI ranks, and for each rank, 7 cores were reserved. As a reminder, we associate one patch per actor. In the strong scale scenario, the domain size remained constant with 16384x16384 grid cells. As the number of resources increased, the patch size decreased, and consequently, the number of actors increased. Example: in the 1 node scenario, the 16384x16384 domain was divided into 2048x2048 patches; and then 8x8 actors distributed among the 4 MPI ranks. And in the 32 nodes scenario, the domain was divided into 512x512 patches; which distributed the 32x32 actors among the 128 ranks. In the weak scale scenario, we increased the domain size along with the resources, in order to keep the workload per actor constant. Patches had their size fixed to 512x512. Each node held 32 actors distributed between its 4 MPI ranks. For example, in the 1 node scenario, a 2048x4096 domain was divided into 512x512 patches; and in the 32 nodes scenario, the 16384x16384 domain divided into the constant-sized 512x512 patches.

Figures 7.6 and 7.7 shows the results for weak and strong scaling tests, respectively. A first impression shows that all implementations scale somewhat similarly. At the lower bound, we have the BSP implementation. We entail lower performance due to the global synchronization model. Following BSP, we have both of our MPI implementations. MPI two-sided performs slightly better on the weak scaling tests. The explicit extra synchronization step within the one-sided implementation adds a bit of overhead. Furthermore, MPI two-sided *MPI_Issend* not only leverage communication and synchronization within single calls, but can also perform remote memory access on some MPI implementations, which leverages one-sided communication benefits plus synchronization. Leading in scalability, there is Pond. Its GASNet-Ex remote memory access interface and UPC++ run-time environment ensures that communication is not only optimal, but that it also includes the necessary synchronization steps without extra communication calls.

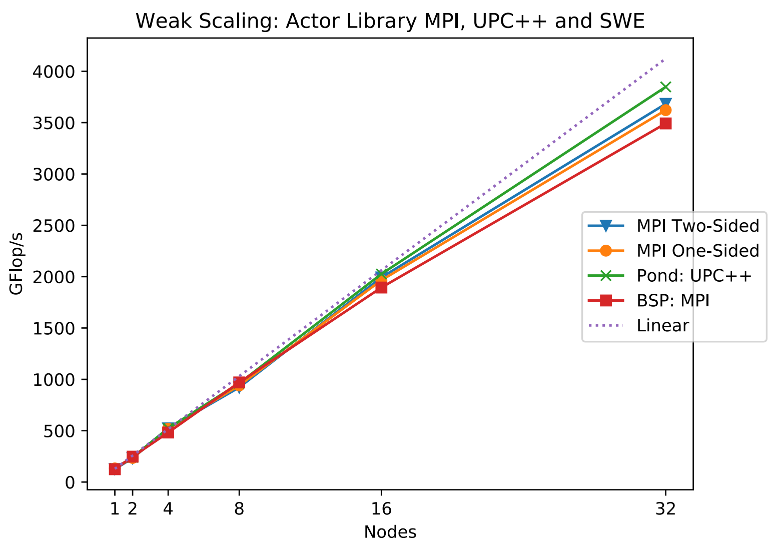


Figure 7.6.: Weak scaling tests for different SWE implementations. The domain size is increased along with the resources.

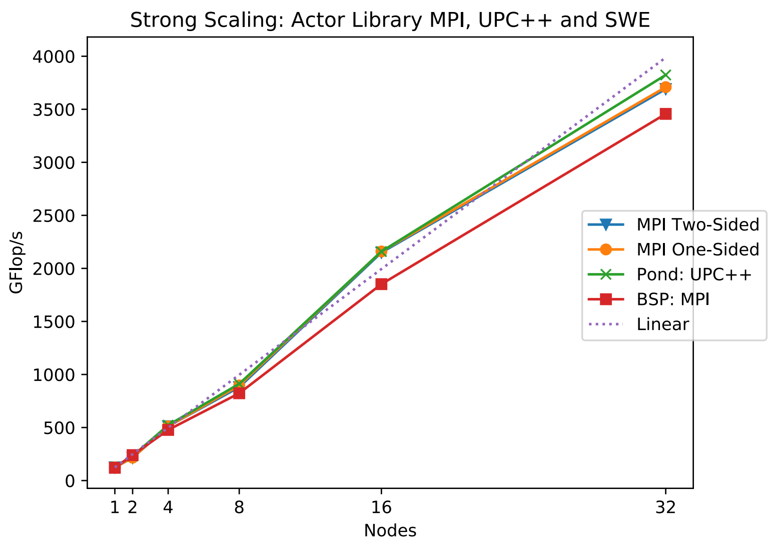


Figure 7.7.: Strong scaling tests for different SWE implementations. The domain size is fixed while the resources is increased.

