Technische Universität München
Fakultät für Informatik
Lehrstuhl für Wissenschaftliches Rechnen

Evaluation of the Actor Model for the Parallelization of Block-Structured Adaptive HPC Applications

Alexander Ludwig Pöppl

Vollständiger Abdruck der von der Fakultät für Informatik der Technische Universität München zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigten Dissertation.

Vorsitzender: Prof. Dr. Helmut Seidl

Prüfer der Dissertation:

1. Prof. Dr. Michael Georg Bader
2. Prof. Dr.-Ing. Michael Glaß

Die Dissertation wurde am 28.09.2020 bei der Technische Universität München eingereicht und durch die Fakultät für Informatik am 18.01.2021 angenommen.

Technical University of Munich
Department of Informatics
Chair of Scientific Computing in Computer Science

Evaluation of the Actor Model for the Parallelization of Block-Structured Adaptive HPC Applications

Alexander Ludwig Pöppl

Full imprint of the dissertation approved by the Department of Informatics
of Technical University of Munich to obtain the academic degree of
Doctor of Natural Sciences (Dr. rer. nat.)

Chairman: Prof. Dr. Helmut Seidl

Examiners of the Dissertation:

1. Prof. Dr. Michael Georg Bader
2. Prof. Dr.-Ing. Michael Glaß

The dissertation was submitted to Technical University of Munich on 28.09.2020 and was accepted by the Department of Informatics on 18.01.2021.

Acknowledgements

First and foremost, I would like to thank Prof. Dr. Michael Bader for his support and advice during these last years. I'd also like to thank my collaborators, both from within my chair and the invasive computing project. Furthermore, I would like to thank Prof. Scott Baden, Ph.D. for enabling me to spend time at Berkeley Lab, and the UPC++ team for their advice and counsel during my stay there. The time there enabled me to explore a new direction for my research. Moreover, I'd like to thank my students Ludwig Gärtner, Andreas Molzer, Jurek Olden, Martin Bogusz, Bruno Macedo Miguel and Yakup Budanaz for their work in conjunction with my project. I'd also like to thank the various funding agencies that funded my research and enabled me to use their resources.

I'm very grateful to my family for enabling me to study without financial issues. Finally, I'd like to thank my wife, May, for her support and patience, and for always being there to help when I needed her.

Abstract

Future HPC systems are expected to feature an increasingly complex and heterogeneous execution environment with parallelism on multiple levels. At that point, HPC developers need to worry about distributed and shared memory parallelism, and potentially deal with accelerators on top of that. Future hardware architectures feature tiled chip architectures or heterogeneous CPU cores. In contrast, the prevalent model of computation in HPC today is Bulk Synchronous Parallelism (BSP), embodied typically through a mixture of MPI for distributed memory parallelism, OpenMP for shared memory parallelism, and optionally a vendor-specific framework for the accelerator. While this provides for a straightforward computational model, it may prove too inflexible for future adaptive applications.

The actor model has been a popular technique in the domain of embedded computing to enable predictable execution in parallel systems. In my thesis, I explore the advantages of using the FunState actor model, with its clear separation of computation and control flow, within tsunami simulation as an example application domain. The actor model adds a level of abstraction over the conventionally used approach and therefore allows for the implementation of more complex coordination schemes. In an actor-based application, the application developers specify the behavior of the actor, the inputs it consumes, and the outputs it provides. Actors are then connected to form a graph that propels the computation through the production and consumption of tokens. The target architectures for the application are classical HPC systems, as well as the novel Invasive Computing compute stack with its heterogeneous tiled MPSoC architecture.

I implemented actor libraries for the use in the aforementioned hardware landscapes. The benefits of using the model is demonstrated using actor-based tsunami proxy applications. For use in the invasive stack, the application is extended to exploit a reconfigurable processor developed as part of the project. The applications use actors for patch coordination, allowing for an implicit overlap of communication and computation. Furthermore, the actor model was used to implement a "lazy activation scheme", where computations of sections of the ocean not yet reached by the tsunami wave may be avoided. In a comparison with a BSP-based tsunami application on a cluster employed with Intel Many-Core processors, the actor-based approach exhibited a higher performance and better scalability as well as lower complexity of implementation.

Contents

Part I.

Introduction and Theory

1. Motivation

The field of High Performance Computing (HPC) stands at the intersection of computer science, mathematics, engineering and the natural sciences. Natural scientists and engineers use simulation alongside physical experimentation to gain insights about natural phenomena or the properties of a system under test, e.g. an engine or an airplane. Simulation of the behavior of many of these problems require a large amount of calculations to be performed. Performing these calculations in a time frame that makes the simulations useful often requires orders of magnitude more computing power than a single computing device may provide. Problems that will require the full available compute performance of upcoming exascale supercomputers include simulations using multiple physical models to solve a problem (Vázquez et al., 2016), simulation of computational fluid dynamics (e.g. to simulate entire airplanes (Borrell et al., 2020)) or plasma physics (used for example to simulate nuclear fusion reactors (Heene et al., 2018)). At the same time, mathematicians provide novel numerical methods to solve problems posed by the application domain, such as the Arbitrary High-Order Discontinuous Galerkin (ADER-DG) method (Dumbser et al., 2008). Use of these new methods enables more accurate results, potentially along with a reduction of the number of computations that need to be performed, but may come with an increase in program complexity. For computer scientists, the goal is to provide an efficient implementation that makes full use of the capabilities of the hardware the problem is executed on. At the same time, software ought to be extensible so that future requirements can be met. It should also be resilient in the face of hardware failures and easily understood by the domain scientist.

Twenty years ago, this was a comparatively simple task. The then-new cluster architectures featured a number of single-core processing units that used scalar instruction sets and an interconnection network (Christon et al., 1997). In the time that followed, the hardware landscape became increasingly more complex. What used to be single-core processing units became compute nodes featuring several multi-core (or even many-core) processors, potentially with hundreds of execution units available to them. Instead of scalar instructions, modern instruction sets use Single Instruction Multiple Data (SIMD) instructions that compute the same operation on more than one data element at the same time. Furthermore, many current supercomputers utilize accelerator cards that provide additional performance for floating-point operations. At the same time, the fundamental building block used to program these machines remains the same: *message passing* using the Message Passing Interface (MPI) standard. This alone is no longer sufficient, however. Therefore, application developers typically use further frameworks on top of MPI. For shared-memory parallelization, OpenMP has emerged as the prevalent standard. Accelerators are typically programmed using vendor-specific languages such as CUDA[1].

[1] At the time of writing, no open standard has gained a significant share of the market, although SyCL (Khronos Group, 2020) and Kokkos (Edwards, Trott, and Sunderland, 2014) are interesting candidates.

Alongside the technology stack, the bulk synchronous parallel (BSP) approach, a canonical style to write HPC applications has emerged. When it is used, the processing units participating in a computation follow a (potentially repeating) sequence of so-called *super steps* consisting of computation, communication and synchronization. However, more complex applications as well as a more heterogeneous hardware landscape make this approach increasingly unsuitable. As a reaction to this, a wide range of new methods to write HPC applications has emerged. In the Transregional Collaborative Research Center Invasive Computing (InvasIC) project (c.f. chapter 8), we investigate how to program these new systems from both the views of HPC and of embedded systems.

In the latter domain, the actor model is a known technique for modeling parallel applications while retaining predictable execution. It enforces the separation of computation, coordination and computation, and makes each separately analyzable. In the FunState (Strehl et al., 2001; Roloff et al., 2016) actor model, active objects called *actors* send each other *tokens* over defined communication *channels*. The actors within a system execute concurrently, based on the data that is available to them on their communication endpoints (*ports*). The behavior of an actor is governed by its *finite state machine*. Whenever the data in an actor's ports changes, its finite state machine may perform a state transition and perform the functionality associated with that transition, assuming that the data within the ports matches the *activation pattern*. Actors and the channels that connect them form the *actor graph* of an application. This graph may then be distributed onto the resources available to the computation. Actor-based applications are contributing towards the goal of predictable application execution within the invasive technology stack. By making the structure of the computation explicitly available to the outside, a deeper analysis of the performance characteristics is enabled. The tiled hardware architecture of the invasive computing technology stack has strong similarities with the clustered hardware architectures of modern supercomputers. My goal was therefore to implement and to evaluate the actor model on both the invasive hardware as well as HPC architectures and to evaluate:

1. *Does the actor model ease the development of HPC applications compared to traditional models?*
2. *Does using the actor model in HPC applications yield performance competitive with the traditional approach?*

To evaluate these questions, I collaborated on the implementation of ActorX10, an X10 actor library, and used it to implement SWE-X10, a tsunami proxy application. Furthermore, I developed Actor-UPC++ for larger scale HPC applications. It is based on the lessons learned of ActorX10, and able to accommodate a larger number of actors compared to its predecessor. It is based on modern C++, which makes it more comparable to the traditional HPC software environment. The library is evaluated using Pond, another tsunami proxy application.

1.1. Thesis Structure

This thesis is centered around the two actor libraries *ActorX10* and *Actor-UPC++* and their use in tsunami proxy applications. The actor model has been used successfully in other domains of computer science, such as embedded computing.

The remainder of this thesis is organized along the aforementioned research questions. In part I, I introduce the context this work is grounded in, starting with an overview of parallel programming terminology in chapter 2. This is followed by a review of the available HPC parallelization frameworks. First and foremost, there is the prevalent HPC technology stack (in chapter 3), consisting of MPI for distributed-memory parallelization and of OpenMP for shared-memory parallelization. These libraries are used in most production applications today. As such, it is, on the one hand, a competing approach to the one proposed in this thesis. On the other hand, MPI and OpenMP are sufficiently low-level to be used in higher-level libraries such as mine. Both OpenMP and MPI are used: OpenMP is used for parallelization on the node-level in Actor-UPC++, and MPI is used for communication within Actor-MPI. Next, UPC++ is discussed in chapter 4. The library's focus is on communication in a PGAS environment. It exploits one-sided remote direct memory access operations offered by modern interconnection networks and aims to enable the programmer to maximize overlap between communication and computation through systematic use of asynchronous communication operations. Again, UPC++ may be viewed as both a competing approach and as a platform for an actor library (Actor-UPC++). Like MPI, UPC++ does not prescribe a model of control, and therefore allows the application developer to implement her own. I used it as the communication backend for Actor-UPC++. Using this, the overhead of the actor library was sufficiently small to run an actor-based tsunami application efficiently on a Xeon Phi cluster. The X10 programming language (discussed in chapter 5) forms the basis for the other actor library introduced in this thesis, ActorX10. X10 provides a global view of the computational domain, and allows for a direct transfer of entire object graphs between the different processes within the application. For the actor library, this enables the transparent use of arbitrary token types transferred between actors without custom serialization provided by the application developer as well as the migration of actors and their associated object graphs between processes. Chapter 6 discusses task-based parallelism, a competing approach to the actor-based parallelism discussed in this thesis. Both HPX and Regent use asynchronous task execution as a parallelization scheme. There, code is distributed into small work packages (tasks). Each task's dependencies are specified, and then the resulting task graph is brought to execution on a distributed system. As with my approach, this prescribes a specific application structure: in Regent, the tasks are created implicitly from sequential source core, while in HPX, tasks are specified either manually or using a set of predefined parallel operations. Chapter 7 focuses on frameworks and languages based on the actor model. These follow, for the most part, a different variant of the actor model that is based around a central mailbox per actor rather than ports and channels. Nevertheless, the implementations have aspects that may also be useful to add to the libraries discussed in this thesis. Specifically, this pertains to the resilience feature of Erlang, and the dynamic load balancing that is part of Charm++. Finally, I introduce the Invasive Computing project landscape in chapter 8. The project proposes a novel technology stack that uses a dynamic and exclusive resource allocation to obtain a predictable execution environment that is nevertheless able to flexibly adapt to changing application and system demands. Within the project, the actor model is used to formalize the structure of applications for use in the invasive design flow. When the actor graph of an application is known, one may then explore different mappings of the graph to the available resources, and therefore obtain an efficient execution environment. The resulting mappings may then be embedded into the application to be selected at runtime.

Part II introduces the actor libraries implemented in this thesis. The libraries are based on the FunState Actor model (chapter 9), proposed originally for use in the embedded domain. The

model adds structured communication paths and an explicit control model to the traditional actor model. It serves as the theoretical foundation for the implementations discussed here. ActorX10, discussed in chapter 10, was created as a collaboration within the Invasive Computing project. It is implemented in X10, and uses the language's PGAS features to realize a distributed actor library. Actors may communicate between ranks, and be moved freely between them. The target environment for ActorX10 is both the invasive stack and HPC systems. X10 is able to generate code that uses MPI for communication between ranks, and may therefore be used on distributed systems. On the invasive stack, however, the library profits from a direct compilation to native code, and from communication operations that utilize hardware acceleration. After that, I discuss Actor-UPC++ (chapter 11). It is proposed as a more light-weight alternative to ActorX10 using C++ and the UPC++ communication library. Compared to ActorX10, the individual actors carry less overhead, and do not have a dedicated thread. This becomes especially important for larger degrees of available parallelism, such as the one available on many-core processors. Finally, Actor-MPI, developed in the context of a master's thesis, replaces UPC++ as the communication backend for MPI.

Part III discusses the application of the actor-based approach for the parallelization of tsunami simulations. The theoretical model of the application domain is discussed in chapter 14. Thereafter, I introduce the shallow water teaching code SWE (chapter 15). In a collaboration with bachelor students and another colleague from my chair, we added support for different parallelization frameworks within SWE. This serves as contrast to the two actor-based solutions introduced thereafter. SWE-X10 (chapter 16) serves to demonstrate the viability of the actor-based approach. It targets both the invasive technology stack as well as small HPC systems. After an evaluation of SWE-X10 both on CPUs and GPUs (based on the results of a bachelor's thesis), the benefits of lazy activation of actors is shown. SWE-X10 serves as one of the demonstration applications for the invasive technology stack, and is is able to utilize the custom silicon developed within the project. In chapter 17, I discuss the results of optimizing SWE-X10 into the invasive technology stack. The performance of Pond, a tsunami simulation based on Actor-UPC++, is evaluated on a many-core cluster in chapter 18. This serves as demonstration that it is possible to use the actor model in larger-scale computations, and that the resulting performance is at least competitive, or even better than the traditional approach.

2. Parallel Programming Concepts

Modern Computer architectures exhibit a wide range of parallelism across all levels. Within a CPU core, it is possible to work on multiple instructions concurrently, interleaving the instruction fetch, execution and retirement as well as computing multiple instructions at the same time. These instructions may now not only perform an operation on a single value, but, depending on the architecture, on up to 8 Double Precision (or 16 Single Precision) floating-point numbers[1]. Within a CPU, there are usually multiple independent cores. Each of them will typically have its own Level 1 and Level 2 cache, and a Level 3 cache that is shared across all cores. Possibly, there will also be multiple CPUs in a computer. On this level, all cores typically have shared access to the main memory of the system. In more sophisticated workstations or servers (nodes), this gets more complicated as different cores will have varying access bandwidth and latency for different regions of memory. Finally, for more complex computational tasks, a single machine may no longer be sufficient. To solve these, multiple nodes are connected to form a compute cluster. Each node in the cluster has its own working memory and compute resources, and they use a fast interconnect to exchange data necessary to complete the computation. In many of these cases, direct access to all parallel features leads to an exponential increase in complexity of the software, and is, therefore, not feasible for application developers. Additionally, software written in this way would not be portable to other computer architectures. Instead, frameworks and libraries are used to make the parallelism accessible to developers in a simplified and structured way.

In this chapter, I will introduce terms and models frequently encountered in parallel programming. There are multiple different classification approaches and models, with some focusing more on the computer architecture perspective, and some more on the data perspective. This loosely follows the work of Eijkhout, van de Geijn, and Chow (2011) and Hager and Wellein (2011).

2.1. Classification of Parallel Program Execution

An early classification of parallel program execution is Flynn's taxonomy (Flynn, 1972). Flynn classifies parallelism along two dimensions, instructions and data.

Single Instruction Single Data (SISD) comprises classic uni-processor systems following the von Neumann architecture. In each computing step, one instruction is applied to one element of data. In general, this model is still the underlying assumption for most imperative programming languages. For example, the C programming language was originally targeted towards the PDP-11 computer architecture, and its execution model still resembles the constraints imposed by

[1] Using *Intel Advanced Vector Extensions 512 (AVX-512)* instructions (Intel Corporation, 2011).

that architecture(Chisnall et al., 2015). In modern CPUs, there are typically multiple SISD units working in concert.

Single Instruction Multiple Data (SIMD) originally referred to vector architectures. In these, the same instruction is applied to multiple data elements at the same time. A lot of applications from the scientific computing domain greatly benefit from this, as the same calculation is typically applied to numerous data points. For this reason, vector computers such as the early Cray machines dominated the supercomputer market of the 1970s and the 1980s, until they were superseded by architectures using clusters of commodity processors (Espasa, Valero, and Smith, 1998). The Cray-1 supercomputer was able to compute 64 double precision floating-point operations at the same time (Cray Research Inc., 1976). Modern microprocessor architectures still use vector instructions to accelerate workloads with regular patterns. For modern architectures, the typical vector length is two (SSE) to eight (AVX-512) double precision floating-point values. SIMD parallelization therefore still is an important component in fully utilizing modern HPC systems.

Multiple Instruction Single Data (MISD) is more difficult to summarize. Few computer architectures actually implemented this model (Barney, 2010). The idea here is that multiple, potentially different instructions, are applied in parallel to the same data element. One conceivable use case would be a redundant computation using different methods, combined with an arbitration step in the end, to obtain a system that is, as a whole, more fault-tolerant. Such a system is useful in safety-critical use cases, and was used for example for the computer system of the Space Shuttle (Spector and Gifford, 1984). In the realm of HPC, however, it is not commonly encountered.

Multiple Instruction Multiple Data (MIMD) architectures comprise basically any modern hardware architectures, tightly coupled in the form of multi-core CPUs, and more loosely coupled in the form of interconnected compute clusters (Barney, 2010). The former may be found in all current form factors today, from mobile phone to server CPUs, and the latter is the prevalent organization form for modern supercomputers. As such, MIMD is of great importance in general, and in HPC specifically, where it is encountered on multiple layers of the parallelization hierarchy.

In the following years, this classification was extended to match different execution models of emerging architectures. In the context of general purpose GPU computing, the term *Single Instruction Multiple Thread (SIMT)* is frequently used. One may view it as a sub-category of SIMD. It refers to the GPU's execution model, where multiple threads of a kernel are executed concurrently by multiple small compute cores in lockstep. For the application programmer, these instructions behave independently, but internally, they share the same instruction control units (NVIDIA Corporation, 2017). Finally, *Single Program Multiple Data (SPMD)*, a subcategory of MIMD refers to executing the same program on multiple processing units (Darema, 2001). In contrast to SIMD or SIMT, program execution may diverge between different processes based on the data elements computed. SPMD maps ideally to the prevalent cluster architectures, and is, therefore, the most common execution model for HPC applications today.

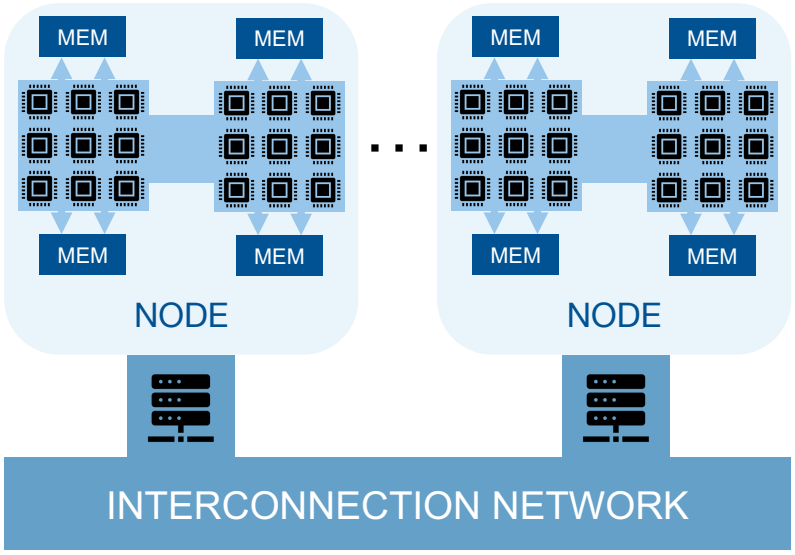


Figure 2.1.: Sample system architecture for a compute cluster. Each compute node has two sockets, and four NUMA domains. The nodes are connected using an interconnection network.