8. Summary

We have introduced the actor programming model and a respective implementation for distributed and shared-memory systems.

The actor model consists of entities – *actors* – which encapsulates finite-state machines, internal states and explicit communication channels. Actors execution are triggered by explicit exchange of messages, which ensures that their internal states are not directly accessible. A communication channel is created when an actor's output port is connected to another actor's input port. An actor is idle until at least one of its ports is updated: either an output port completes a send operation, or an input port receives a new message. The application runs until all actors are in a final state. Actors are distributed among different processors, and can communicate through shared-memory or distributed-memory message-passing operations.

At compile-time, the user define the computation requirements and distribution among actors, and then specify communication dependencies among them. At run-time, the application follows a single program multiple data – *SPMD* – parallelism model: multiple copies – *ranks* – of the application is instantiated into different and common nodes. The application abstractly creates a directed multi-graph of actors, where vertices represent actors, and edges, communication channels.

With the help of METIS [KK98], a graph partitioning library, the actor graph is optimally partitioned into subsets of actors to ensure minimal edge-cuts; and thus, remote communication bandwidth. Subsets are distributed among different ranks. Communication between actors hosted by distinct nodes is done through message-passing protocols using MPI implementations. We have implemented two variations for our library's remote communication layer: one using two-sided communication operations, and the other using one-sided remote memory access – *RMA* – operations. For actors sharing the same node, communication is done through shared memory, where lock-free data structures are used to ensure low-latency memory access utilization.

An extensive research has been done on MPI in order to extract the best performance for remote communication operations, which usually bound the overall application performance. Due to the high-concurrency nature of our implementation, where parallelization occurs not only at distributed-memory level, but also within shared-memory, an extra care has been taken on hybrid programming models and multithreaded MPI. Following our findings, we had concluded that to extract the lowest communication latency and the highest bandwidth, MPI calls had to be funneled to a single thread. A decoupled architecture were designed, where computation within and between actors are parallelized, while remote communication requests and asynchronous progress are handled by a single thread.

Our library is derived from a previously actor model implementation from Pöppl [PBB19], which uses UPC++ to abstract distributed-memory into a partitioned global address space – *PGAS*.

Pöppl's version, similarly to our MPI one-sided implementation, uses RMA operations. Differently from MPI, UPC++ is backed by a network API – GASNet-EX [BH18] – and a run-time system to provide PGAS abstraction. Combined, these features promote data transfer and synchronization within single remote procedure calls; whereas in our implementation, RMA is accompanied with explicit two-sided synchronization messages. In the other hand, the usage of MPI leverages lower abstractions than UPC++, which allows fine-grainier control and optimization opportunities within communication operations. Similarly, both MPI and UPC++ supports asynchronous communication at user-level thread, which require active library calls to ensure asynchronous progression.

8.1. Conclusion

To analyze the performance and scalability of our implementation, we have tested it by proxying a discrete, scalable and structured mathematical model. The model, derived from the *shallow water equations* – SWE – in [BB12; Rol+16; PBB19], describes the propagation form of a wave within a specific domain, given an initial volume disturbance.

The domain is represented by a two-dimensional Cartesian grid where each cell depicts the state of a specific water volume: the sea floor height, the water height and the water velocities across the cell's boundaries. Patches – groups of adjacent cells – represent subsections of the domain. The SWE computation model resembles a classical BSP – bulk-synchronous parallelism – model, where: 1. Adjacent patches exchange their boundary values. 2. Patches calculate their inner cell values. 3. Synchronization occurs to ensure the validity of values (time-stepping model where the global wave speed could invalidate local computations).

Following the Pond's implementation [PBB19], we have applied our actor library to proxy SWE, where each patch is encapsulated within an actor, and adjacent actors are connected to each other. Actors are then distributed across different and within nodes, where remote communication is done through MPI while local uses shared-memory constructs.

We have executed four implementations of SWE and tracked their floating points operations per second: 1. Our MPI Two-Sided version. 2. Our MPI One-Sided version. 3. Our predecessor, Pond, which uses UPC++ as its communication backend. 4. Pond's predecessor, BSP [BB12], which uses MPI two-sided operations saving any programming model abstractions.

The tests aimed to compare the weak and strong scalability of the implementations. In the former, the available resources (nodes and cores) were scaled up according to the problem size. In the latter, the problem size remained constant while the resources were scaled up.

The results, shown in figures 7.6 and 7.7, indicate that all implementations performed similarly. BSP is encountered at the lower bound, probably due to its coarse-grained parallel model and usage of global synchronization constructs. Our MPI implementations came second, with the two-sided variation performing slightly better. Two-sided operations uses RMA techniques and implicitly creates synchronization points, which allows better optimizations. Pond, came at first with the best scalability. Besides the inspiring implementation done by Pöppl, we credit the UPC++ run-time environment and its underlying GASNet-Ex network layer which provides remote memory access operations and synchronization with single transmissions.

We believe that our library is not only applicable for structured mathematical models, but specially for unstructured ones, where communication occurs irregularly. The actor programming model provides great grounds for better system utilization and load balancing. Due to the utilization of well-known standards, such as MPI and OpenMP, our implementation should be agnostic to the underlying infrastructure it runs on. During architectural decisions, we were aware that some MPI operations may behave differently according to their underlying implementation and hosting system. And given these discrepancies, we aimed for implementation compatibility by circumventing worst case scenarios.

8.2. Future Work

In our current implementation, actors are distributed among the available resources during application initialization. Regardless of their state during application execution – functional, idle, or finished –, their allocated resources will only be released once the application ends. Depending on the application flow, this can result in underutilized system resources.

Let's assume an application whose first iterations provide a structure computation and communication scheme. After some time-spawn, sections of the application finishes their requirements, while others receive extra workloads. An optimal scenario would entail to deallocate finished actors, and release their hosting resources. Then, re-distribute hot sections of the application within the recently released resources. This accomplishment would require dynamic provisioning and distribution management of actors at run-time.

Another constraint in the current implementation is the premise that resources are homogeneous. Actors can have different data types, and depending on the provider system architecture, reside on CPUs with different characteristics. However, this is not true for different hardware types, such as GPUs. An optimal scenario would allow developers to tag actors as GPU dependents. Then, the library would ensure that logical units would be dispatched to GPUs, and that it would provide the necessary communication transparency between actors residing in different hardwares.

Finally, the current one-sided implementation falls behind in performance against the other implementations discussed. Further work could be done to provide communication and synchronization within single RMA calls, such as how UPC++ run-time system assures this functionality.

A. Implementation

A.1. Cluster

The cluster we have developed, tested and executed performance tests is from Leibniz Supercomputing Center.

In specific, the *Linux-Cluster CoolMUC-2* [LRZb], with 384 *Intel Xeon E5-2697 v3 ("Haswell")* processors, each containing 28 cores and 64*GB* shared RAM, which aggregates to 10752 cores. Interconnection is supported by *FDR14 Infiniband*.

A.2. Installation and Setup

Below we demonstrate how to compile and run our library under the architecture described on appendix A.1.

To compile, navigate to the repository's root folder, and then execute the commands described in figure A.1. They load the required modules and compile using CMake tool.

An example batch job script is shown in figure A.2 ([LRZa] for more information). The job runs a 16384*x*16384 domain subdivided into 2048*x*2048 patches. 8*x*8 (64) actors run those patches. 2 nodes that sums 56 cores will be allocated. 4 tasks per node instantiated, each with 7 cores reserved. Actors are distributed among theses tasks.

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

# Load Required Modules
module unload mpi.intel intel mkl
module load devEnv/Intel/2019 \
            netcdf/4.6.1-intel-impi-hdf5v1.10-parallel \
            gcc/8 metis/5.1.0-i32-r32 cmake/3.10

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

Figure A.1.: Compiling the Actor Library for the architecture described on appendix A.1.

```
#!/bin/bash
#SBATCH -o <some_output_dir>/16384.2.7.28.%j.out
#SBATCH -J mpp2.16384.2.7.28
#SBATCH --clusters=mpp2
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=7
#SBATCH --mail-type=end
#SBATCH --mail-user=<>
#SBATCH --time=00:30:00

# Load Required Modules
module unload mpi.intel intel mkl
module load devEnv/Intel/2019 \
            netcdf/4.6.1-intel-impi-hdf5v1.10-parallel \
            gcc/8 metis/5.1.0-i32-r32 cmake/3.10

# Run-time Variables
export MLX5_SINGLE_THREADED=0
export OMP_NUM_THREADS=7

# Start
mpiexec -n 8 --perhost 4 <compile_dir>/<compile_exe> \
        -x 16384 -y 16384 -p 2048 -e 1 -c 1 --scenario 2 \
        -o output/out
```

Figure A.2.: Example batch job script for the architecture described on appendix A.1.