2.2. Classification Based on Memory Access Types

Another possibility to classify parallelism is motivated through the shape of modern HPC cluster architectures. Figure 2.1 schematically depicts such an architecture. Each node comprises one or more sockets, each with multiple processing units. Nodes are connected using an interconnection network. To obtain the best possible performance, application programmers need to exploit available parallelism within nodes as well as across node boundaries, and adhere to the structure imposed by the hardware.

Memory within a node is freely accessible by all its compute units. In many cases, it is enough to run a single process per compute node. Parallelism is then achieved by using multiple threads. These threads share the same address space, therefore this type of parallelism is referred to as *shared-memory parallelism*. This type usually exhibits a shared view on the data, i.e. all modifications of data are immediately visible by other processing units. This mandates an implementation of the caches that makes write accesses apparent across the cache hierarchies of the other processing units in the shared memory domain. Typically, this is referred to as *cache-coherence*. However, shared memory does not necessarily imply uniform access speed. With the increase of the number of cores per node in recent years, providing a *Uniform Memory Access (UMA)*, i.e. uniform bandwidth and latencies from all processing units, became more difficult. For larger systems, e.g. multi-socket systems or MIC architectures, memory is typically attached to subcomponents, e.g. to a socket in a multi-socket system (Sodani, 2015). That component usually has a fast and direct connection to that memory, while other components access the memory using some bridging component such as a Network-on-Chip. Providing uniform memory access speeds would be possible, but to the detriment of the access speed of the directly connected component. Instead, the speed difference is explicitly exposed to the programmer, as a *cache-coherent Non-Uniform Memory Access (ccNUMA)* architecture. To reach the best possible performance on these architectures, it is important that

memory accesses take place within the NUMA domain of the executing processing unit as often as possible.

Typically, the nodes in a cluster are connected using an interconnection network. Each node executes one or more application instances, each with its own address space, and communicates with instances on other nodes using the network. This is called *distributed memory parallelism*. Communication in this context may be implemented by involving both the sending and the receiving nodes' processing units (*two-sided communication*). Alternatively, if the network interface supports it, *remote memory accesses (RMA)* may be used. Then, only the processing unit initiating the communication, either as sender or as receiver, needs to be actively involved. This is called *one-sided communication*.

There also exist hybrids between shared and distributed memory: it has been proposed to view the entire memory of a large-scale cluster as a single global address space. This makes it possible to directly hold pointers to data on other nodes, and, depending on the implementation, to access remote data directly. Naturally, the latency and bandwidth for remote accesses vastly differs between local and remote accesses. To make the cost obvious to the application developer, the address space is typically partitioned into segments corresponding to the individual hardware units. The result is a Partitioned Global Address Space (PGAS) model. Two instances of the model are discussed in this thesis: UPC++ in chapter 4 and X10 in chapter 5.

2.2.1. Bulk Synchronous Parallelism

The BSP model is an important formalism for parallel computations in distributed memory. Valiant (1990) proposed it as a bridging model that formalizes a way to program distributed systems and to estimate the execution time. The model makes assumptions about both the hardware and the software, but is not restricted to modelling either, hence the term bridging model. On the hardware side, it assumes n homogeneous processors that are fully interconnected using a network with the facilities to perform a synchronization. Software is organized into a finite sequence of N *super steps*. Each supersteps, in turn, consists of three components: In the first step, in the *computation* step the processors involved in the computation perform work on their locally available data. In the second step, *communication* the processors exchange point-to-point messages with other processors in order to communicate all necessary information to perform the next computation step. Finally, in the third step, a *synchronization* step is performed to make sure that all processors are ready to continue with the next super step. Based on this, the model allows for the approximation of the overall runtime of a program as

$$\begin{aligned} T &= T_{\text{compute}} + T_{\text{communicate}} + T_{\text{synchronize}} \\ &= \sum_{N} \gamma n_{\text{Ops}} + \sum_{N} \beta n_{\text{Msg}} + \sum_{N} \lambda \end{aligned}$$

or $N\left(\gamma n_{\text{Ops}} + \beta n_{\text{Msg}} + \lambda\right)$ if all super steps are identical. The individual times for the super step components are summed up into combined computation time T_{compute}, communication time $T_{\text{communicate}}$ and synchronization time $T_{\text{synchronize}}$. For each computation component in each super step, one needs to determine the number of operations n_{Ops} that are performed by each processor

```
void perform_stencil(float *restrict u, float *restrict u_next, size_t xs, size_t
↪  ys) {
    for (size_t y = 1; y < ys - 1; y++) {
        for (size_t x = 1; x < xs - 1; x++) {
            u_next[y * xs + x] = u[y * xs + x]
                - 0.25f * (u[y * xs + (x - 1)] + u[y * xs + (x + 1)]
                           + u[(y - 1) * xs + x] + u[(y + 1) * xs + x]);
        }
    }
}
```

Figure 2.2.: An example of a computation that is data parallel. The computation of an element of the array `u_next` is only dependent on values from `u`, therefore the computation of the values of `u_next` may be performed concurrently

and the total size of all transferred messages n_{Msg}. The units used depend on the desired hardware parameters, but typically, one uses floating-point operations and bytes transferred. Furthermore, one needs to determine the parameters for the compute performance γ, the network bandwidth β, and the synchronization latency λ. In practice, it often occurs that all super steps are identical. It then suffices to calculate each component once, and then multiply by the number of performed super steps. The BSP model clearly separates the different computational phases, and therefore provides for a clear mental model of parallel applications. The clear separation of the different steps through the synchronization step further eases the conceptual burden on the application developer. As long as the model is followed, no deadlocks or lifelocks may occur, and therefore there is no need for more complex synchronization methods (Tiskin, 2011). The model has formed the basis for modern HPC in the last decades, and numerous HPC applications are based on its principles. An example for an application based on BSP is the non-actorized version of the SWE code discussed in chapter 15.

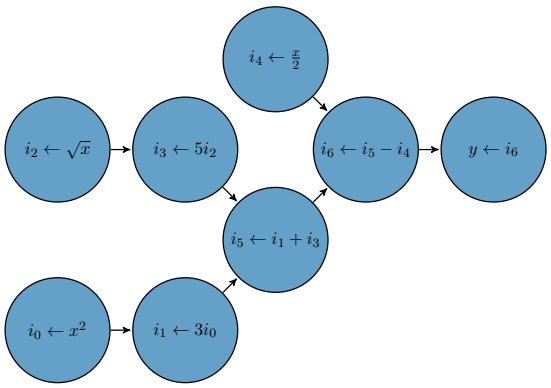
2.3. Classification of Parallelism Based on Type and Granularity of Work Performed

Another possibility for the characterization of parallelism is through the type and the granularity of the work items that are to be concurrently performed. To this end, parallelism is classified on a spectrum between *data parallelism* and *task parallelism* (Eijkhout, van de Geijn, and Chow, 2011).

In High Performance Computing, it is very common to operate on large arrays of data objects (e.g. grid cells, grid points, molecules, ...) and to perform an identical operation on each of them. One example would be a stencil code (see Figure 2.2). In the code, the values of an array, `u_next`, are set based on the values of another array, `u`. As there are no direct dependencies between different values of `u_next`, and `u` is only read, in theory, all values of `u_next` could be computed in parallel. This type of problem may be efficiently parallelized using computers following the SIMD paradigm, such as vector architectures, and also GPUs.

$$
\begin{aligned}
i_0 &\leftarrow x^2 \\
i_1 &\leftarrow 3i_0 \\
i_2 &\leftarrow \sqrt{x} \\
i_3 &\leftarrow 5i_2 \\
i_4 &\leftarrow \tfrac{x}{2} \\
i_5 &\leftarrow i_1 + i_3 \\
i_6 &\leftarrow i_5 - i_4 \\
y &\leftarrow i_6
\end{aligned}
$$

(a) Task Decomposition



(b) Task Dependency Graph

Figure 2.3.: An example of an ILP-level task graph. The vertices of the graph denote elementary operations, and incoming edges signify that a vertex depends on the vertex the edge originates from.

However, most programs contain more potential for concurrent execution. It is possible to subdivide a program into *tasks*, self-contained sections of a program that perform a specific functionality (Barney, 2010). In a further step, the tasks are analyzed to deduce the dependencies. If two tasks are not dependent on each other, they may be executed concurrently. For example, the term

$$y = 3x^2 + 5\sqrt{x} - \frac{2}{x} \tag{2.1}$$

may be decomposed into the following tasks as depicted in Figure 2.3a. Each task depends on the completion of the tasks that compute its intermediate variables. The dependencies are depicted in Figure 2.3b. Tasks that do not have a (transitive) connection may be computed concurrently. Here, this would be, e.g. $i_1 \leftarrow 3i_0$, $i_3 \leftarrow 5i_2$ and $i_4 \leftarrow \frac{2}{x}$, or $i_5 \leftarrow i_1 + i_3$ and $i_4 \leftarrow \frac{2}{x}$.

The tasks in the example above are very fine-granular. If the individual tasks are—like in the example—on the granularity of individual operations, they are also referred to as *instruction-level parallelism (ILP)*. But the approach shown in the example works not only for fine-grained parallelism, but also larger scale problems. In both cases, one first needs to identify tasks and their dependencies, and then decide on a parallel execution scheme. For instruction-level parallelism, the parallel execution is typically implemented by the CPU. Modern microprocessor architectures are able to execute multiple operations at the same time, and are able to reorder the instruction stream as needed to achieve maximum throughput (Hager and Wellein, 2011). More coarse-grained tasks may be used for parallelization in HPC applications. Scheduling and distribution of tasks is an NP-complete problem, even when significantly constrained (Ullmann, 1975). There are specialized runtime systems for HPC applications which can schedule tasks onto shared-memory as well as distributed-memory systems. In section 6.1, I will discuss the Regent programming language[2] for task-based HPC runtime systems.

[2] Website: `https://regent-lang.org`

3. MPI and OpenMP: The Prevalent Contemporary HPC Technology Stack

Up until the early 1990s, vector computers were the most popular computer architectures for scientific workloads with systems such as the Cray-1 dominating the computing centers at the time. At the same time, highly integrated microprocessors comparatively gained in performance and efficiency. At this point, the first HPC systems based on the technology began to appear. Broadly speaking, it is possible to distinguish between two different categories: *symmetric multiprocessor systems (SMP)*, which have multiple CPUs operating on a shared memory, and *massively parallel systems (MPP)*, where many CPUs with their own memory segments are connected to form a cluster. (Espasa, Valero, and Smith, 1998) In modern HPC systems, both categories can be found within a single system. On the coarser level, multiple nodes are connected to form a cluster. This cluster is then typically programmed using MPI. On the node level, there are typically multiple CPUs, each with multiple cores. These are typically programmed using OpenMP. The combination of these two programming frameworks forms the basis for a large majority of current HPC applications. I will discuss MPI in section 3.1 and OpenMP in section 3.2.

3.1. MPI: Message-Based Distributed Memory Parallelism

For the early distributed systems, there existed a number of different, proprietary, and competing standards, amongst others Intel NX (Pierce, 1988; MPI Forum, 2015). Most were specific to certain computer architectures or vendors. This complicated the creation of portable software. MPI emerged from an effort to provide a universally compatible standard. Starting out with a focus on synchronous point-to-point communication, it was extended in a subsequent version, 2.0, with features such as collective operations (first blocking, non-blocking with MPI-3.0), one-sided communication operations or I/O operations (MPI Forum, 2015). It is important to note that MPI is an *interface standard*, it describes the interface of a library and its semantics, but *does not prescribe a specific implementation*. These are typically provided by the vendors of HPC systems. There exist commercially distributed implementations (e.g. Intel MPI[1]), as well as widely used open source implementations (e.g. MPICH [2] or OpenMPI[3]).

In this section, I will discuss the SPMD execution model using the example of MPI, and describe the most important communication operations. As MPI is the currently most widely used parallel

[1] `https://software.intel.com/en-us/mpi-library`

[2] `https://www.mpich.org`

[3] `https://www.open-mpi.org`

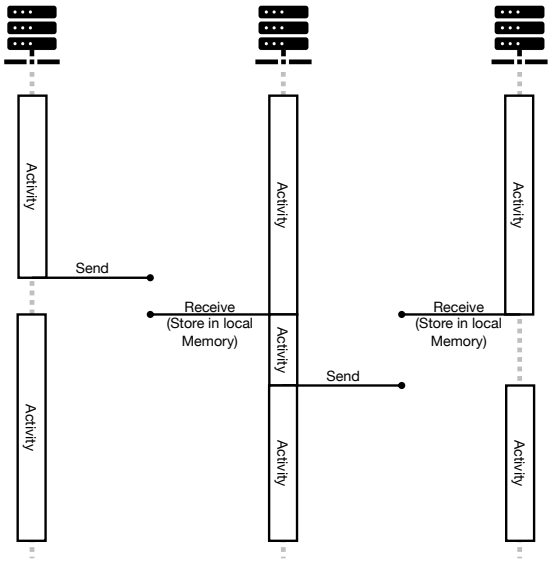


Figure 3.1.: Schematic of synchronous and blocking MPI Send and Receive operations. Each Send has to be matched by a receive on the target rank. The rank that initiates its part of the exchange first has to wait for the partner to initiate its counterpart to perform the actual data exchange.

model for HPC applications, this will serve to establish a baseline for comparison with other models, discussed in the sections below. The description of the communication operations and the interface are based on the standard published by the MPI Forum (2015).

3.1.1. Basic Operations

Most applications written using MPI follow the SPMD style of execution, where multiple instances of the same program are executed in different processes. In the MPI terminology, each process is referred to as a *rank*. Ranks may consist of multiple threads, and have their own private address space. For communication, a *communicator* object is used. Ranks registered with the same communicator are able to communicate. The most default one is `MPI_COMM_WORLD`. It is created upon initialization of the MPI library through a call to `MPI_init()`. At program termination, the Communicator is deleted using `MPI_Finalize()`.

To exchange data, point-to-point messages are sent. The most basic functions to achieve this are `MPI_Send` and `MPI_Recv`. They allow for the exchange of structured data (Integers, Floating Point Numbers, structured MPI data types) between exactly two ranks. A send operation is always matched to exactly one receive operation (and the other way around). A receive operation will only terminate once the corresponding send operation completes. The send operation has different communication types available. In the synchronous case, the operation will only complete once the data has been received. This behavior is illustrated in Figure 3.1. Here, the rank on the left starts a send operation. It has to wait until the corresponding repeat is called by the middle rank. As the sending rank is already waiting, the request will be executed immediately, and the middle

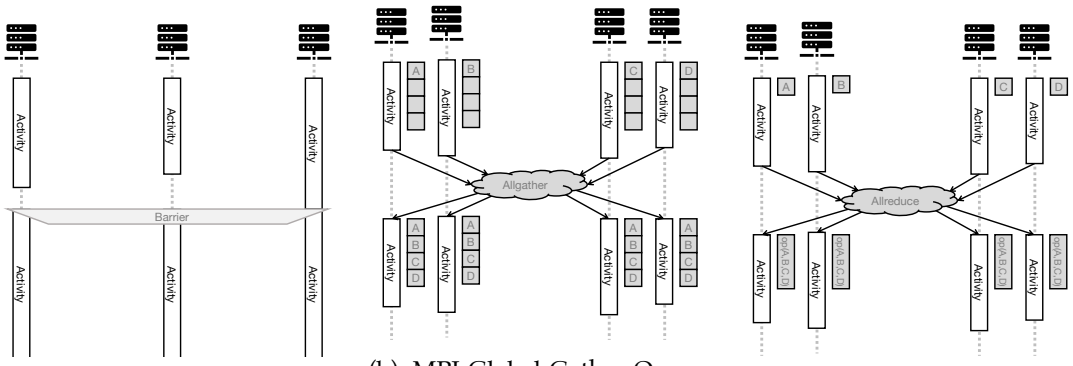
rank may resume computation. Similarly, when it sends data to the rank on the right, that rank is already blocked in the corresponding receive operation.

The communication operations depicted in the example use the default blocking communication mode. This causes the thread invoking the communication operation to block until the rank on the other side of the communication operation has invoked its operation. Alternatively, it is also possible to use non-blocking communication. Here, the rank receives a handle for the communication request that may be queried to determine the operation's state, i.e., to see whether the communication partner has invoked the operation on its side. Non-blocking communication operations have an "I" prefix to their operation name, e.g., `MPI_Send` would become `MPI_Isend`. For send operations, there are three different modes available. *Synchronous send* operations only return once the corresponding receive operation has invoked by the receiving rank. *Buffered send* operations return once the data being sent is no longer needed for the send operation. This may either be when the data has sent successfully, or when it has been copied to an MPI-internal buffer. Finally, the *Ready send* operation assumes that a corresponding receive operation has already been posted. The operation returns immediately. If that assumption does not hold, the behavior is undefined. The application developer may select the type of the send operation to be used using one of the prefixes for the operation ("B" for buffered, "R" for ready and "S" for synchronous), and combine them with the non-blocking version. When no prefix is used, the MPI implementation may decide the mode to use based on the input.