List of Figures

Bibliography

[Agh85] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Tech. rep. AITR-844. MIT Artificial Intelligence Laboratory, June 1985.

[Arm96] J. M. Armstrong. "a Survey of the Language and its Industrial Applications." In: 1996.

[BB12] A. Breuer and M. Bader. "Teaching Parallel Programming Models on a Shallow-Water Code." In: *Proceedings of the 2012 11th International Symposium on Parallel and Distributed Computing*. ISPDC '12. IEEE Computer Society, 2012, pp. 301–308. ISBN: 978-0-7695-4805-0. DOI: `10.1109/ISPDC.2012.48`.

[BH18] D. Bonachea and P. H. Hargrove. *GASNet-EX: A High-Performance, Portable Communication Library for Exascale*. Tech. rep. LBNL-2001174. To appear: Languages and Compilers for Parallel Computing (LCPC'18). Lawrence Berkeley National Laboratory, Oct. 2018. DOI: `10.25344/S4QP4W`.

[Car18] M. S. Carlos R. *Tuning the Intel® MPI Library: Advanced Techniques*. Jan. 2018.

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

[CHS16] D. Charousset, R. Hiesgen, and T. C. Schmidt. "Revisiting Actor Programming in C++." In: *Computer Languages, Systems & Structures* 45 (Apr. 2016), pp. 105–131.

[For15] M. P. I. Forum. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015.

[HBS73] C. Hewitt, P. Bishop, and R. Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence." In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[HLL97] A. Harten, P. D. Lax, and B. van Leer. "On Upstream Differencing and Godunov-Type Schemes for Hyperbolic Conservation Laws." In: *Upwind and High-Resolution Schemes*. Ed. by M. Y. Hussaini, B. van Leer, and J. Van Rosendale. Springer, 1997, pp. 53–79. ISBN: 978-3-642-60543-7. DOI: `10.1007/978-3-642-60543-7_4`.

[IBM] IBM. *Buffer management for eager protocol*.

[Int19] Intel. *Asynchronous Progress Control*. Mar. 2019.

[KK98] 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. DOI: `10.1137/S1064827595287997`.

[KSK13] S. Kumar, Y. Sun, and L. V. Kale. "Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q." In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 689–699. ISBN: 978-0-7695-4971-2. DOI: `10.1109/IPDPS.2013.83`.

[Kum+08] S. Kumar, G. Dózsa, G. Almási, P. Heidelberger, D. Chen, M. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. E. Smith, and C. Archer. "The deep computing messaging framework: generalized scalable message passing on the blue gene/P supercomputer." In: *ICS*. 2008.

[LGB11] R. J. LeVeque, D. L. George, and M. J. Berger. "Tsunami modelling with adaptively refined finite volume methods." In: *Acta Numerica* 20 (2011), pp. 211–289. ISSN: 1474-0508. DOI: `10.1017/S0962492911000043`.

[LRZa] LRZ. *Example parallel job scripts on the Linux-Cluster*.

[LRZb] LRZ. *Overview of the Cluster Configuration*.

[Nor+11] P. Nordwall, J. Andrén, J. Rudolph, A. Engelen, C. Batay, and H. Edelson. *The Akka Actor Library Documentation*. 2011.

[PBB19] A. Pöppl, M. Bader, and S. Baden. "A UPC++ Actor Library and Its Evaluation on a Shallow Water Proxy Application." en. In: *Parallel Applications Workshop, Alternatives To MPI+X*. IEEE. Denver, Colorado, United States of America: IEEE/ACM/SigHPC, Sept. 2019.

[Pri+12] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella. "Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems." In: 2012.

[Rol+16] 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: ACM, 2016, pp. 24–29. ISBN: 978-1-4503-4386-2. DOI: `10.1145/2931028.2931033`.

[Tar+14] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. "X10 and APGAS at Petascale." In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '14. Orlando, Florida, USA: ACM, 2014, pp. 53–66. ISBN: 978-1-4503-2656-8. DOI: `10.1145/2555243.2555245`.

[TG09] R. Thakur and W. Gropp. "Test suite for evaluating performance of multithreaded MPI communication." In: *Parallel Computing* 35.12 (2009), pp. 608–617.