Figure 3.2 depicts a simple MPI program. Each process executes the `main()` function. The first operation is the `MPI_init(argc,argv)` call, which initializes the MPI library. The calls `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` and `MPI_Comm_size(MPI_COMM_WORLD, &numRanks)` serve to determine the number of the own rank and the overall number of ranks, respectively. The first parameter in this case is a constant that holds the handle for the communicator encompassing all ranks that is created during initialization, and the second one serves as the location to store the respective result. At the beginning of the communication, Rank 0 sends a message to the next rank. Afterwards it will wait for incoming messages. Ranks 1 to `totalRanks - 1` will initially receive messages. Once a rank receives a message, it will decrement the received number by one and send it on to the next rank, wrapping around at `numRanks`. A rank will terminate once it receives a number that is ≤ 0. The example uses the default point-to-point operation primitives, `MPI_Send` and `MPI_Recv`, to send and receive values. When using these blocking operations, care needs to be taken not to create deadlocks. In this simple example, it is enough to have rank 0 kick off the sending with a single `MPI_Send`. In cases, where there is an exchange between all ranks, e.g. a ghost layer exchange, the deadlock needs to be resolved explicitly. For example, a deadlock would happen, if all ranks tried to send initially, followed by a receive. Then, all processes would wait for the corresponding receive operation, and no progress could be made. To mitigate this, one could, for example, have all evenly numbered ranks receive first, then and all oddly numbered ranks receive first, and then the other way around. However, this is a common problem, and, therefore, the standard also includes a combined `MPI_Sendrecv` operation that takes care of the deadlocking issue transparently.

```c
#include <stdio.h>
#include <stdlib.h>

#include <mpi.h>

int main(int argc, char **argv) {
    int rank;
    int totalRanks;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &totalRanks);

    int data = 100000;

    if (rank == 0) {
        data = 10;
        MPI_Send(&data, 1, MPI_INT, (rank+1)%totalRanks, 0, MPI_COMM_WORLD);
    }

    while(data > 0) {
        MPI_Recv(&data, 1, MPI_INT, (rank + totalRanks - 1) % totalRanks,
                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("MPI Rank %d of %d received token %d.\n",
                rank, totalRanks, data);
        data--;
        MPI_Send(&data, 1, MPI_INT, (rank + totalRanks + 1) % totalRanks,
                0, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Figure 3.2.: Simple MPI Example using synchronous send and receive operations to bounce a message between ranks.



(a) MPI Barrier (b) MPI Global Gather Operation (c) MPI Global Reduction

Figure 3.3.: MPI Collective operations. The first figure depicts the parallel flow of a MPI barrier. The middle figure depicts an `MPI_Allgather()` operation. The figure on the right depicts an `MPI_Allreduce`. The ranks of the given communicator are blocked once they call the collective operation until all other ranks of the communicator have called it as well.

3.1.2. Collective Operations

Using just point-to-point messages, it is possible to implement any communication pattern already. However, there are communication patterns that are used frequently in HPC applications, and re-implementing these for each new application would not make sense. Furthermore, the MPI runtime may possess knowledge about the lower network layers that allow for optimizations in the way the operations are implemented, that would not be available to the application developer[4]. To this end, the MPI standard specifies *collective operations*. These describe specific and frequently used patterns such as broadcasts, gathers, scatters, parallel reductions or all-to-all communications. The semantics of these operations are described in the MPI Standard, but library developers are free to implement optimized versions as long as they adhere to the requirements of the standard. There are three types of operations that are supported: synchronization (barriers), communication (gather, scatter) and computation (reductions). Common to all collective operations is that all ranks of a communicator are involved (Barker, 2015; MPI Forum, 2015). In the following, I will describe examples of each type.

Barriers are used to make sure all ranks in a communicator have reached a certain point in the code before computations may resume. This is done by calling `MPI_Barrier(comm)`, where `comm` is the communicator that is to be synchronized. The function call will only return once all other ranks of `comm` have called the barrier as well. The behavior is shown in Figure 3.3a. *Gather operations* are used to make pieces of operations available to all ranks. `MPI_Allgather` specifically collects a value from all ranks of a collective, and makes them available to all ranks, as depicted in Figure 3.3b. *Global Reductions* are used if values from multiple ranks need to be processed into a single value, e.g. the determination of a global maximum. MPI offers the operation `MPI_Allreduce` that performs a global reduction on a value passed by each rank, and then makes the result of the reduction operation available to all participating ranks (depicted in Figure 3.3c. There are a number of

[4] It may be possible to optimize the pattern for a specific system, but that would result in a non-portable implementation.

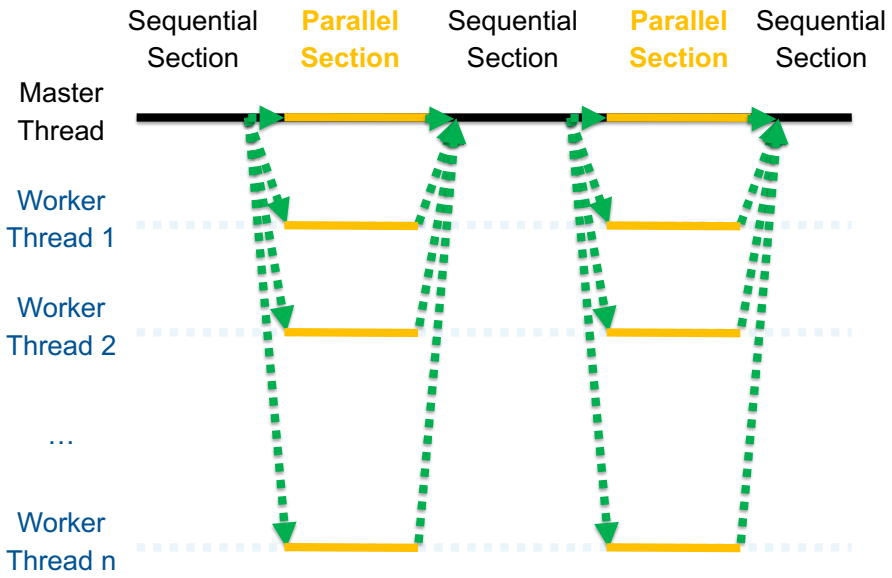


Figure 3.4.: Fork-Join Parallelism using OpenMP. The Master thread controlling the flow of execution is depicted in black, worker threads are depicted in blue. In parallel sections, work is distributed onto the worker threads. Once all workers are done, sequential execution is resumed.

available operations (e.g. `MPI_MAX` for the largest value, `MPI_SUM` for the sum of all values, and `MPI_LAND` for a logical And operation over all values).

3.2. OpenMP: Fork-Join-Parallelism

While MPI is the de-facto standard for distributed applications, the most common technology for parallelism within a shared memory domain is OpenMP (Dagum and Menon, 1998; OpenMP, 2018). Here, threads have access to the same memory segment, and thus explicit message passing is not necessary. In many modern HPC applications, OpenMP is used to parallelize code on the node-level, while MPI is used to parallelize across node boundaries. OpenMP is typically shipped as an extension of a C, C++ or Fortran compiler. Clang, GCC or the Intel Compiler, for example, ship with support for OpenMP. OpenMP consists of two components: a *runtime library* and a set of *compiler pragmas*. The pragmas allow for a high-level and declarative programming style that is then transformed by the compiler into multi-threaded code.

The main type of parallelism implemented in OpenMP is called *fork-join parallelism*. Here, the application's main thread is in control of the execution flow. Sections of code that are computationally intensive and have little dependencies may be parallelized. In OpenMP, the overall workload is distributed onto *worker threads*. Each worker thread performs part of the work, and afterwards sequential execution is resumed. This behavior is illustrated in Figure 3.4. The advantage of OpenMP is that the application developer does not need to specify *how* to parallelize a code

```
float compute_element_average(
    float *a, float *b,
    float *c, size_t n
) {
    float avgSum = 0.0f;
    for (size_t i = 0; i < n; i++) {
        c[i] = (a[i] + b[i]) / 2.0f;
        avgSum += c[i];
    }
    return avgSum;
}
```

(a) Sequential Version

```
float compute_element_average(
    float *a, float *b,
    float *c, size_t n
) {
    float avgSum = 0.0f;
    #pragma omp parallel for
    ↪ reduction(sum:avgSum)
    for (size_t i = 0; i < n; i++) {
        c[i] = (a[i] + b[i]) / 2.0f;
        avgSum += c[i];
    }
    return avgSum;
}
```

(b) OpenMP-parallel Version

Figure 3.5.: Reduction performed using OpenMP. The for loop itself is identical in both versions. In Figure 3.5b, the OpenMP pragma `parallel for` is used to parallelize the loop. The clause `reduction(sum:avgSum)` leads to the partial sums computed by the respective worker threads to be combined into the final result through summation.

segment. Instead, it is sufficient to specify *what* should be executed in parallel. The specified work is then split up based on the amount of resources available for the computation (which may be set for example by the batch scheduler according to the user's preference, or based on the `OMP_NUM_THREADS` environment variable). There are multiple different annotations for this, the simplest being one for parallel for-loops. The annotations are implemented in the form of compiler pragmas. Pragmas are custom directives that may be interpreted by the compiler, but are not formally part of the language. These pragmas may then be further annotated using clauses that specify or constrain the behavior of the pragma. The compiler then generates code based on the pragma and the code that follows it.

Consider a simple function `compute_element_average(float *a, float *b)`, depicted in two versions in Figure 3.5. In both the sequential and the parallel case, the function computes the element-wise average of two vectors `*a` and `*b`, and stores the result in `*c`. Furthermore, the sum of the averages is returned as the result of the function. In the parallel case, the loop is parallelized using `#pragma omp parallel for`. This leads to the loop being divided into chunks that are then executed by the worker threads. However, the summation of `avgSum` introduces dependencies between the loop iterations. Some of these dependencies may be resolved using reductions. In the given case, for example, the clause `reduction(sum:avgSum)` may be used. It directs the compiler to generate code to collect the partial sums from the chunks at the end of the parallel computation, and to fold the partial results into a singular result using the specified reduction operation.

Aside from the reduction clause, there are a number of other clauses that developers may annotate parallel for loops with. For example, one may influence the scheduling of threads by the OpenMP runtime through the `schedule(<strategy>,<chunksize>)` clause. Amongst others, the developer may choose from *static* scheduling, which assigns `chunksize` number of iterations cyclically onto

worker threads. When no chunk size is specified, the total work is split up into one chunk per thread. *Dynamic* scheduling assigns each chunk onto the first idle worker thread when it is encountered. The correct choice of scheduler depends on the work performed in the loop. For loops with a constant work per iteration, the static scheduling will typically yield better performance, while for unpredictable work loads per iteration, the dynamic scheduling may be better (Thoman et al., 2012). The default assumption of OpenMP is that all variables are shared between threads. This may lead to unforeseen side effects, such as data races if variables are modified concurrently by multiple threads. To control the data sharing behavior, OpenMP has data sharing clauses. For example, the *private()* clause specifies that all variables passed to the clause are private to each iteration. Now, in every iteration, these variables hold unique values. However, the private annotation does not initialize them to their previous value. If that behavior is desired, the *lastprivate()* clause may be used instead. Variables that are constrained in that way are still private to each iteration, but they are initialized to the value they held before the loop.

OpenMP also offers support for less regular parallelization patterns and other constructs frequently encountered in parallel programming. Using *sections*, the application developer may specify a number of segments that ought to be executed concurrently. Based on a code fragment annotated thus, the compiler will generate code that distributes each segment onto a worker thread, and then executes them in parallel using the aforementioned fork-join pattern. When the nature of the parallel work is irregular, and the number of parallel chunks is not necessarily known in advance, *tasks* may be used. While the tasking functionality implemented using OpenMP is not as advanced as in some of the more recent task-based runtime systems, OpenMP still allows for the dynamic creation of tasks, the specification of inter-task dependencies (using the *depends* claus) and the awaiting of their termination (using the *taskwait* pragma). Finally, OpenMP offers compiler pragmas for barrier synchronization and for atomic operations as well as pragmas to guide the compiler to emit vectorized code.

4. UPC++: PGAS-Based Distributed Memory SPMD

While MPI has been the prevalent API for distributed memory parallelism for the last two decades, there have always been efforts to provide alternative runtime systems and libraries for inter-node communication. Many of those are based on the PGAS model of parallelism. In contrast to the message-passing paradigm described in section 3.1, in PGAS systems there exists a global view of the memory, and there are facilities in place to access data on remote ranks. The way these are implemented differs between the different runtime systems. Some, like OpenSHMEM (Baker et al., 2017), UPC (UPC Consortium, 2013) or Co-Array Fortran (part of the Fortran language starting with the Fortran 2008 standard) allow access of data on remote ranks directly, while others, such as UPC++ (Bachan et al., 2019) or X10 (Tardieu et al., 2014), restrict access, and instead require explicit transfer operations. The operations are always one-sided, i.e. only the initiating node needs to become active, while on the communication partner's side, the runtime or the network adapter (if Remote Direct Memory Access (RDMA) is available) handles the request.

In the following, I will describe UPC++, an example of a PGAS library that follows the SPMD model (similar to MPI), but uses modern language features to implement the PGAS characteristics. Moreover, UPC++ is also the basis for Actor-UPC++, one of the actor libraries discussed in this thesis. UPC++ is developed at the Lawrence Berkeley National Laboratory as part of the US Department of Energy's Exascale Computing Project. Its implementation is based on a formal specification (Bachan, 2019). Using the language as an example, I will explain the basic features common to many PGAS languages.

4.1. The UPC++ Machine Model

UPC++ is a library for modern C++, and is typically executed with multiple instances on a distributed memory system. The abstract machine model of UPC++ comprises a fixed number of processing elements (or *ranks*), as depicted in Figure 4.1, each with its own local memory. Per physical node, there may be one or more ranks based on the needs of the application. Depending on the parallelization strategy used by the application, different configurations may be used. For applications with one thread per process, it may be reasonable to use one rank per (logical) core. Applications using shared-memory parallelism may use configurations with, e.g., one rank per Non-Uniform Memory Access (NUMA) domain. Ranks local to a node are grouped together in a *local team*. The memory of a rank is segmented into two segments, private and shared. The private segments contain all local objects. These are allocated and accessed using the facilities of C++, e.g. through raw or shared pointers, or through references. The shared segment contains data allocated using library allocator functions `upcxx::new_<T>` and `upcxx::new_array<T>`. These operations return a *global pointer*, an object that serves as a global handle to locate the object within the global

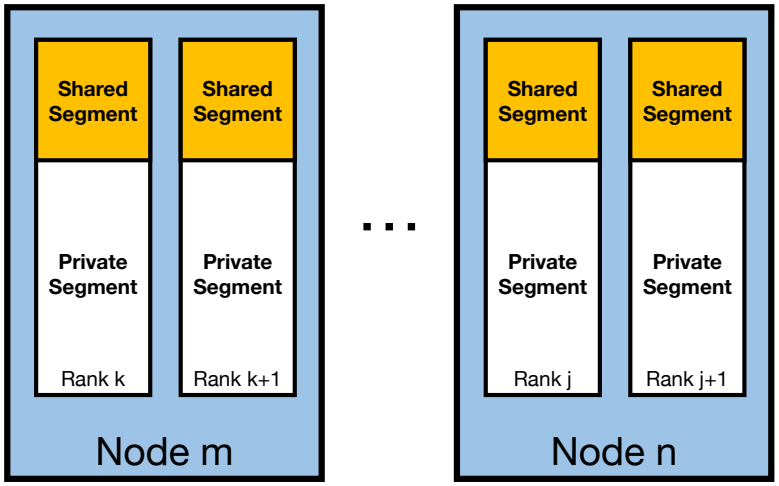


Figure 4.1.: UPC++ Machine Model. Each physical node may have one or more UPC++ ranks. The memory available to each rank is divided into a private and a shared segment. Data resident in the shared segment is accessed via global pointers that may be dereferenced if it is stored on the same node. Otherwise, explicit and asynchronous operations are needed.

address space of the application instance. That object contains the rank that the allocated object was located on, and a raw pointer to the object on that rank. Global pointers may be dereferenced only by the rank owning the data, if the rank storing the data item is part of the dereferencing rank's local team. All other ranks need to use explicit remote read (`upcxx::rget<T>()`) or write (`upcxx::rput<T>()`) operations to access the data.

4.2. The UPC++ Execution Model

One of the design goals specific to UPC++ was making the cost of operations explicit to application developers, and to enable them to interweave longer-running communication operations with computations (Bachan et al., 2019). This is achieved using the following two measures. The first measure is to make all communication operations asynchronous. For each communication operation, a handle is created and pushed onto a queue of deferred operations. While a communication operation is being processed, it is moved to a second queue. Finally, there is a third queue for handles with completed operations. UPC++ does not handle communication itself, but uses GASNet-EX internally. GASNet-EX is a low-level library for one-sided communication that provides backends for widely-used interconnection fabrics such as InfiniBand or Cray Aries. Furthermore, MPI is supported as a fallback mode of operation (Bonachea and Hargrove, 2018; Bachan et al., 2019).

The second measure is to require the explicit assignment of computation time to the library by the application developer. Neither UPC++ nor GASNet-EX start their own threads. Instead, the UPC++ API specifies for each function if and what kind of operations may be performed

by the library. Two cases are distinguished: *internal* and *user-level* progress. Internal progress comprises the communication operations that do not change the user-observable application state, e.g. posting operations on other ranks or network operations. User-level progress consists, e.g., of notifications to the application about completed operations, Remote Procedure Calls (RPCs) or the execution of callbacks. Generally, internal-level progress is performed whenever communication operations are issued, while user-level progress has to be provided manually using the `upcxx::progress()` function. For an optimal performance, it is important to enable the library to perform internal and user-level progress regularly.

UPC++ makes the state associated with handling a thread explicit in the `upcxx::persona` class. The data structure contains internal data structures associated with a thread, and serves as a destination point for notifications. Each operating system thread is assigned a persona. Further personas may be created at will by the application developer. The master thread of an application is assigned the rank's master persona at library initialization. In addition to being the destination for notifications, they also serve as the destination for incoming RPCs.

4.3. PGAS Characteristics of UPC++

Aside from global pointers and one sided memory transfer operations, UPC++ offers two important features for communication with other address space partitions, *RPCs* and *distributed objects*.

RPCs allow a rank to execute an arbitrary code segment on another rank, along with the possibility to transmit parameters as well as a to return a value. As with other operations that affect multiple ranks, they are executed asynchronously. They are one-sided operations, which means that the only effort that needs to be made from the remote rank is sufficient user-level progress. There are two variants of the function: `upcxx::rpc` and `rcp_ff`. Both take a receiving rank, a function object (e.g. an object with a publicly overloaded `operator()`, a function pointer or a lambda), and the parameters the function object is to be called with. The first variant, `upcxx::rpc`, allows for notifications when the local part of the RPC is completed and when it has been executed on the remote side. The latter may also contain a return value. The second variant does not allow the sender to track the operation completion, and hence provides slightly better performance, there need not be any messages sent back to the initial rank once the operation completes. As an example for the first variant, consider a function that adds a given number to all the elements of memory segment on different rank (see Figure 4.2a). The RPC executes the lambda-function passed to the `rpc_ff` call on the rank that owns the data in the global pointer. The lambda-function first unwraps the global pointer, and then performs the operation on each element in the target memory segment. The second variant for the RPC is shown in Figure 4.2b. Here, the contents of a remote memory segment are summed up, and the result is returned to the rank that initiated the operation. In both examples, the RPC is executed asynchronously. In the first case, the remote rank needs to ascertain from the modified data that the call has been executed, and in the second case, a future is returned, which allows the initiator of the operation to wait for its completion.

```
1 void multiplyRemoteWithConstant(float constant, global_ptr<float> remoteArray,
  ↪ size_t length) {
2     rpc_ff(remoteArray.where(), [constant, remoteArray, length]() {
3         float *memory = remoteArray.local();
4         std::transform(memory, memory + length, memory, [constant](float element)
          ↪ {
5             return constant * element;
6         });
7     });
8 }
```

(a) RPC (Fire and Forget)

```
1 future<float> getSum(global_ptr<float> remoteArray, size_t length) {
2     return rpc(remoteArray.where(), [remoteArray, length] {
3         float *mem = remoteArray.local();
4         float res = std::accumulate(mem, mem + length, 0, std::plus<float>());
5         return res;
6     });
7 }
```

(b) RPC Returning a value

Figure 4.2.: UPC++ RPCs. The call in Figure 4.2a multiplies a given constant number to the contents of a remotely stored array. The function in Figure 4.2b accumulates the contents of an array into a single floating-point value.

Distributed Objects are global object identifiers that hold a value on all ranks. In UPC++, they are templated as `upcxx::dist_object<T>`, they are created collectively by all ranks, and they consist of a global identifier and an object instance of the specified type `T`. One of the problems solved by distributed objects is the initial distribution of information between ranks. If, for example, an RPC wants to modify data on another rank, this is not possible to do without having a handle to it. Global Pointers are one possible option here, but without a known destination on the other rank, it is impossible to send them. Distributed objects may be used as a solution to that problem. For example, if one wants to accumulate pointers to arrays on other ranks onto a single rank (e.g. rank 0), it is possible to implement it as shown in Figure 4.3. The example resembles a one-sided collective gather operation. It is a frequently utilized pattern that may be used during initialization in order to be able to send more information at a later time. As demonstrated in this example, distributed objects are especially useful combined with RPCs. Along with global pointers, they are the only way to access data of the receiving rank within the RPC. In the example, the call to the depicted function `accumulateGPtr()` has to be performed on all ranks. First, all ranks create a vector of pointers, and a distributed object that contains a pointer to the object. On rank 0, the pointer may simply be added to the vector directly, and then the UPC++ runtime may be queried for progress until all the pointers from the other rank have been added. The other ranks simply post an RPC to rank 0 with the global pointer and the distributed object as parameters and waits for its completion. For the operation to work, the object lifetime needs to be watched. As all inter-rank communication is performed asynchronously, the application developer needs to make sure that a distributed object exists on the other ranks when the access is made. For the RPC, the runtime makes sure of the distributed object's creation on the destination rank if a such an object is passed as parameter. However, it is up to the application developer to make sure that the object still exists on all ranks where it still may be accessed. In this example, it is ensured that a rank only leaves the scope of the function once its data will no longer be accessed.

4.4. Asynchronous Completions

When dealing with asynchronous communication functions of UPC++, it is often necessary to track the status of the operation (e.g. to manage the lifetime of local buffers). There are three operation states that are tracked: *source completion*, *remote completion* and *operation completion*. Source completion may be signalled when resources on the side of the source of the communication operation are no longer needed. Remote completion occurs when the data has reached the remote process and may be used there. It is signalled to the context of the master persona of the remote rank. Operation completion is signalled when the operation is completed in the eyes of the initiating process, and any transferred data is now available to the initiating process. The specification (Bachan, 2019) defines for each operation which kind of completion events may be signalled (and which values may be attached to them). For example, the library function `upcxx::rget(upcxx::global_ptr<T> src, T *dst, size_t count, Completions cxs)` only provides an operation completion signal, as soon as the requested data has been transferred successfully. On the other hand, `upcxx::rput(T *src, upcxx::global_ptr<T> dst, size_t count, Completions cxs)` provides a source completion event once the data to be copied has been injected into the network (or an intermediary buffer). The event signals the sender that the source data may now be modified or deleted. Furthermore, there is a remote completion event sent to the master persona of the receiving rank

```
std::vector<global_ptr<float>> accumulateGPtr(global_ptr<float> localPtr) {
    std::vector<global_ptr<float>> res;
    dist_object<std::vector<global_ptr<float>> *> resHandle(&res);
    if (!rank_me()) {
        res.push_back(localPtr);
        while (res.size() < rank_n()) {
            progress();
        }
        return res;
    } else {
        rpc(0, [](dist_object<std::vector<global_ptr<float>> *> &rv,
        ↪  global_ptr<float> ptr) {
            std::vector<global_ptr<float>> &remoteVec = **rv;
            remoteVec.push_back(ptr);
        }, resHandle, localPtr).wait();
        return res;
    }
}
```

Figure 4.3.: UPC++ Distributed Objects. In this example, distributed objects are used together with RPCs to accumulate pointers to global arrays on a single rank. Every rank greater than rank 0 sends an RPC to rank 0 with its global pointer and a reference to the distributed object containing an array of pointers. Rank 0 queries the UPC++ runtime for progress until its vector contains the global pointers from all other ranks.

```
void sendLpcs(std::vector<global_ptr<float>> &ptrs) {
    float result = perform_complex_operation();
    int completedSends = 0;
    for (auto &ptr : ptrs) {
        auto completions =
            source_cx::as_buffered()
          | operation_cx::as_lpc(current_persona(), [&]() {completedSends++;});
        rput(result, ptr, completions);
    }
    while (completedSends < rank_n()) {
        progress();
    }
}
```

Figure 4.4.: UPC++ LPC completions. In this example, I send the result of a complex operation to a number of remote memory locations and use LPCs to track the progress.

once data is available at the destination, and an operation completion on the side of the initiating rank once transfer has completed and the data is available on the remote rank.

UPC++ provides several ways to be notified of completion events (but some types of notifications only work for specific completion events). The most basic way to be notified of a completion event is to use *Blocking*. Here, the operation simply blocks, until the requested operation may access the network to inject the data directly. Similarly, *Buffered* completion blocks until source completion is reached. In contrast to the blocking behavior, the application may choose to buffer data internally, or to wait for the network. Both these operations are available for source completion events only. For operations that offer the remote completion event, it is possible to attach a *RPC* to notify the remote rank of the completion of the operation (and to perform work on the remote data or to provide meta information). RPCs are discussed in greater detail in section 4.3. Similarly to posting an RPC for remote completion, it is possible to to attach a so-called *Local Procedure Call (LPC)* to an operation. LPCs are function invocations, i.e. function objects along with calling arguments, that are delivered to a specific persona on the same rank. If the operation returns a value, the function object's type signature has to provide an argument of the type of the returned object, otherwise it has to have an empty argument list. The LPC will be enqueued for execution as soon as the completion event they are attached to occurs. They may be used for completions on the issuing rank, i.e. source completion and operation completion. As an example, I used LPCs to broadcast the result of some complex operation to a number of remote memory locations (see Figure 4.4). It is necessary to track the completion of all operations, and block to make sure that the result is not overwritten. Therefore, I use `source_cx::as_buffered()` as the source completion, and `operation_cx::as_lpc(...)` as the operation completion. In the LPC, I simply increment the number of completed operations. After all the communication operations are posted, I query the runtime for progress until all operations are completed. The downside to this approach is that it only tracks one completion at a time. On the other hand, it enables the application to perform more complex tasks rather than just tracking the completion through a counter.

In some cases, it may be interesting to keep track of multiple asynchronous operations at the same time. UPC++ offers two different structures for this, namely *futures* and *promises*. Futures were originally proposed by Baker and Hewitt (1977) as an encapsulation of a value that may not be available yet. They may be returned by longer-running functions that perform work concurrently to the caller of the function instead of blocking until the operation is completed. When the value is eventually needed, it is either returned directly if the operation producing it has terminated, or the execution of the caller is blocked until the value is available. Alternatively, it is possible to compose multiple futures. Given a number of futures, it is possible to create a future that becomes ready when all the futures used for its creation become ready ($when$: α future $\rightarrow$ β future $\rightarrow$ ... $\rightarrow$ ω future $\rightarrow$ $(\alpha \times \beta \times ... \times \omega)$ future). One may also pass a function that will be executed once the future is ready. The result of that operation is then another future whose return type depends on the value returned by the function to be performed ($then$: α future $\rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$ future). UPC++'s use of futures follows that same pattern. Futures are returned by asynchronous operations, or may be created explicitly. They hold a value, either a singular one or a tuple, they may be composed using `upcxx::when_all` and it is possible to react to their completion using `upcxx::then`. This principle is also best illustrated using an example. Here, I accumulate a result from a number of remote memory locations. The naïve way to implement this (see Figure 4.5a) copies the data from each rank, waits for completion of the transfer, and then adds it to the partial result. This approach does not make use of the asynchronous nature of UPC++, and leads to an unnecessary sequentialization of the remote accesses. A better way (shown in Figure 4.5b) is to use future chaining, where instead of starting with a value directly, I start with a trivially fulfilled future called `globalSum` holding the value 0.0f. For each remote access, a new, combined future (`combined`) is created that becomes available as soon as the future holding for the accumulated value and the value of the operation are ready. To this future, I then attach a callback to obtain a third future that will hold the result of accumulating both prior values. That third future is then assigned to `globalSum`. This is repeated for all remote operations. Finally, the rank waits for the final value to be ready. The advantage compared to the version shown in Figure 4.5a is that all communication operations are started without delay, and the values are accumulated as they arrive[1], while communication is still in progress.

Not all communication operations produce values. In some cases, it is enough to make sure that a number of operations are completed, without the need for the production of a return value. For this, UPC++ offers the concept of promises. Promises are initially created with a single dependency. The application programmer may then attach–or *register*–further dependencies, either explicitly, or by passing the promise object to a completion event. Once all desired additional dependencies are registered, the initial promise is fulfilled, and a future object that may be tracked by the application developer is created. Aside from tracking completion of UPC++ functions, promises may also be helpful in tracking the completion of concurrently executing user code. One may use promises, for example, to wait until communication and local computations are completed, and only then initiate the subsequent step. As an example, I implemented the send operation discussed above using promises Figure 4.6. Here, I just register each remote operation with the promise object, finalize it to obtain an empty future, and wait for it to become ready. Finally, both promises and futures may be attached to events that are signalled to the issuing rank, i.e. source completion and operation completion.

[1] The performance benefit will be more apparent for more complex operations.

```
float receiveSequential(std::vector<global_ptr<float>> &ptrs) {
    float globalSum = 0.0f;
    for (auto &ptr : ptrs) {
        globalSum += rget(ptr).wait();
    }
    return globalSum;
}
```

(a) Sequential Gathering

```
float receiveFutures(std::vector<global_ptr<float>> &ptrs) {
    future<float> globalSum = make_future(0.0f);
    for (auto &ptr : ptrs) {
        auto remoteResult = rget(ptr);
        future<float, float> combined = when_all(globalSum, remoteResult);
        globalSum = combined.then([](float a, float b) {return a+b;});
    }
    return globalSum.wait();
}
```

(b) Gathering Using Future Chaining.

Figure 4.5.: UPC++ Futures. The function on the left gathers values from other ranks sequentially, while the version to the right uses asynchronous future chaining to overlap communication and computation.

```
void sendPromises(std::vector<global_ptr<float>> &ptrs) {
    float result = perform_complex_operation();
    int completedSends = 0;
    promise<> allCompleted;
    for (auto &ptr : ptrs) {
        auto completions = operation_cx::as_promise(allCompleted);
        rput(result, ptr, completions);
    }
    future<> done = allCompleted.finalize();
    done.wait();
}
```

Figure 4.6.: UPC++ Promises. In this example, I send the result of a complex operation to a number of remote memory locations and use promises to track the progress.

5. X10: Asynchronous Partitioned Global Address Space

With the goal of improving programmer productivity while keeping a high performance, the United States Department of Defense funded a number of projects in the early 2000s to develop new methods to program modern HPC systems (Dongarra et al., 2008). Two of these languages, Chapel[1] and X10 [2], follow the Asynchronous Partitioned Global Address Space (APGAS) paradigm. In contrast to "traditional" PGAS, the aim is not just to provide a global and coherent view on the memory, but also onto the entire parallel computation. In the following, I will explore the APGAS paradigm on the example of *the X10 Programming Language*. I chose X10 as an example over Chapel mainly for its role in InvasIC. The project chose X10 as the primary application language, as the APGAS paradigm it implements was seen as a good fit for the proposed model of a cache-incoherent Multiprocessor System-on-Chip (MPSoC). Thus, it is also the basis for ActorX10 and SWE-X10. However, at this time, only Chapel is still actively developed. The technical details of the language are taken from the X10 language specification (version 2.3.1[3]) by Saraswat et al. (2013), unless indicated otherwise.

X10 is a strongly object-oriented (everything is an object), statically typed and compiled programming language. It was originally developed as a language targeting novel hardware architecture developed in the same project (Dongarra et al., 2008). The scope was extended later to include classic cluster architectures (Dongarra et al., 2008) as well as cloud architectures (*ElasticX10* 2015). There are two compiler targets for X10. With the *NativeX10* target, the X10 compiler transpiles the source code to C++-98, which may then be compiled to machine code using a suitable compiler such as GCC or the Intel Compiler. Alternatively, with the *ManagedX10* target, it is translated to Java, which is then compiled to Java Byte Code and executed by a suitable Java Virtual Machine instance. Depending on the selected target, it is possible to interact with source code written in either target language. The compiler, IDE, runtime and utilities are available as free software under the Eclipse Public License.

X10 represents the computational resources available to the application explicitly through *places*. Places represent shared memory domains, akin to ranks in UPC++ or MPI. Like ranks, they may encompass one or more threads and a bounded amount of memory that is uniformly accessible by all hardware threads (Saraswat et al., 2013). Threads are not exposed directly in X10, instead concurrency is expressed through activities that are mapped to a finite number of operating system threads. Program execution is initiated by executing the `main` method on *place 0*. From there, the

[1] `https://chapel-lang.org`

[2] `http://www.x10-lang.org`

[3] We use this version to maintain compatibility to the implementation used within the Invasive Computing project. The current version of X10 at the time of writing is 2.6.2

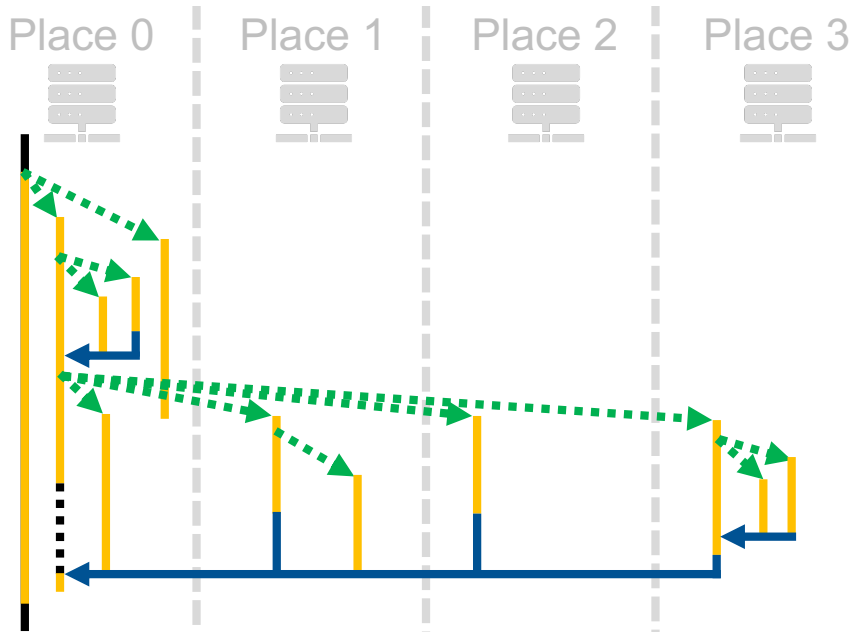


Figure 5.1.: APGAS style parallel Execution in X10. The figure depicts a parallel execution on multiple places. Yellow bars denote activities performing computations, the dashed green arrows demote new activities being spawned (possibly on other places), and the blue arrow denotes controlled termination of activities.

application developer may start other activities locally as well as on remote places. Activities may hold references to local data as well as data on remote places, but only locally available data may be dereferenced, i.e. read or modified. Data from remote ranks may be obtained as a copy for read accesses, or modified directly by starting a remote activity at the place containing the data.

5.1. X10 Core Language and Type System

X10 is a statically typed, object-oriented language. Its early versions were an extension of Java, and to this day, it shares some of its characteristics. Most notably, all code is implemented in the scope of *class* (or *struct*) definitions. The basic class definitions closely follow Java semantics: one defines a class by specifying its super-classes, implemented interfaces, attributes and methods. As in Java, each class is situated in a package, and one is able to specify visibility of attributes, methods and the class itself using the accesses specifiers *public* for global visibility, *private* for visibility only to other class instances, or no access specifier for package-level visibility. Instance objects of classes are assigned and passed to functions as references. Similarly to Java, generic types are supported. It is possible to parametrize a class with one or more parameters, and thus implement generic containers.

In contrast to Java, there are no primitive values. However, it is possible to define custom value types using the *struct* keyword. In contrast to classes, structs are allocated on the stack, do not support inheritance, and are generally immutable. They are passed and assigned by value.

Together with support for operator overloading, it is possible to define custom arithmetic types. For example, the struct `Complex` provided by the X10 standard library for the representation of complex numbers is simply a struct definition with overloaded operators. In addition to "traditional" objects, one can also define function objects. They may be assigned to variables, and used as class attributes, or as parameters for methods. For example, a simple function may be defined as `val squared = (f:Float) => x*x`. It would be of type $(\text{Float}) \rightarrow \text{Float}$, and is called the same way as one would call other methods: `squared(4.5f)`.

A notable difference from Java is the inclusion of *dependent types*. Dependent types are types that depend on a value (Barthe and Coquand, 2002; Pöppl, 2011). For languages with a classic, strong type system, such as Standard ML, or Haskell, it is not possible to specify the types of some functions with varying number of parameters statically. A simple example relevant for scientific computing are multi-dimensional arrays. Many scientific applications use them to represent their unknown quantities. Often, several different arrays of the same size and dimensionality are needed, but that requirement is not visible in the code, but only at the allocation site. Instead, memory is often accessed directly by reconstructing the memory layout based on knowledge that is not explicitly stated in the code. Alternatively, one may try to formalize and represent that knowledge in the code directly. To implement that, one option is to specify the number of dimensions as an instance property. It is possible to implement generic functions then. However, then one has to perform runtime-checks for the number of dimensions for every call, and it would be impossible to enforce certain limitations statically (e.g. only two-dimensional arrays). Furthermore, the generated code for iterations over the array may not be as efficient, as the compiler may not be able to determine the structure of the iteration, and therefore may not be able to perform any loop transformations. Another option is the introduction of subtypes for the desired categories, e.g. one-dimensional, two-dimensional or three-dimensional arrays. This allows for the implementation of shared functionality by using a common super-class, but sacrifices flexibility. Here, it is possible for the compiler to generate efficient loops, as the structure is known at compile-time. However, extensions, such as support for four-dimensional arrays, necessitate the implementation of a new subclass.

Using dependent types, the number of dimensions may be specified as part of the type definition. When the type is used, it may be constrained, e.g. by only allowing arrays of specific dimensions. This allows the use of arbitrarily dimensioned arrays without forcing checks of the number of dimensions at runtime. Instead, constraints are enforced during type checking. Depending on the type, type definitions may result in complex, recursive declarations[4]. This makes type-checking dependently typed programs undecidable (Augustsson, 1998; Barthe and Coquand, 2002). To ensure that compilation terminates eventually, one can either insert an explicit termination condition (such as in Cayenne), or limit the type system to exclude any properties that may result in divergent behavior. In X10, the latter approach is chosen for dependent typing. The language enables for the constraining of classes using so-called *properties*, immutable values with public visibility. The values specified in class properties may then be used to constrain parameters and return types of functions. Constraints are of the form $\bigwedge_{c \in C} c$, where C is the set of the constraint expressions of a given Type. These expressions may be, amongst others, value (in)equalities, type (in)equalities, expressions specifying type hierarchies and nullability checks. Within these expressions it is possible to use literals, immutable class attributes, `this` (referring to the instance

[4] cf. `printf()` definition in Augustsson (1998)

the method is called on in instance methods), `self` (for the object instance of the type being constrained), `here` (as the stand-in for the place the code is executed on), property methods (pure methods with a single return statement following this set of constraints) and attribute accesses such as `t.f`. Adhering to these restrictions allows the X10 type system to remain decidable, while still enabling interesting use cases, such as the aforementioned multi-dimensional arrays. X10 offers support for multi-dimensional arrays with arbitrary coordinate sets. One may use dependant types to constrain the structure of the coordinate sets. For appropriately constrained sets of coordinates, essentially amounting to zero-based, continuous and rectangular arrays, the X10 compiler is able to generate efficient nested loops. This optimization is an important building block for obtaining good application performance in X10.

Type constraints allow for the generation of safe code without the need for checks performed at runtime. I demonstrate this using the example of matrix-matrix multiplication. The class is consistently annotated using type constraints to ensure that only matrices of fitting dimensions are multiplied with each other. In classical programming languages, these constraints would have to be checked as part of the implementation of the class's functionality, but here, they are checked and enforced by the compiler. Using constraints consistently enables the compiler to avoid inserting runtime checks of the type constraints. This is another important building block to obtain good application performance in X10. It is possible to prevent the compiler from inserting runtime checks by treating them as type errors instead. The product $C_{m,n}$ of two matrices $A_{m,k}$ and $B_{k,n}$ may be defined element-wise as $c_{m,n} = \sum_{i=1}^{j} a_{m,i} b_{i,n}$. This leads to two constraints on the size of the matrix: (1) The size of matrix C depends on the number of rows in A and the number of columns in B. (2) The number of columns in A needs to be equal to the number of rows in B. This may be expressed directly using X10 type constraints, as shown in Figure 5.2. In the code listing, dependent types are used in multiple places. The declaration of the backing storage constrains the array holding the data to be two-dimensional and rectangular. For the element-wise matrix sum, all three matrices need to have the same size. This is guaranteed with the type constraint `Matrix{(self.rows == this.rows) && (self.columns == this.columns)}`, where `this` references the left-hand-side of the operation, and `self` the right-hand-side. For the matrix-matrix multiplication, constraint (1) with `Matrix{(self.rows = this.rows) && (self.columns == other.columns)}` for the parameter and (2) with `Matrix{this.columns = sef.rows}` are satisfied. Using these constraints, no more size checks need to be performed at program execution time.

The implementation of constraints in X10 has severe drawbacks, however, especially when dealing with type casts. For instance, using the matrix class from the previous example (Figure 5.2), it is possible to construct the following sequence of statements that compile without errors with all static checks enabled.

```
1 val a = new Matrix(4,4, (p:Point(2)) => 1.0f);
2 var mutableMtrix : Matrix = a;
3 val evil : Matrix{rows == 7 && columns == 7} = f as Matrix{rows == 7 && columns
  ↪  == 7};
```

In practice, this implies that one cannot really trust the guarantees given by the type constraints, especially when objects are taken from contexts outside the developer's control.

```
1  public class Matrix(rows:Int, columns:Int) {
2      val storage : Array[Float]{rect, rank==2};
3
4      public def this(rows:Int, columns:Int, initialization:(Point(2))=>Float) {
5          property(rows, columns);
6          val matRegion = Region.makeRectangular([0,0],[rows - 1, columns - 1]);
7          this.storage = new Array[Float](matRegion, initialization);
8      }
9
10     public operator this + (other:Matrix{(self.rows == this.rows) &&
       ↪  (self.columns == this.columns)})
11         : Matrix{self.rows == this.rows && self.columns == this.columns} {
12         return new Matrix (rows, columns, ((p:Point(2)) => this.storage(p) +
           ↪  other.storage(p)));
13     }
14
15     public operator this * (other:Matrix{this.columns == self.rows})
16         : Matrix {(self.rows == this.rows) && (self.columns == other.columns)} {
17         return new Matrix (this.rows, other.columns, ((p:Point) => {
18             val row = p(0);
19             val column = p(1);
20             val target = this.columns;
21             var res : Float = 0.0f;
22             for ([i] in 0 .. (target - 1)) {
23                 res += this.storage(row, i) * other.storage(i, column);
24             }
25             return res;
26         }));
27     }
28
29     public def setMatrix(other:Matrix{(self.rows == this.rows) && (self.columns
       ↪  == this.columns)}) {
30         for ([y,x] in storage) {
31             this.storage(y,x) = other.storage(y,x);
32         };
33     }
34 }
```

Figure 5.2.: Dependently typed Matrix class in X10. The implementation of the Matrix-Matrix multiplication uses the X10 type constraints to statically ensure that the matrices used in the computation have the correct type.

```
1 def doWork() {
2     async f();
3     async {
4         async g1();
5         async g2();
6         g3();
7     }
8     h();
9 }
```

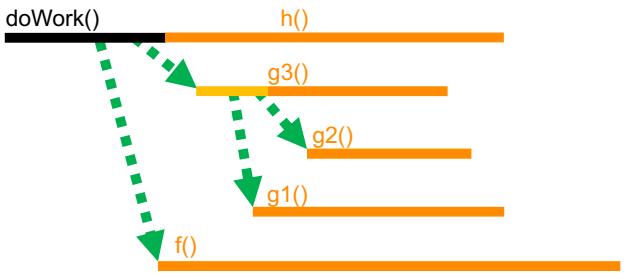


Figure 5.3.: Left: sample function that spawns asynchronous activities. On the right, the activities are visualized using an activity tree. Green arrows denote the spawning of a new activity.

5.2. Concurrency in X10

The main concept to express intra-node parallelism in X10 are *activities*. Their execution behavior is similar to user space threads, in that they may have their own local state, and that they are managed and scheduled onto a pool of operating system level worker threads by the X10 runtime. A new activity may be started by using `async S`. Anything within statement `S` will be executed concurrently to the activity that spawns it. One may combine this with other statements, e.g. `for ([i] in 0 .. 7) async f(i)` executes `f(i)` on eight activities in parallel. It is also possible to spawn activities recursively. This essentially leads to a tree-shaped structure of the activity spawn graph, a concept that is used to monitor the lifetime of activities. An example tree of activities may be seen in Figure 5.3. The function `doWork()` spawns two activities, one that executes `f()`, and a second one that itself recursively spawns other activities. Depending on the complexity of the executed work, child activities may outlive their parent. In the X10 terminology, an activity is *terminated locally* once its own execution has concluded. For example, the activity started in line 3 is locally terminated once the function call to `g3()` returns. On the other hand, an activity is *globally terminated* once itself and all descendant activities are terminated. In the example, global termination of the activity started in line 3 is reached once itself, the two activities executing `g1()` and `g2()` and all their descendants have reached local termination. In cases where global termination of all child-activities is needed before proceeding with the computation, the `finish S` statement may be used. It only terminates locally once all activities spawned in `S` are terminated globally, i.e. all activities spawned in `S` finished their work. This is best illustrated with another example: the work performed here (see Figure 5.4) is identical to the one performed in the prior example (Figure 5.3). The difference is that I ensure that the invoking activity only returns from the function call to `doWork()` once all child activities are terminated, and that `g3()` is only called once the two activities calling `g1()` and `g2()` are terminated. If necessary, the activity that contains the `finish` statement is sent to sleep until its descendants are done with their work.

The last important ingredient for local concurrency in X10 is atomic execution. There are two statements for this: `atomic S` executes statement `S` atomically, and `when (c) S` executes `S` atomically once condition `c` is met. The atomically executed statement must not spawn any activities, block, or place-shift. Doing so results in a runtime exception being thrown. An important restriction

```
1  def doWork() {
2      finish {
3          async f();
4          async {
5              finish {
6                  async g1();
7                  async g2();
8              }
9              g3();
10         }
11         h();
12     }
13 }
```

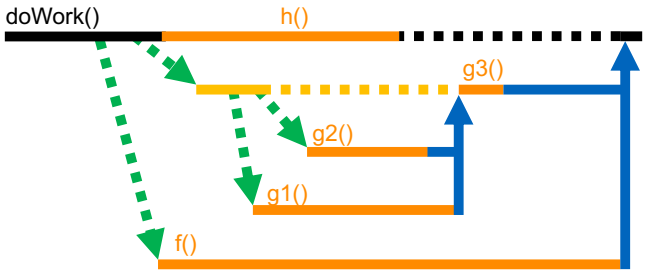


Figure 5.4.: Left: sample function that spawns asynchronous activities, and use of the `finish` statement to enforce execution order. On the right, the activities and their dependencies are visualized using an activity graph. Green arrows denote the spawning of a new activity. Blue arrows denote dependencies onto an enclosing finish, and dotted lines denote time activities have to spend waiting.

of the feature is that atomic execution is only guaranteed relative to other atomic blocks. In essence, this is done by only allowing a single atomic block per place to be executed at the same time. Furthermore, the when-statement only gets notified of state-changes within atomic sections, otherwise the change may be ignored. These restrictions make the use of atomic blocks in X10 somewhat problematic in performance-sensitive code, which results in the X10 performance guide's recommendation to use `atomic` blocks only sparingly, and to completely avoid `when` (*X10 Performance Tuning* 2015).

5.3. Partitioned Global Address Space in X10

Parallelism across node boundaries in X10 is supported via *places*. The number of places is fixed at application start-up. Each place corresponds to a shared-memory domain (akin to ranks in MPI or UPC++). All objects in X10 reside on the place they were declared on. Data is exchanged using active messages. An active message is sent using the `at (p) S` expression, where `p` is a valid place and S a statement that may also include a return value. The execution of the `at`, also referred to as *place-shifting*, is synchronous. When an activity enters a place shift, it is suspended, the `S` is executed on place `p` and then resumed as soon as `S` is finished. Before `S` can be executed, all objects accessed in it are collected by the X10 runtime. Each object is then serialized recursively, i.e. all fields of the object, as well as all objects it references, are converted into a linear representation and sent to the other place. There, they are deserialized and recursively reconstructed. In essence, a complete, deep copy of the object graph accessed in `S` is made. The code for the serialization and deserialization of a class is generated automatically by the compiler without interference by the application developer. Like UPC++, X10 supports global pointers (implemented as `GlobalRef[T]`) and distributed objects (as `PlaceLocalHandle[T]`).

It is possible to combine `at` with other X10 language constructs. By combining it with `async`, one can asynchronously start an activity on another rank whose termination may be awaited using the `finish` construct. This may be used to create large amounts of parallelism very concisely, as demonstrated in the code segment below:

```
finish for (p in Place.places()) async at (p) for ([i] in 0..7) async {
    f(data,p,i);
}
```

This statement essentially executes a function `f` using shared as well as distributed memory parallelism. The outer for loop iterates over all places, and asynchronously starts activities on all places in the computation. The inner loop then creates eight asynchronous activities that each execute `f` with the specified parameters. All activities register with the enclosing `finish` on the initiating place. The original computation only resumes once all other activities are terminated.

Place shifting is a very powerful concept, but there are some subtle pitfalls. Most notable is the implicit capture of objects. As an object-oriented language, most data is part of larger object structures. But in some cases, only a certain attribute within a larger structure is needed remotely. However, the X10 runtime is not able to statically determine this, and will always copy the entire object, as shown on the left side in the example below:

```
// m will be transferred
val m : Matrix = getMatrix();
at (here.next()) {
    performSomeWork(m.rows * m.columns);
}
```

```
val m : Matrix = getMatrix();
val rows = m.rows;
val columns = m.columns;
at (here.next()) performSomeWork(rows *
↪  columns);
```

This results in the entire matrix `m` including all matrix elements being transferred over to the other place, even though only information about the matrix's size are needed. While the behavior is still fairly obvious in this example, it is more difficult to see when attributes or methods of the current object are accessed. This leads to access of the corresponding `this` object, and therefore to its serialization and transfer. However, it is possible to avoid the overhead by creating local variables that hold the values that are supposed to be transferred, as demonstrated in the listing above, on the right.

Another potential pitfall may arise when it comes to the copy-semantics of place-shifting. Whenever a place shift takes place, a deep copy of accessed objects is created, even when the object resided at the destination place originally. This may lead to bugs, as demonstrated in the following case. The goal is always to perform a complex task asynchronously on another rank, and then to modify the local object with the result of the remote operation. The straightforward approach taken by the code in the following listing on the left side seems intuitively correct.

```
// Original Place is Place(0).
val m : Matrix = getMatrix();
at (Place(1)) async {
    val tmp = m + m;
    at (Place(0)) async {
        // Incorrect. Working on a
        // copy of a copy of m!
        m.setMatrix(tmp);
    }
}
```

```
// Original Place is Place(0).
val m : Matrix = getMatrix();
val mRef = GlobalRef[Matrix](m);
at (Place(1)) async {
    val tmp = m + m;
    at(mRef.home) async {
        mRef().setMatrix(tmp);
    }
}
```

However, as stated above, each place shift leads to a copy. When the program shifts to place 1, it copies m^{P0} to obtain $(m^{P0})^{P1}_{\text{copy}}$, and uses it to compute the temporary result tmp. The intention of the next place shift is to modify the original m^{P0} on place 0. Instead, a copy $(m^{P0})^{P1}_{\text{copy}}$ is created. The copy $((m^{P0})^{P1}_{\text{copy}})^{P0}_{copy}$ is then modified to hold the data of the copy of tmp. As soon as the place shift ends, both objects will be removed, as there are no more references to it. Therefore, the update is lost. The listing on the right solves this problem using global references. As in the other listing, the function copies m^{P0} to obtain $(m^{P0})^{P1}_{\text{copy}}$ and to use it in the computation. However, when the result is propagated back, a global reference m^{P0}_{Ref} is used. This reference is also copied, first to place 1, then back to place 0, yet, unlike before, this time only a reference is copied this time, and that copy of the other reference still points the original object. We encountered this type of bug frequently during the implementation of actor migration in ActorX10, as the migration requires an extremely precise tracking of different, potentially circular references to actor graph components on different Places.

6. Task-Based Parallelism

Aside from MPI and OpenMP, the prevalent style of parallelism in HPC today, as well as various incarnations of PGAS, there are a number of other notable approaches that follow a comparatively high-level approach to parallelism. Two of the more widely used frameworks are High Performance ParalleX (HPX) and Legion (with Regent). Both are representatives of task-based runtime systems. Like actors, tasks may be used to formally describe concurrent program execution. Formally specified program segments (tasks) and their inter-dependencies form a task graph. Tasks that do not depend on each other may be executed concurrently. However, unlike actors, each task only exists intermittently for a single execution, whereas actors typically[1] have an extended lifetime. For coarse-granular parallel parallelism, there are several runtime systems such as Legion (Bauer et al., 2012), Uintah (Meng, Humphrey, and Berzins, 2012), HPX (Kaiser, Brodowicz, and Sterling, 2009) or StarPU (Augonnet et al., 2011). Recently, this model has also been implemented in shared memory by OpenMP (see section 3.2). First, I will describe Legion and Regent, as they enable the implementation of parallel programs without reqiring the explicit formulation of parallelism. With Regent, only the dependencies of a task need to be described. The formulation of the task graph is performed as the programm is generated. My second example, HPX, implements task-based parallelism solely using modern C++, using futures and asynchronous program execution. This approach is similar to the one taken to asynchronous communication in UPC++. Furthermore, HPX is one of the frameworks we evaluated in our performance study of the SWE-PPM code (see chapter 15).

6.1. Legion and Regent

In the following, I will discuss Regent (Slaughter et al., 2015), a high-level task-based programming language as an example implementation of task parallelism for distributed HPC systems. The Regent language is implemented as a Domain-Specific Language (DSL) on top of Lua as an extension of Terra (Bauer, 2014; Slaughter et al., 2015). Terra is a low-level programming language that realizes a multi-stage programming model (DeVito et al., 2013). In essence, Lua is used as a meta-programming language to generate a Terra program. The Terra constructs are then compiled into machine code using LLVM. Regent extends Terra with additional constructs to express task parallelism. Compared to using C++, this approach allows the Regent/Terra compiler a greater insight into the structure of the computation and the data structures involved, which in turn enables more opportunities for optimizations and especially for vectorization. For the application developer, the usability is increased, as the entire program is expressed in Lua and its DSLs,

[1] Depending on the precise formulation of the model. For the FunState model discussed in this thesis, they typically exist for the entire duration of the computation.

compared to using the C Preprocessor, C++ template meta-programming, and the C++ language itself. The parallel constructs in Regent are mapped to equivalent constructs in the Legion runtime. Legion itself is a C++ framework that implements a task-based parallel programming model, but leaves more implementation details, such as the data layout used for a specific task's data, to the application developer. The code generated by Regent uses knowledge about the task graph in order to provide reasonably optimized generated Legion code (Slaughter et al., 2015).

Tasks are the fundamental building block in Regent. Each task specifies the data segments (or *Regions*) they operate on, as well as the type of operation (reading, writing, or reducing) that is being performed. Tasks are submitted sequentially, and Regent ensures that the parallel execution produces the same results as the sequential one. If it is provable, either statically by the compiler, or at execution time through the runtime system, that two tasks do not interfere with each other's execution, they may be scheduled concurrently. To determine the existence of dependencies, each task is annotated with the data that is accessed, and what kind of operations are performed by the data. For example, if two tasks access disjoint data segments, or they access the same segment, but only by reading it, they may be scheduled concurrently (Slaughter et al., 2015; Slaughter, 2017). A simple task to generate the n^{th} Fibunacci number in parallel may be written as

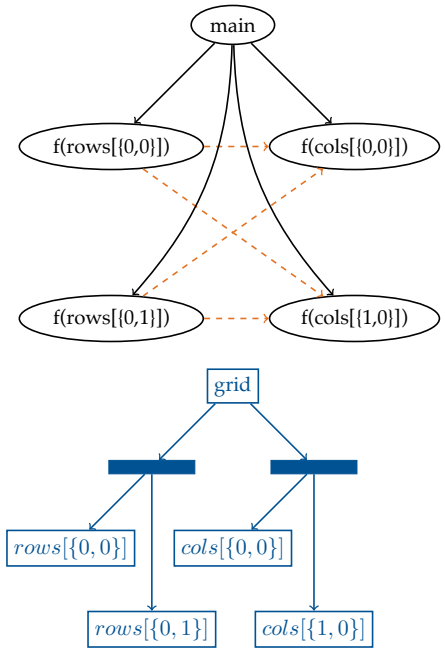
```
1 task fibunacci(n) do
2     if n < 1 then
3         return 1
4     else
5         var left = fibunacci(n-1)
6         var right = fibunacci(n-2)
7         return left + right
8     end
9 end
```

Each invocation of `fibunacci` is scheduled as a task. Two tasks may be executed concurrently, if they do not access shared data structures in a way that might produce race conditions. Regent requires tasks that access shared data to be annotated to specify what data will be accessed, and how. For the `fibunacci` example, there are no accesses to shared data, so no annotation is needed. The fibunacci task itself is executed sequentially, but tasks may spawn other tasks recursively. To avoid waiting for results from other tasks, the Regent compiler automatically encapsulates results from tasks started asynchronously into futures. Additionally, some of the basic operations of Regent, such as the "+" in the example, are overloaded to support futures directly, and therefore do not block themselves. Thus, the `fibunacci` task only blocks once the return statement is reached. This simple example already demonstrates the advantage of the approach taken with the Regent language: the implicit generations of future objects allows for the creation of arbitrary tasks graphs while still reading like sequential code.

For large-scale parallel applications, it is necessary to spawn large number of tasks very efficiently. Regent supports this using the *index launch* construct. It allows for the large-scale launch of tasks in parallel. If tasks are launched in a loop, and there are no dependencies between the iterations, the compiler will automatically launch all tasks in the loop at once. In order to make large-scale parallel computations on a singular dataset feasible, it has to be possible for multiple tasks that work on the set concurrently without introducing data dependencies. Regent and Legion support

```
1  task f(r : region(...))
2      where reads writes(r)
3  do
4      -- Implementation
5      -- omitted
6  end
7
8  -- Main Simulation:
9  var N = 8
10 var B = 2
11 var I = 0..N x 0..N
12 var R = 0..1 x 0..B
13 var C = 0..B x 0..1
14 var grid = region(I, ...)
15 var rows = partition(equal, grid, R)
16 var cols = partition(equal, grid, C)
17 for i = 0, B do f(rows[{0, i}]) end
18 for j = 0, B do f(cols[{i, 0}]) end
```

(a) Region Task with Partitions (Slaughter, 2017)



(b) Resulting Task Graph and Partitioning Scheme (Slaughter, 2017)

Figure 6.1.: Regent Data Partitioning example. In Figure 6.1a, a $N \times N$ region is subdivided into partitions of B rows and B columns. In Figure 6.1b, the dashed orange arrows denote sibling relationships, and black arrows denote parent-child relationships. Blue rectangles depict the relationship of region and its partitions. (Slaughter, 2017)

this though the concept of *regions* and *partitions*. Regions are array abstractions of arbitrary shape (e.g. regular with multiple dimensions, or unstructured). They are allocated lazily, may be moved around and replicated if necessary. Using partitions, it is possible to subdivide regions into arbitrary sub-regions to distribute a larger region onto multiple tasks. An example for such a partitioning is shown in Figure 6.1.

In the code a task is given that performs some work involving writes to a region passed as parameter. Then a region of size $N \times N$ is created, as well as two partitions. The first, *rows*, slices the region in a row-wise fashion in two sub-regions, while the other one slices in a column-wise fashion. Finally, tasks are started that operate on parts of the partitions. Figure 6.1b depicts the resulting task graph and the resulting partitioning scheme. The sub-regions are partially overlapping, and therefore, dependencies between the four child-tasks are introduced. In the figure, these are denoted through sibling-relationships. A task graph annotated thus may be executed on a distributed system. Based on the data-dependencies, the Regent compiler will move the data to the compute resources executing tasks as necessary (Slaughter, 2017).

Regent has been used successfully to parallelize applications on CPUs (Slaughter et al., 2015) as well as GPUs (Lee et al., 2019). In Lee et al. (2019), the authors use automatically derived partitions to place tasks on up to 512 GPU nodes of the Piz Daint cluster. The automatically generated partitions were performing comparably to hand-tuned partitions in simple applications, and also

in more complex ones, once some hints were provided. In addition to Regent, the task-based model of Legion has also been ported to Python (Slaughter and Aiken, 2019) and used to scale sequential NumPy code to a GPU cluster (Bauer and Garland, 2019).

6.2. HPX

The goal of the HPX runtime system (Kaiser, Brodowicz, and Sterling, 2009; Heller et al., 2017) is to provide a modern HPC runtime system based on the Parallex model. Its scope includes functionality for shared-memory and distributed parallelism, and is, essentially, to be "an open source C++ standard library for parallelism and concurrency" (Heller et al., 2017). Similarly to the PGAS environments described previously, the resources available to the computation are subdivided into separate shared-memory domains, the so-called *localities*. Localities are connected through the Active Global Address Space (AGAS), an abstraction layer over the underlying message-based communication layer (such as MPI or *libfabric*[2]). AGAS-aware objects (*Components*) are assigned a global ID. This enables objects to be migrated away from the locality they were initially created on, and hence makes it possible to access objects, independently of their physical location.

Within each locality, there is a thread manager, that maps the concurrent work onto the available compute resources. The runtime supports multiple scheduling policies, and allows the application developer to define custom ones. Currently, the default option is to have separate task queues for each CPU core, and to enable task stealing (Heller et al., 2017). HPX provides higher-level abstractions to specify work that may be performed in parallel. Like in Legion, it is possible to specify tasks and their dependencies. Using asynchronous execution (`hpx::async`), futures and the ability to specify dependencies in between, as in UPC++, one may create a task graph that is then scheduled onto the available resources of the locality. Furthermore, the library implements parallel versions of frequently used algorithms. Once more, parallelism is created by dividing the work into smaller tasks. Finally, it is also possible to create active messages, i.e. functions that are invoked on a different rank, in a similar manner as remote activities in X10. These active messages are scheduled on the receiving locality like any other task. As in X10, this is a one-sided operation that does not require user intervention.

Another notable feature of HPX is support for heterogeneous environments (Daiß et al., 2019). Daiß et al. use HPX in conjunction with a novel CUDA backend to simulate the merger of two stars in a binary solar system. The CUDA backend utilizes asynchronous streams to tie the GPUs into the task scheduler of HPX. Using this method, the fast-multiple-method-based application, Octo-Tiger, is able to scale up to 5400 nodes of the Piz Daint cluster. On the GPU nodes, they manage to reach 21% of the node's peak performance. In a scaling test spanning the entire cluster, they managed to reach a parallel efficiency of 68%.

[2] `https://ofiwg.github.io/libfabric/`

7. Actor-Based Parallel Programming

The actor model was originally proposed in the 1970s as a conceptual model for computation in the field of artificial intelligence, and later formalized as a model of computation. It is used as the base for Erlang, the main programming language for the implementation of industrial telecommunication operations, as well as in general-purpose programming languages and libraries. In this chapter, I will introduce some of them, and explain similarities and differences between them and the FunState actor model used in this thesis.

7.1. The Actor Formalism

The actor model has originally been proposed as a tool to express parallelism in the field of artificial intelligence (Hewitt, Bishop, and Steiger, 1973). The authors describe how actors might be used as a conceptual framework to express the interplay of different components within a larger system. In this framework, actors are used as an abstraction: the concrete implementation and internal structure of the actor is irrelevant, only the interactions with other actors matters. Hewitt, Bishop, and Steiger (1973) provide the definition: *" Intuitively, an ACTOR is an active agent which plays a role on cue according to a script. We use the ACTOR metaphor to emphasize the inseparability of control and data flow in our model. "* This refers to the basic definition of actors as *active objects*. An actor has an internal state and a behavior that is governed by the internal state and received messages. The arrival of messages also serves as the invocation of an actor. The actor model introduced by Hewitt, Bishop, and Steiger was formalized into a full computational model comparable to the λ-Calculus by Agha (1985). He defines actors as:

> *Actors are computational agents which map each incoming communication to a 3-tuple consisting of:*
>
> 1. *a finite set of communications sent to other actors;*
> 2. *a new behavior (which will govern the response to the next communication processed); and,*
> 3. *a finite set of new actors created.*
>
> *(Agha (1985))*

These tuple components outline key components of the actor model. First, actors make information flow explicit. There is no indirect communication, such as thorough shared data structures, between actors. Instead, actors send explicit messages to other actors. They may only send

messages to actors that are actually known to them, that is, actors that they know from previous communication steps, actors whose addresses were contained in the messages, or actors they themselves created in the course of the activation. Second, the behavior of actors is driven by messages. When a message is received, the actor may perform actions that result in the generation of messages sent to other actors, and a modification of its internal state, which may change how future messages are handled. To that end, each actor contains a *mail address* with an attached *mail queue* that stores incoming messages linearly by time of arrival, and an *actor machine* that describes the behavior of the actor for a given message. When an actor machine X_n processes the n^{th} element of the mail queue, it creates actor machine X_{n+1}, which will then process element $n + 1$, and so on. The different actor machines are independent, which means that X_{n+1} may already by processed before X_n has concluded without influencing each other. Third, according to the model of Agha, actors may create other actors when needed. This makes the conceptual model self-contained, i.e. it is possible to express arbitrary computations using the formalism. (Agha, 1985; Agha and Hewitt, 1988).

The actor model used for the actor libraries in this thesis retains the key concepts of explicit information flow as well as the message-based activation. As with the classic model, the state of the actors is only mutated in response to messages from the outside. However, a somewhat more static approach to actors is utilized here. In the FunState model, actors are created at the beginning of the computation. Communication is performed not through mailboxes, but through communication channels that are made explicit in the code. The channels are unilateral connections for messages of specific types. This allows for the static analysis of the actor graph at design time.

7.2. The Erlang Programming Language

The actor model is widely used today in communication systems by the *Erlang*[1] language. The language was originally developed at Ericsson Telecom AB for telephony systems. These systems were massively concurrent (tens of thousands of calls happening at the same time) and had soft real-time constraints for the individual actions. Typically, a single instance of the software would be distributed onto several servers. In order to provide uninterrupted service, the program's execution should never be halted, not even to perform software updates, to install security fixes, or to recover from software failures (Armstrong, 2007).

These requirements and continued development of the Erlang language resulted in a concurrency-oriented functional programming language. The main ideas of concurrency-oriented languages are: *" 1. Systems are built from proesses. 2. Processes share nothing. 3. Processes interact by asynchronous message passing. 4. Processes are isolated " (Armstrong, 2007).* These principles describe an actor-based model similar to the one formalized in Agha (1985), with processes taking the role of the actors. The actor-based style of programming reduces the amount of communication between different software components, as there are no shared data structures, locks or semaphores. In Erlang, this is extended with resilience characteristics. Following the environment above, it is possible for actors to fail without compromising the overall system (Hebert, 2013). For this, the language

[1] Homepage: `https://www.erlang.org`

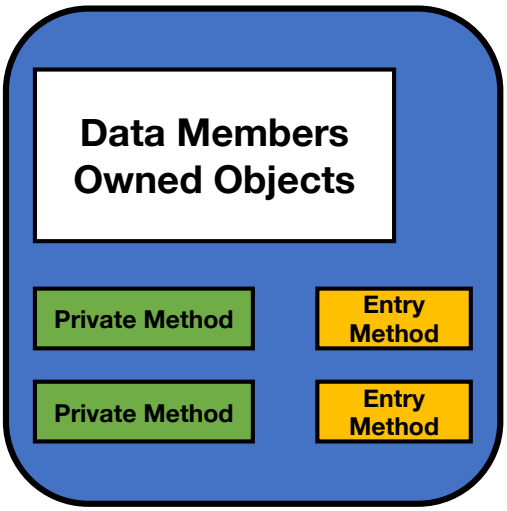
includes functionality for processes to react to failure of other processes they depend on. For process groups with strong dependencies, failure mitigation may be performed on the entire group by creating *links* between the individual processes. When a process that is linked to another dies, it will send a special message *(signal)* that cannot be processed through the normal message-handling to its counterpart, causing the other process to die as well. Links are bidirectional, and may be used to designate substructures of the actor graph that always exist together. This makes it easier to keep the actor graph in a consistent state, as failures will be propagated quickly instead of causing actors to wait for their counterparts that either do not exist anymore, or are themselves waiting for data. Once the failure has been contained, it is usually necessary to rebuild the graph. This may be done using *traps*. If a process within a linked sub-graph fails, and the failure signal is propagated to a process with traps enabled, that process is able to receive and handle the message, and to restart the computation when appropriate. For processes with weak dependencies, monitors may be used. These allow processes to track another processes' status without creation of a strong dependency. Whenever a monitored process fails, the monitoring process will be sent a message (Hebert, 2013).

The actor model used in Erlang is more closely related to the one proposed by Agha than to the one used in this work. Furthermore, the focus of Erlang is different. The environment Erlang was designed for never placed emphasis on large-scale numerical computations, and therefore the effort expended on supporting and optimizing them was smaller than in other languages, such as X10 and C++ (Hebert, 2013). However, Erlang's language features pertaining to resilience may also be very interesting in the field of high performance computing. Fault tolerance and resilience will be of increasing importance in the face of growing number of compute nodes involved in a computation, as the mean time between failures is inversely proportional to the number of nodes involved in the computation (Naksinehaboon et al., 2008). A failure-aware actor model may be an interesting way to contain failures and enable mitigation without forcing a complete restart of the computation. The actor libraries introduced in this thesis currently do not support this. However, the investigation of such a fault-tolerant actor model for HPC may be an interesting venue for future work.

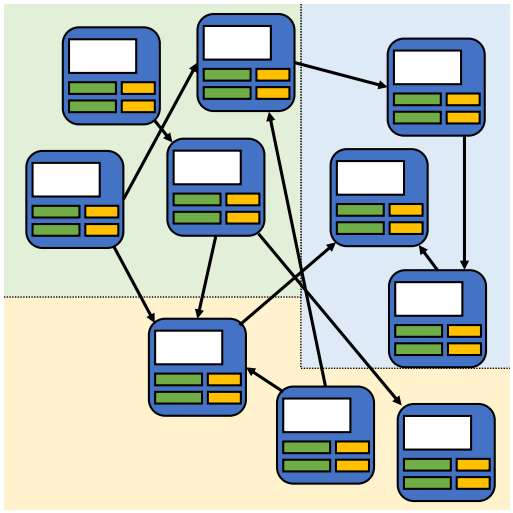
7.3. The Charm++ Distributed Runtime System

A well-established parallel runtime system with actor-like characteristics in HPC is *Charm++* (Kalé and Krishnan, 1993; Kalé and Zheng, 2016). In Charm++, computations are subdivided into separate objects *(chares)*. Each chare may uniquely own an arbitrary amount of "normal" C++ objects. For the programmer, each chare forms a separate processing entity, and its actual location on the distributed computer is hidden from the application developer. Chares may interact with each other using asynchronous procedure calls to specially designated chare member functions, the *entry methods* (Kalé and Zheng, 2016).

This model allows the Charm++ runtime system to move chares around as necessary in order to improve load balancing. The runtime system is also responsible for the scheduling of chares based on the incoming method invocations. This adds a layer of abstraction between the application developer and the underlying hardware. For the application developer this means that they do



(a) Single Chare. (Adapted from Kalé and Zheng (2016))



(b) Multiple chares on a distributed cluster. (Adapted from Kalé and Zheng (2016))

Figure 7.1.: Charm++ Object Model. (Figures adapted from Kalé and Zheng (2016))

not need to take care of load balancing and overlapping of computation and communication, as long as there is enough work available. To this end, Charm++ relies on the application developer to over-decompose the problem, which in turn allows the runtime system to balance the computational load and to overlap communication and computation. These basic features of Charm++ are similar to the actor model as proposed by Agha and Hewitt, but there are some notable differences: First, there is no mail list, instead, asynchronous method invocations are used. Second, Charm++ also supports collective operations on chares, and there is the possibility to create chare arrays, with multiple chares in an index space (e.g. one-dimensional array, or sparsely populated many-dimensional array). One may not communicate with chare arrays directly, but has to use proxy-stand-in (Kalé and Zheng, 2016). The computational model employed by Charm++ is more closely related to the one of Agha than to FunState. While in Charm++, every chare can talk to any other, in the FunState actor model, the communication between actors uses explicit communication channels. Furthermore, all interactions in the FunState model are on an actor-to-actor level; there are no collective operations. We implemented support for Charm++ in SWE in an effort to evaluate different modern runtime systems. The implementation is described further in chapter 15.

7.4. Actor Libraries in General Purpose Programming Languages

The actor libraries introduced in this thesis are not the only full actor libraries for general-purpose programming languages. A notable example is the *Akka* library[2] for the Scala programming language (Nordwall et al., 2011). The library features an actor model similar to Agha's formal-

[2] `https://www.akka.io`

ism, and has been added to the Scala standard library in more recent versions. However, high performance computing was never a focus of the library.

The computational model employed by *C++ Actor Framework (CAF)*, an actor framework written in C++, also remains close to Agha's formalism with the use of a central mailbox per actor instead of clearly structured communication paths using typed ports and channels (Charousset et al., 2013; Charousset, Hiesgen, and Schmidt, 2016). However, similar to Actor-UPC++, it is built on top of the widely used C++ programming language. Furthermore, it is possible to execute it in a distributed environment using socket-based communication over TCP, or using OpenMPI, making this library a candidate for distributed computing applications.

Finally, the X10 task library proposed in (Roloff, Hannig, and Teich, 2014) is a precursor to the ActorX10 library. The contribution outlines a prototypic, work-in-progress implementation of the actor model in X10. ActorX10 (see chapter 10) improved on this by formalizing the model, by structuring the communication using typed ports and channels, and by supporting migration of actors between places.

8. Invasive Computing

As discussed in chapter 2, the increasing amount of concurrency in modern CPU architectures necessitates novel concepts for their effective and efficient use. Transregional Collaborative Research Center Invasive Computing (InvasIC), a project funded by the German Research Foundation (DFG), explores the interplay of hardware and software in this emerging space. The project is motivated by the increase in processor complexity. Its original prediction was that there would be a thousand processing units per chip by the early 2020 (Teich et al., 2011). As of today, this prediction has been surpassed for some types of accelerators: GPUs[1] have thousands of small compute units working in parallel, and many-core CPUs with more than fifty cores within a node have become a[2] common[3] sight[4] in modern HPC architectures. In conventional systems these cores are managed and allocated by the operating system, which allocates the available computing time to the currently active applications. The user is able to change the resource mapping through directives to the operating system, but the applications have little awareness of the resources they are running on, and less choice on when and on which resource they get to perform computations. In HPC systems, this is somewhat mitigated through static resource allocations, but the influence of the operating system (which may execute other background applications on the allocated node nevertheless) may cause some performance fluctuations (Petrini, Kerbyson, and Pakin, 2003). Without user intervention[5], the threads of an application may be shuffled freely between cores, which leads to a less predictable and often lower performance (Treibig, Hager, and Wellein, 2010).

The assumption of the InvasIC project is that to program these systems most effectively, resource-awareness is paramount. Applications for future many-core systems need to be conscious of the type and amount of resources they are running on, and of theeir respective capabilities. Furthermore, the project stipulates that for predictable performance to occur, the system's resources must not be shared. However, unlike in HPC, resource allocations are not static, but initiated dynamically by the application in concert with the runtime system. Resource allocations are distributed based on the requirements of the application and the overall system characteristics as observed by the runtime system. This requires the interplay of all layers in the system stack, from the application, over the middleware, i.e. the operating system and the programming environment, down to the hardware. (Teich et al., 2011)

---

[1] The NVidia Tesla A100 has 6912 so-called CUDA cores (NVIDIA Corporation, 2020)

[2] Compute nodes on NERSC's Cori use Intel Xeon Phi processors with 68 cores, and 272 hardware threads (NERSC, 2020)

[3] Lrz's SuperMUC-NG uses Intel Xeon Processors with 2×24 cores with a total of 96 hardware threads per node (LRZ, 2020b)

[4] HLRS's Hawk uses a dual socket AMD Epyc 2 configuration with a total of 128 cores, or 256 hardware threads (HLRS, 2020)

[5] By pinning threads to CPU cores

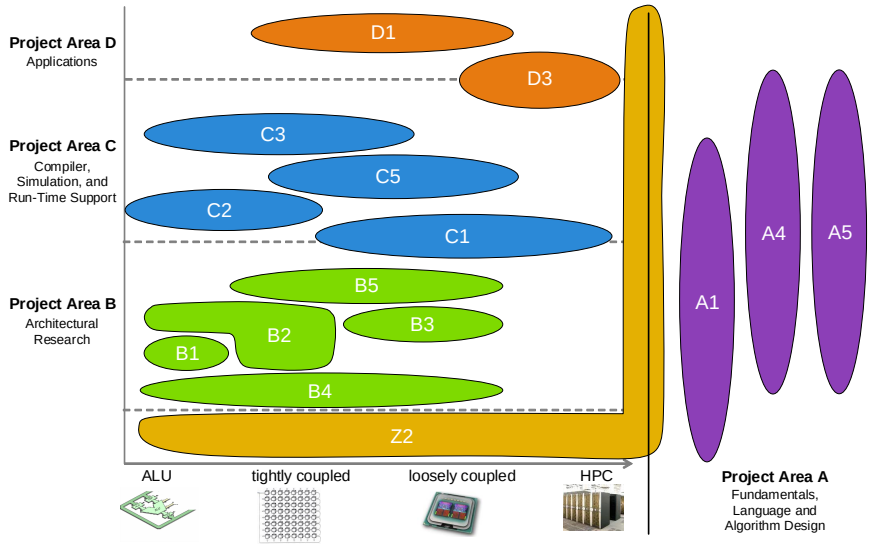


Figure 8.1.: Invasive Computing project overview. The project is distributed between four project areas. (Adapted from InvasIC (2010))

The scope of the project is mirrored in its organizational structure. Work is distributed amongst 13 sub-projects split up between three universities, the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), the Karlsruhe Insitute for Technology (KIT) and Technical University of Munich (TUM). Figure 8.1 depicts the organizational structure with a subdivision into four project areas, (A) *Fundamentals*, (B) *Hardware*, (C) *Middleware*, and (D) *Applications*. The sub-projects, each with its own focus points, deal with parallelism on all hardware levels (represented in the figure on the X axis), from parallelism within the Core up to distributed compute clusters. Furthermore, the projects provide the custom components of the invasive technology stack: from the invasive hardware platform, over the operating system OctoPOS, the invasive Runtime Support System (*i*RTSS), the Invasive X10 Language (InvadeX10) and its compiler `x10i`, and ActorX10 up to the applications. The project was funded in three phases between 2010 and 2022. In the first phase, the main focus was to establish the basics of invasive computing: the invasive hardware architecture, the runtime support, and the invasive programming model. In the second phase, the main focus was on predictable execution: how can program execution be guaranteed to be within certain quality numbers such as throughput, latency or power usage? To achieve this, techniques such as design space explorations, specific hardware accelerators and the actor model are used. Finally, in the third phase, the main focus is on requirement enforcement. Given established theoretical bounds on the quality numbers of an application execution, how can these requirements be enforced to be able to provide more tight upper and lower bounds on the predicted quality numbers? The main instrument here will be runtime requirement enforcement, using a combination of design-time and of run-time measures to ensure adherence to strict bounds in the quality numbers.

The remainder of this chapter contains an overview of the invasive programming model and its various incarnations in section 8.1, a summary of the invasive hardware components in section 8.2, and finally the invasive design flow in section 8.3. A large part of this thesis discusses work

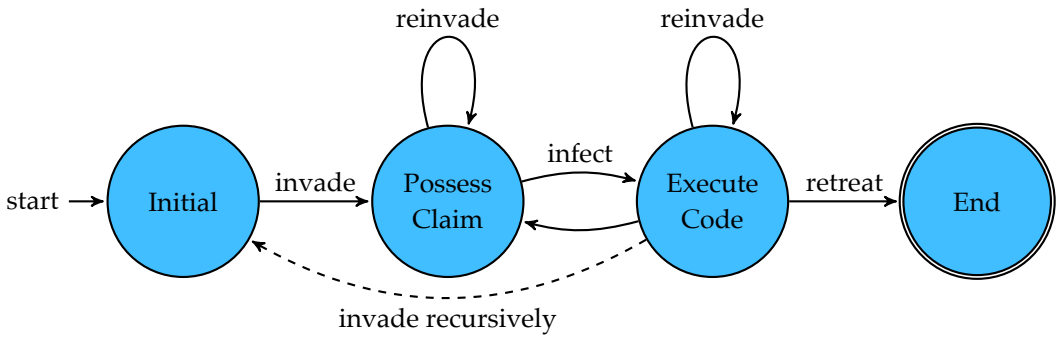


Figure 8.2.: State diagram for an invasive application. Applications initially invade a claim. They infect the claim to perform a computation. Claims may be changed using the reinvade operation. Once the claim is not needed anymore, it may be relinquished by retreating from it.

related to InvasIC, specifically, the actor model introduced in Part II, the actor-based tsunami simulation SWE-X10 (in chapter 16) and its optimization for the invasive compute stack discussed in chapter 9.

8.1. The Invasive Programming Model

The primary goal of invasive computing, resource-awareness, is realized through explicit allocation of resources, as depicted in Figure 8.2. An application starts on a single core of the system. When more resources are needed, the application needs to request them explicitly. The allocation of resources is realized by an explicit allocation, or *invasion* of the resources by the application at runtime. At the start of a new phase of the computation, a set of resources is *invaded*. This set is *constrained* by the arguments passed to the invade invocation. Based on these constraints, the *i*RTSS will select the *claim* (set of resources) most appropriate for the specified constraints and the overall system state (e.g. available resources, quality numbers of the system) and return it to the application. The application may now *infect* the claim with the piece of code to be executed, referred to as invade-let (*i*-let) in the invasive terminology. Upon completion, it may either execute further *i*-lets on the claim, or *reinvade* the claim to obtain a different one that better matches future requirements. Once the claim is no longer needed, the application may relinquish it by *retreating* from it.

Figure 8.3 depicts the claim constraint hierarchy (claim constraints are depicted in blue). There are two different types of constraints available, one for conventional applications, and the other for actor-based applications. *Actor constraints* (depicted in green) are used in conjunction with actor-based applications. They allow for the allocation of one of several pre-characterized actor-resource mappings delivered as part of the application. Actor constraints will be discussed in more detail in section 8.3. For conventional invasive applications that use the *invade* operation, it is possible to specify different *claim constraints*, simple ones such as the number of processing units or the availability of certain hardware characteristics, or more complex ones such as performance characteristics of the algorithm or tile sharing. It is also possible to compose different constraints

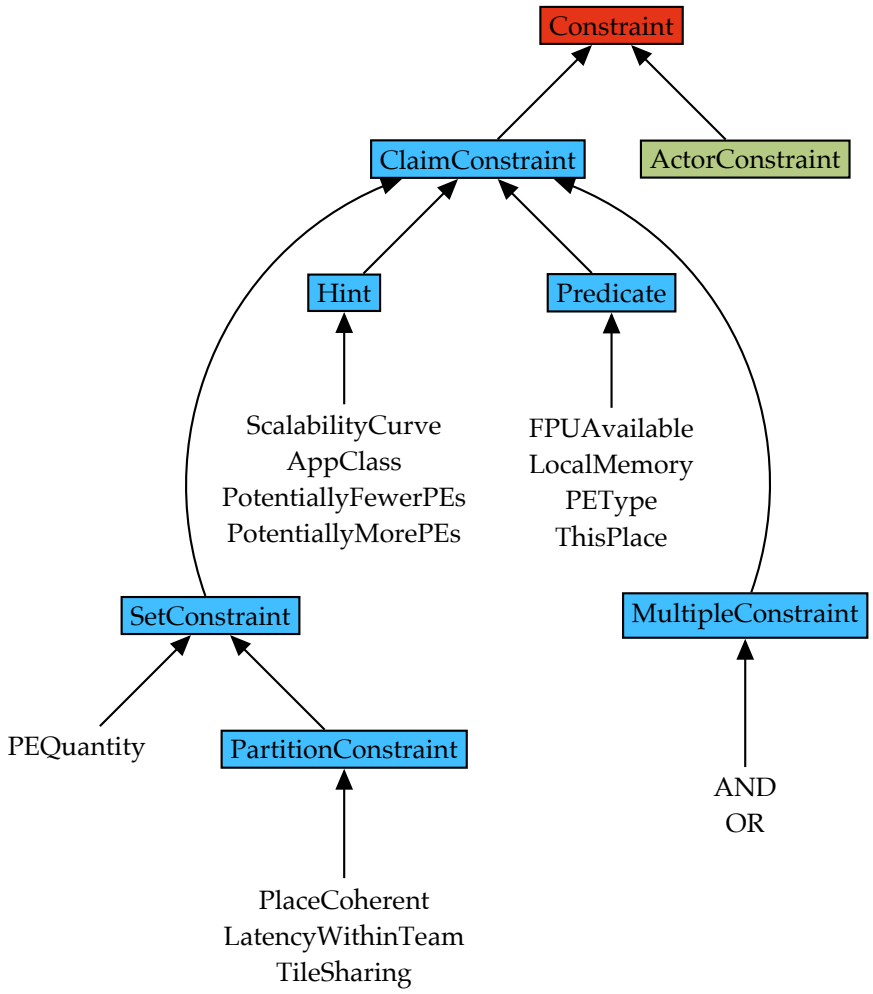


Figure 8.3.: Overview of the available constraints for the invasion of resources. There are two types of constraints: claim constraints may be made to a concrete claim that allows for arbitrary code execution within it. Actor constraints are limited to claims used for actor graph executions. Claim constraints are adapted from Bungartz et al. (2013)

```
1  public static def complexWork(d:Data);
2
3  public static def main(Rail[String]) {
4      val data = loadData();
5      try {
6          val constr = new PEQuantity(8,16) && TileSharing.ONLY_WITHIN_APPLICATION;
7          val claim = Claim.invade(constr);
8          claim.infect((IncarnationId) => {
9              finish for (p in Place.currentWorld()) async {
10                 val dataPiece = data.forPlace(p);
11                 at (p) {
12                     for (i in 0 .. (claim.getNumPEs(here) - 1)) async {
13                         complexWork(dataPiece.chunk(i));
14                     }
15                 }
16             }
17         });
18         claim.retreat();
19     } catch (ex:NotEnoughResources) {
20         // Insufficient Resources
21         // Perform Work sequentially.
22         complexWork(data);
23     }
24 }
```

Figure 8.4.: InvadeX10 code example. Depending on the success of the invasion, a function is either executed on the allocated claim, or sequentially.

logically, using the `AND` or the `OR` constraint. The claim constraints chosen by the application are then used by the *i*RTSS to select a suitable claim. If the runtime is successful, the claim is returned, otherwise the runtime signals failure.

8.1.1. Invasive Computing in X10

The main programming language in InvasIC is InvadeX10, an extension of X10 (see chapter 5) with the invasive language constructs and support for a dynamically changing amount of places. The language is compiled using the invasive X10 compiler `x10i`, developed as part of the invasive technology stack. It provides the aforementioned vocabulary to implement invasive applications. In contrast to the conventional X10 provided by IBM, Places are not static, but assigned as part of the invasion. To this end, the InvadeX10 runtime provides the ties to *i*RTSS and OctoPOS, which provide the resources to the application. Figure 8.4 depicts a simple invasive program written using InvadeX10. First, the application allocates a claim using `Claim.invade(constr)` with a set of constraints, namely four to eight processing units and tile-exclusivity. If the invasion does not succeed, an exception is thrown, and the work is performed sequentially on the initial claim. If

invasion succeeds, the claim is infected with the *i*-let performing the application. The *i*-let performs its work asynchronously on all cores and Places it is assigned using the conventional X10 language constructs. Finally, once work is finished, the claim is relinquished using `c.retreat()`.

In Bungartz et al. (2013), this approach was used to implement an invasive multigrid application. The geometric multigrid method they use has to apply a stencil operation on different resolution levels (from fine to coarse, and then fine again) in each time step. This leads to a greatly varying resource demand within a single time step. For the coarser grid levels, the amount of work required is only a very small fraction of the amount required for the finer resolution levels (for each level of coarsening, the number of cells is divided by four). The authors used invasive techniques to change the resource set for the different levels in the cycle. Before each coarsening and after each refinement, the claim is reinvaded with a fitting constraint set, and the data re-distributed, if necessary. Compared to the non-invasive variant, this approach allows for a more efficient use of the available resources, as currently idle resources may be used by other applications. This is illustrated using a demo scenario involving the invasive multigrid application and two instances of an invasive numerical integration application. When static resource allocation is used, the applications are restricted to the initial resource allocation even when one of the applications is already terminated and does not require the resources anymore. Conversely, with invasive computing, these resources are dynamically redistributed throughout the execution of the application, and once an application is terminated, its resources are available for utilization by the remaining applications. Thus, the invasive approach has the potential to drastically reduce system idle time, as in Zwinkau (2018) and Teich, Kleinöder, and Mattauch (2016).

The basic X10 API described above was later expanded by Buchwald, Mohr, and Zwinkau (2015) with the creation of externally malleable applications using an additional `Malleable` constraint during invasion. By providing a resize handler that specifies how to rebalance work based lists of newly added and removed Processing Elements (PEs) as a parameter for the constraint, the runtime is able to mold the resource set of invasive applications based on the overall system situation. The main advantage for the runtime is that a change in the resource set may happen without relying on the application to call reinvade. As a potential user for the method, the invasive task queue was implemented. Applications using the job queue for parallelization benefit from the asynchronous malleability without any additional implementation effort (Teich, Kleinöder, and Mattauch, 2015).

8.1.2. System Programming using the OctoPOS API

The high-level constructs for invasive computing are mirrored on the lower levels of the invasive compute stack. As with the X10 API, the OctoPOS API exposes operations to allocate resources and to execute code on them. Resources may be allocated using different *invade* functions, for example `int invade_simple(claim_t, uint32_t quantity)` for allocations of claims on the same tile as the allocating code. For more complex allocations, calls are asynchronous, and return a future, so that the initiating *i*-let does not need to idle until the operation is completed. On the level of the OctoPOS API, *i*-lets are the primary way to express parallelism. An OctoPOS *i*-let consists of a function (with a `void *` as its parameter) to be executed, and the pointer that is to be passed as a parameter. It is brought to execution using one of the different *infect* operations. For example,

`infect_simple(simple_ilet *)` executes a single *i*-let on the current claim context. Alternatively, an array of *i*-lets may be executed on a given claim using `infect(claim_t, simple_ilet *, uint32_t)`. Finally, the API offers local and remote *retreat* operations. Similarly, there are also specialized *invade* and *infect* calls available for inter-tile communication using Direct Memory Access (DMA) operations. (*OctoPOS API Description* 2020)

An example code for use of the API, adapted from the *OctoPOS Application Development GIT Repository* (2020) is given in Figure 8.5. The code performs a simple *i*-let execution on a remote tile. In the beginning, a single core on a remote tile is allocated using `proxy_invade(1, &future, 3)`. The operation returns a future object that represents the status of the operation. By invoking `invade_future_force(&future)`, the main *i*-let is suspended until the claim is ready. Now, the remote *i*-let may be created with the function to be executed (`remoteILetFunc`) and the signal that will be used to await the remote *i*-let's completion. Invocation of `proxy_infect(pc, &iLet, 1)` executes the *i*-let on the remote claim. At the end of the remote *i*-let's execution, it signals the main *i*-let by running another *i*-let on the claim that was used to invoke the remote *i*-let. Once the main *i*-let receives that signal, it retreats from the remote claim, and terminates.

The InvadeX10 runtime uses the lower-level API for the X10 parallel language features (Mohr et al., 2015). While the high-level invade operations described in subsection 8.1.1 are implemented using an agent system that negotiates the resource use on behalf of the X10 application, the original X10 language functionality for parallelism (e.g. `at`, `async` and `finish`) is implemented by mapping to the appropriate calls of the OctoPOS API. Parallelism within a node is mapped to `simple_ilets`, and finish statements are implemented using the signals introduced in Figure 8.5. Operations involving a place shift are the most difficult to implement. As *i*-lets only have a data payload of two pointers, and the X10 specification requires the entire object graph pertaining to every accessed object to be copied, data transfer is split into separate parts. First, as with the regular X10 runtime, the object graph is serialized into a buffer. Then, a remote *i*-let is started on the place that the computation is to be executed at to allocate the memory, and to manage the transfer. The OctoPOS API offers support for DMA and appropriate completion notifications (using *i*-lets on the source and destination place). the *i*-let on the source side deallocates the buffer, and the *i*-let on the destination side reconstructs the object graph. Once the transfer is finished, the remote *i*-let may execute the code within the `at` statement. Depending on the nature of the place shift, another data transfer may be necessary, which is performed similarly. Finally, after the execution of the place shift is complete, a final *i*-let is started on the remote place to wait for all the dependent activities created during the place shift to terminate (global termination). In later work, data transfer has been simplified using Pegasus (Mohr and Tradowsky, 2017) for direct copying of the object graphs without serialization, and the use of hardware accelerated queues (Rheindt, Maier, et al., 2019) for data transfer (Rheindt, Fried, et al., 2019).

8.1.3. The Invasive MPI Runtime

The two aforementioned ways to write invasive software are a useful approach for the implementation of new applications. However, this is not always feasible for large legacy HPC applications that are actively used and extended (both by HPC developers and domain experts). Most of the current HPC applications are written using a combination of MPI and OpenMP. Porting such

```
#include <octopos.h>
#include <stdio.h>
#include <stdlib.h>

static void remoteILetFunc(void *arg);
static void doSignal(void *arg);

extern "C" void main_ilet(claim_t claim) {
  invade_future future;
  if (proxy_invade(1, &future, 3) != 0) {
    fputs("proxy_invade failed. Tile 1 unavailable.\n", stderr);
    abort();
  }

  proxy_claim_t pc = invade_future_force(&future);
  if (pc == 0) {
    fputs("invade_future_force failed, insufficient resources.\n", stderr);
    abort();
  }

  simple_signal signal;
  simple_signal_init(&signal, 1);

  simple_ilet iLet;
  simple_ilet_init(&iLet, remoteILetFunc, &signal);

  proxy_infect(pc, &iLet, 1);
  simple_signal_wait(&signal);

  proxy_retreat_sync(pc);
  shutdown(0);
}

static void remoteILetFunc(void *arg) {
  printf("Hello from tile %u!\n", get_tile_id());
  simple_ilet reply;
  simple_ilet_init(&reply, doSignal, arg);
  dispatch_claim_send_reply(&reply);
}

static void doSignal(void *arg) {
  simple_signal_signal(static_cast<simple_signal *>(arg));
}
```

Figure 8.5.: OctoPOS API code example. First, a claim with one core on another tile (Tile 3) is allocated asynchronously. A signal and an *i*-let are allocated, and there remote claim is infected with the *i*-let. The original claim waits for the remote *i*-let's completion, and then terminates. Code example adapted from the *OctoPOS Application Development GIT Repository* (2020).

applications to InvadeX10 may prove to be an unreasonably expensive endeavor, as this would essentially amount to a complete rewrite. The use of the OctoPOS API does not require a complete re-implementation, but parts pertaining to parallelization and inter-rank communication still need to be rewritten. Nevertheless, HPC applications may still profit from invasive techniques. Most notably, the static nature of job allocations used on current HPC systems might be changed through invasive computing. The currently available resource managers for distributed systems allocate resources statically for each application. Once the application starts its execution, its resources remain as they were when the computation started. However, many applications do not absolutely require the specified number of nodes, but would also work efficiently with less (or more) resources. A more efficient use of the available resources may be attained if another application with fixed demands is able to claim some currently running elastic application's resources along with the remaining idle resources. Furthermore, elastic applications could claim further resources if there are no other potential users currently available. Other applications have varying resource demands. For example, a tsunami application may require a different set of resources initially when the wave propagates across the ocean where a more coarse resolution suffices, compared to when it reaches the coast and a very fine resolution is needed. To enable more flexible and more efficient use of computing resources, InvasIC proposed *i*MPI.

*i*MPI (Comprés et al., 2016; Mo-Hellenbrand, 2019) extends the standard feature set of MPI with a set of functions to accommodate dynamic resource managers. Applications using *i*MPI may periodically give the resource manager of the system they are running on the opportunity to change the number of MPI ranks involved in the computations. Applications that support *i*MPI invoke `MPI_Init_adapt(int *, char***, int*)` (instead of `MPI_init(int *, char ***)` for conventional MPI applications) at the beginning. In addition to the two parameters for the program arguments, the operation also writes the status of the application into an integer variable. This allows the new application to see whether it has been started as part of the initial set of rank (status: `MPI_ADAPT_STATUS_NEW`) or during an adaptation of the resource set (status: `MPI_ADAPT_STATUS_JOINING`). Depending on the state, the application may then perform the usual initialization routine, or skip ahead to participate in the adaptation. During the computation, the ranks of the application periodically collectively query the resource manager through `MPI_Probe_adapt(int*, int*, MPI_Info*)`. The first parameter is written with the information on whether an adaptation should be performed or not, the second one contains the status for the invoking rank, either that it is joining (`MPI_ADAPT_STATUS_JOINING`), staying (`MPI_ADAPT_STATUS_STAYING`) or leaving (`MPI_ADAPT_STATUS_LEAVING`) the computation. If the probe indicates the need for a resource adaptation, it may be initiated by all participating ranks by invoking `MPI_Comm_adapt_begin(MPI_Comm*, MPI_Comm*, int*, int*, int*)`. The first parameter contains the inter-communicator that allows for communication between all involved ranks (joining, staying and leaving). The second parameter contains the communicator that will be used by the ranks once the adaptation is completed. Finally, the last three parameters contain the number of staying, leaving and joining ranks. Now, depending on the nature of the adaptation, the application has to move all essential simulation data away from any rank leaving the computation, provide the necessary information [6] to the newly joining ranks, and finally redistribute the data amongst the remaining ranks. Once the adaptation is completed, it may be finalized using `MPI_Comm_adapt_commit()`.

[6] e.g. computational phase, iteration number, current time step, ...

The framework was successfully integrated into the framework sam(oa)^2 (Meister, 2016) by Mo-Hellenbrand et al. (2017) and Mo-Hellenbrand (2019), and used to simulate the Tohoku tsunami. The framework uses a dynamic and adaptive mesh refinement to accurately simulate tsunami propagation. Typically, segments of the domain that do not contain any activity are subdivided into coarser cells, and coastal areas and the wave front are divided into fine cells. The distribution of cells changes as the tsunami propagates across the domain, and typically the overall amount of cells in the simulation increases. When this behavior is made known to the resource manager, this may lead to significant savings in compute resources. In tests performed on the SuperMUC cluster of Leibniz Supercomputing Centre (LRZ), it was possible to save about half of the CPU time using an allocation of resources following the resource demands of the application instead of a static resource allocation. The overhead for the adaptivity was determined to be around $\sim 5\%$ in a separate experiment.

8.2. Overview of the Invasive Hardware

An important claim made since the project's inception is that for the invasive idea to work, software support alone is not sufficient, but new hardware is needed (Teich, 2008; Teich et al., 2011). To satisfy this requirement, and to provide a platform for invasive applications, the invasive hardware architecture was introduced. On the highest level, it is implemented as a MPSoC architecture with multi-core tiles of different configurations, connected using a Network-on-Chip (NoC). A similarly tiled architecture is employed today by current Intel many-core processors such as the Intel Xeon Phi Knights Landing (Sodani, 2015).

A tiled architecture allows for easy reconfiguration and customization for different scenarios. The project's current hardware prototype employs 4×4 tiles, but the architecture may be scaled up to thousands of cores if desired, and given the availability of the hardware resources. A sample configuration, serving to illustrate the most important characteristics of the invasive hardware architecture, and the custom components developed for the project, is depicted in Figure 8.6. The architecture consists of sixteen tiles of different configurations.

Some tiles are conventional compute tiles featuring four general-purpose LEON 3[7] CPU cores. Other tiles contain so-called Invasive Cores (*i*-Cores) in addition to conventional CPU cores. These are general-purpose CPU cores with an attached reconfigurable fabric to enable application-delivered hardware acceleration. For processing of loop-based programs, there are tightly coupled processor array (TCPA) tiles. These provide a large number of simple, but tightly connected processing elements that may be used to compute multiple iterations of a loop-based program in parallel (Hannig et al., 2014). Finally, there is a tile specialized for communication with off-chip components, the memory and I/O tile. It provides access to the main memory of the system (containing heap-allocated memory objects), and to different I/O devices such as Ethernet or image acquisition. The tiles are connected through an invasive Network-on-Chip (*i*NoC) consisting of one NoC router per tile, and horizontal and vertical links in between them.

[7] Open Source CPU core provided by Cobham Gaisler (Cobham Gaisler AB, 2016)

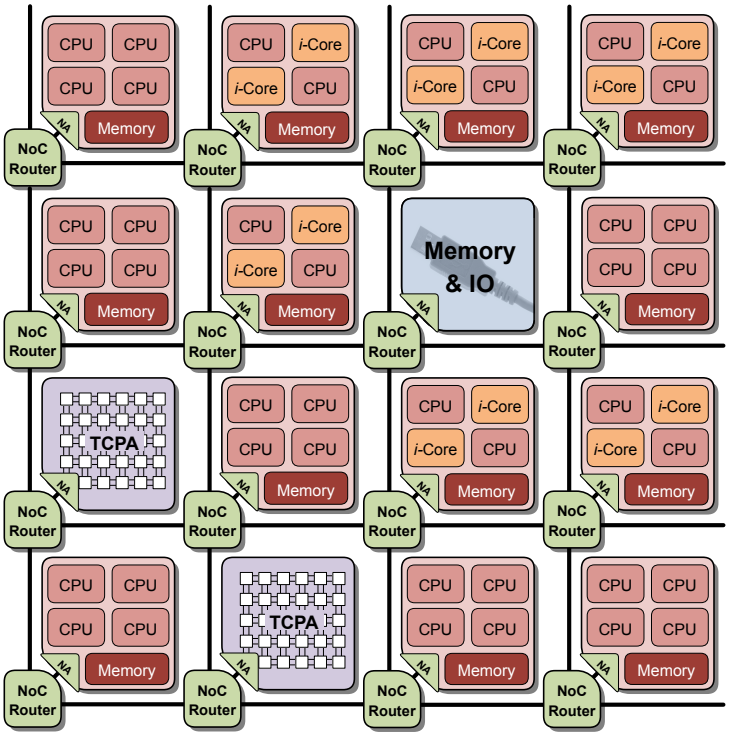


Figure 8.6.: Invasive Computing Hardware Architecture Overview. The current default hardware architecture contains 4×4 tiles with four different configurations. There are tiles with four conventional CPU cores, tiles containing *i*-Cores, tiles containing TCPAs, and tiles containing memory and I/O interfaces. (Adapted from InvasIC (2010))

The compute tiles (see Figure 8.7 for a tile with four conventional LEON 3 cores) are a multi-core design customized from a design by Cobham Gaisler AB (2016). The tiles contain a number of CPU cores that are connected to the tile's central bus through Level 1 instruction and data caches. Attached on the bus are also a tile-local memory (TLM), that typically contains the application binaries and the local variables for the *i*-lets executed on the compute tiles. Access to non-local memory, such as the TLM of other tiles, or the off-chip DDR-RAM is provided through the *i*NoC, and cached through a Level 2 Cache for faster repeated accesses. Finally, the C*i*C is a hardware scheduler that is responsible for scheduling the *i*-lets, both locally created and incoming remote ones, based on a round-robin scheduler, and optionally the operating conditions (e.g. temperature) of the tile (Henkel et al., 2012).

The *i*NoC (Heißwolf, König, and Becker, 2012; Heisswolf et al., 2014) connects the different compute tiles. Each tile has a network adapter (NA) that connects it to a NoC router within the network. In accordance with the general principles of invasive computing, it also allows for resource-aware applications. Message transmission between two routers is split up into a number of round-robin-scheduled, periodically repeating time slots. Using that, it is possible to define a *virtual channel* that occupies one or more of these slots. Applications may choose either a state-less *best-effort* routing of packages, or a *guaranteed-service* connection with predictable

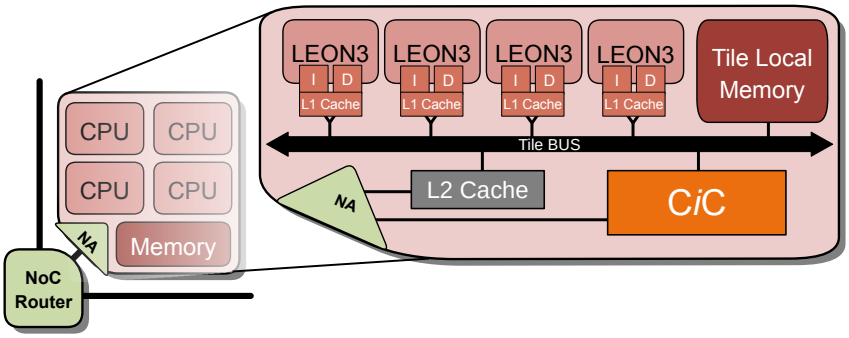


Figure 8.7.: Overview of a compute tile within the invasive architecture. The tile has four LEON 3 cores, each with a L1 Data and Instruction Cache. The various components are connected using a bus. The connection to the system's main memory is provided through the NoC, and cached through the L2 Cache. The core *i*-let controller (C*i*C) is a hardware scheduler that is responsible for scheduling local and remote *i*-lets. (Adapted from InvasIC (2010))

execution quality. For applications with best-effort routing, the message is sent directly and without establishing a formal connection first. For guaranteed-service connections, a fixed route using virtual channels between routers has to be established[8]. An example for this is depicted in Figure 8.8. The connection is established router-by-router by sending a header package through the network. When such a package is encountered by a router, it will try to allocate a virtual channel that satisfies the requirements specified in the package. If the allocation succeeds for all connections between the source and the destination NA, the thus established virtual channels is reserved exclusively for that application until it is relinquished. The use of virtual channels allows for predictable transmission of information between tiles. Applications with a guaranteed-service connection have a predictable latency and throughput (which may be computed based on the number of hops, the link bandwidth, and the fraction of the connection being used), and are not subject to network congestion.

In addition to the conventional LEON 3 cores, there are also two types of specialized invasive computing units, the *i*-Core and the TCPA. The former uses application-specific micro-instructions for the acceleration of compute-intensive parts of the application, while the latter uses numerous customizable and freely connectable PEs.

The TCPA tile (Hannig et al., 2014; Teich, Tanase, and Hannig, 2014), depicted in Figure 8.9, may be used to accelerate specific loop-intensive workloads. Unlike conventional multi-core tiles, it only contains one LEON 3 core—responsible for the coordination with the rest of the system—and the TCPA itself. The TCPA utilizes relatively simple PEs. Early versions were based on a Very Long Instruction Word (VLIW) Instruction Set Architecture (ISA) which allowed for instruction-level parallelism without requiring sophisticated pipelining. More recently, the addition of floating point arithmetic required the replacement of VLIW by orthogonal instruction processing (Brand et al., 2017). Only PEs at the borders of the array may access data from the I/O buffers. The inner PEs access data, such as results previously computed by neighbors, or intermittently buffered

[8] Or, one might say, invaded.

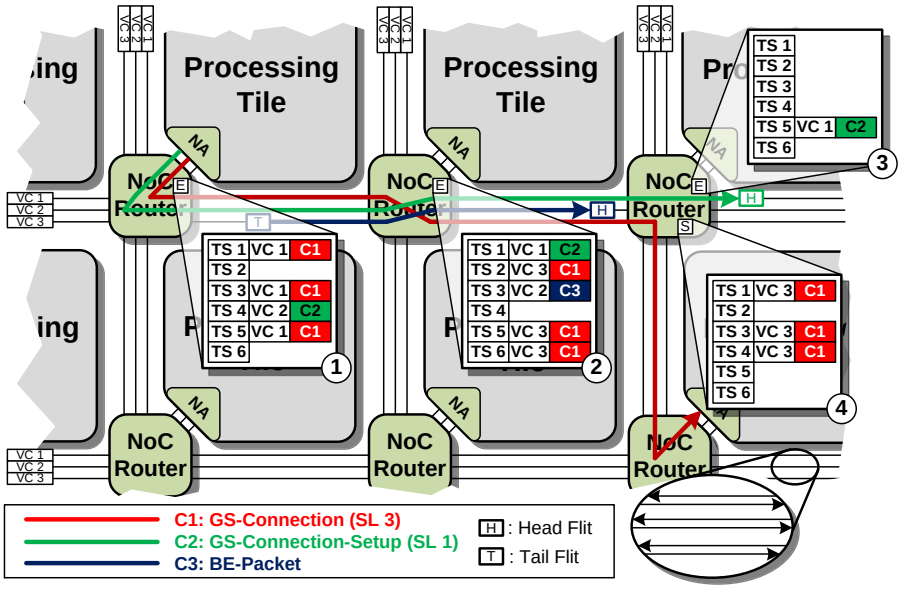


Figure 8.8.: *i*NoC Example. There are three connections currently being processed. C3 (depicted in blue) is a best-effort connection that is currently being routed through router port ②. C1 (red) is an already established guaranteed-service connection that occupies virtual channels with three time slots each in router ports ①, ② and ④. C2 (green) is a guaranteed-service connection in the process of being allocated. Its header package just established a connection through router port ④. (Heißwolf, König, and Becker, 2012).

data provided their neighbors. This enables a very low latency data exchange: newly available data is accessible by neighboring PEs in the next clock cycle. The PEs themselves possess a very basic set of functional units: addition, multiplication, bit-shifting or logical operations. This allows for a combination of design-time and execution-time configurability; PEs may mix and match functional units according to the requirements of the intended applications. In contrast to prior approaches, the TCPAs used in the InvasIC hardware architecture are not limited to a static configuration, but allow for a dynamic reconfiguration and reconnection of the PEs in the system based on application requirements. Applications may invade PEs of the TCPA in different ways, based on strategies described by (Lari et al., 2011). For image applications and other two-dimensional loops, a two-dimensional area of the TCPA may be invaded starting from a core at the boundary of the array. For other applications, a one-dimensional pipeline may be invaded following either a straight line or a "meandering[9]" approach. The configuration of the TCPA is performed using the *TCPA Utility* that allows for the configuration of a TCPA and the generation of the corresponding hardware description data using a graphical user interface. The TCPA may be programmed either using a low-level assembly approach, with the PAULA DSL which allows for the formulation of the parallel computation without explicitly specifying the loops (Hannig et al., 2014), or using symbolic loop transformation (Teich, Tanase, and Hannig, 2014; Witterauf, Hannig, and Teich, 2019).

The *i*-Core (Bauer, 2009; Damschen, 2019; Damschen et al., 2020) tile (see Figure 8.10) has a similar structure to the conventional compute tile, but one or more of the LEON 3 cores is replaced by an *i*-Core. *i*-Core combines the execution capabilities of a conventional LEON 3 core with the ability to execute application-specific acceleration. Acceleration is achieved through *i*-Core Custom Instructions (CIs), i.e. Micro-Programs (μPrograms) that may utilize accelerators imprinted onto small embedded field programmable gate arrays (FPGAs) within the chip. Applications are able to provide both μPrograms and accelerators. The basic principle of CIs is similar to a μProgram called by a CPU instruction. These are typically used to implement a more complex multi-step operation such as a square-root or division that cannot be implemented in hardware in a single step (Intel Corporation, 2016). Similarly, the CI consists of a number of operations and their dependencies, resulting in a data flow graph. Operations are either loads or stores from the TLM or cache, or compute operations implemented in one of the application-provided accelerators. The latter forwards the specified data to the accelerator loaded in one of the *i*-Core's reconfigurable containers, and brings it to execution. Depending on the application, it is possible to have multiple different accelerators, or a number of instances of the same accelerator loaded at the same time. A notable feature of the *i*-Core is the 128Bit-wide connection to the TLM, which enables the *i*-Core to forgo accessing the tile bus for loading data.

The concept of combining reconfigurable fabric with a general-purpose CPU is in itself not a new idea. Intel combined a Xeon SP processor with an Arria 10 FPGA on a single board (Huffstetler, 2018), and Xilinx offers the Zync 7000 series that combines an FPGA with an ARM CPU. However, the *i*-Core couples the reconfigurable fabric very tightly to the general-purpose components of the CPU. Unlike in the solutions above, the *i*-Core may access registers directly, and using previously loaded CIs is as simple as calling the corresponding processor instruction.

[9] i.e. going in a zig-zag fashion away from the initially invaded PE

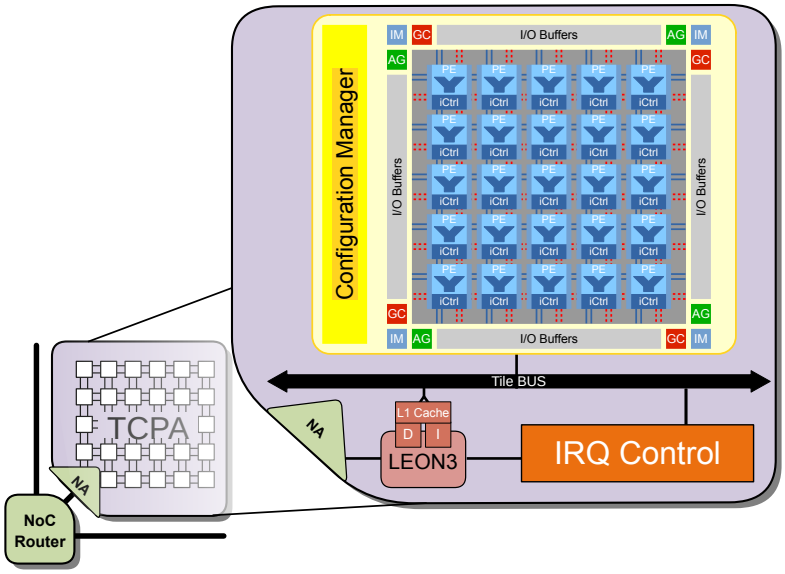


Figure 8.9.: Invasive TCPA accelerator tile. Aside for the LEON 3 core for coordination tasks and an interrupt controller, the tile contains the TCPA itself. The TCPA consists of a number of interconnected PEs, each with its own invasion controller (*iCtrl*), the configuration manager, invasion managers (IM), address generation units (AG), global controller (GC), and I/O buffers. (Adapted from Hannig et al. (2014))

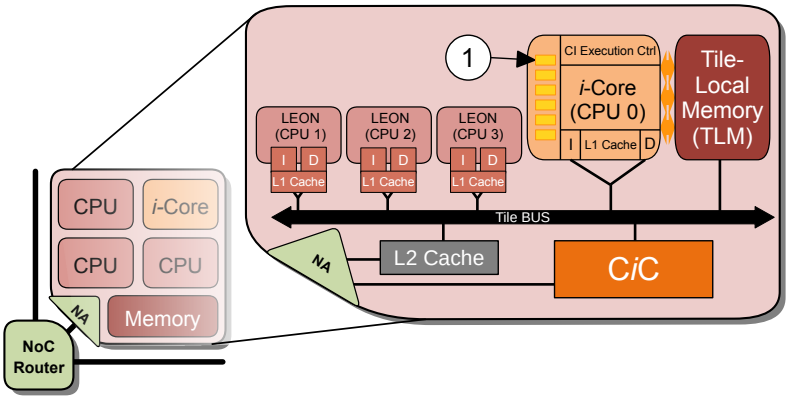


Figure 8.10.: *i*-Core accelerator tile containing three LEON 3 cores and one *i*-Core. In addition to the functionality of a general-purpose core, it also contains reconfigurable containers (marked with ①) and the controller to program them, and to bring them to execution. The orange arrows denote the low-latency and high-bandwidth connection to the TLM. (Adapted from Pöppl et al. (2018))

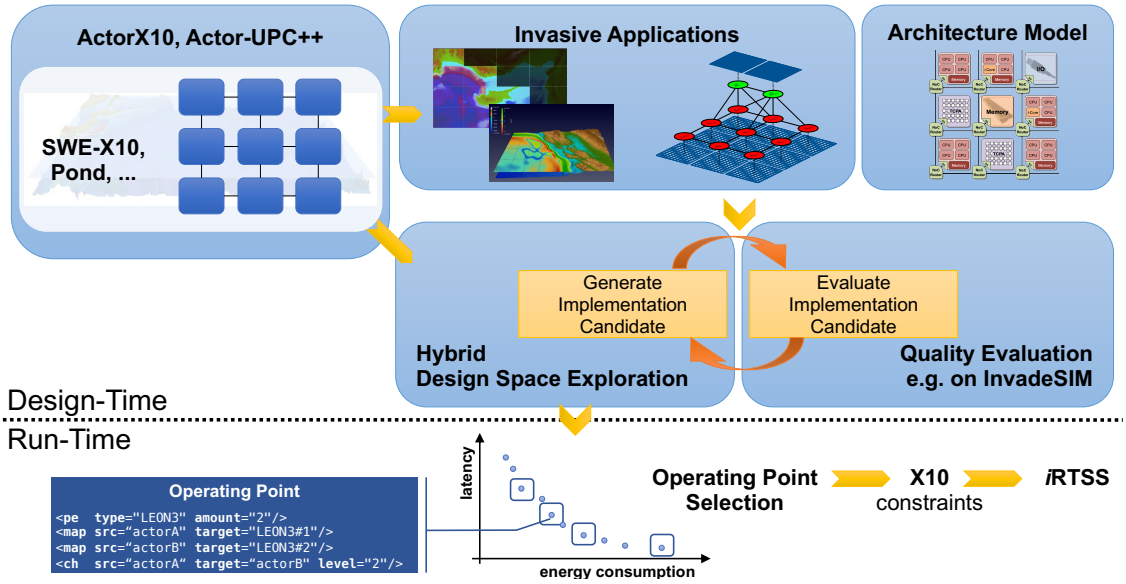


Figure 8.11.: Design flow for an Invasive application. (Adapted from InvasIC (2010))

8.3. Invasive Design Flow

For multi-core environments such as the one described in this chapter, maintaining predictable execution is difficult without sacrificing flexibility. While it is especially important in conjunction with hard real time problems[10], it is also an interesting technique for other fields where predictable execution quality is desirable. For example, in the field of HPC, one might be interested in obtaining application configurations that adhere to specific energy bounds. On shared resources such as classical computing systems, this is difficult to achieve, as there is always the possibility of interference from other applications. However, both with invasive computing and in classical HPC applications, resource allocations are exclusive. When an application has an allocation, no other application is permitted to compute (or use the communication resources, if allocated exclusively). At the same time, the system may not be available to the application in its entirety. Other applications may be using the system at the same time, and they typically also have their own sets of requirements. The application should therefore be flexible enough in its setup to allow for different resource configurations that still allow it to reach the desired quality numbers.

In the invasive computing project, we (Wildermann et al., 2016) propose a design flow for actor-based invasive applications. The technique is based on *Hybrid Application Mapping (HAM)*. An overview is depicted in Figure 8.11. It is based on a separation of activities to be performed at *design-time* during the creation of the application, and those performed at *run-time*, i.e. when the application is brought to execution on the target system in production. At design time, the application is analyzed and annotated, and at run-time, the *i*RTSS selects a specific configuration for execution. The invasive flow assumes an actor-based application, as it requires a clear separation of the individual components of the computation, and clearly defined communication

[10]This class of problems require the application to adhere to strict deadlines for specific events. If a deadline is not met, a catastrophic failure of the system may occur. (Kopetz, 2011)

```
public def build() {
    // Declare actors
    val v1 = new SourceActor("v1");
    val v2 = new Filter1_1Actor("v2");
    val v3 = new Filter2Actor("v3");
    // ...

    // Declare actor graph with requirements
    @REQUIRE("ag", new Latency(0, 110, "ms", "hard"))     // Performance Req.
    @REQUIRE("ag", new Throughput(20, 40, "fps", "soft")) // Performance Req.
    @REQUIRE("ag", new Power(1, 2, "W", "soft"))          // Power Req.
    @REQUIRE("ag", new Confidentiality(50))               // Security Req.
    val ag = new ActorGraph("ag");

    // Add actors and connect them
    ag.addActor(v1);
    ag.addActor(v2);
    // ...
    ag.connectPorts(v1.outPort1, v2.inPort);
    ag.connectPorts(v1.outPort2, v3.inPort);
    ag.connectPorts(v2.out, v4.inPort);
    // ...

    // This statement is replaced by the design flow
    ag.start();
}
```

Figure 8.12.: ActorX10 actor graph with requirement annotations. The statement in the last line is later replaced by the set of operating points, the result of the Design Space Exploration (DSE). (Wildermann et al., 2016)

paths. Such applications exhibit a clearly outlined communication path, and are able to make their structure known to other applications, in the form of the actor graph. For example, Figure 16.4 in section 16.2 depicts an actor graph for my invasive proxy application. Furthermore, it is possible to annotate the application or individual actors with requirements such as throughput, latency or power consumption. An example for an annotated actor graph is depicted in Figure 8.12. This actor graph, together with a specification of the target architecture, is taken as an input for a DSE. The DSE then performs an iterative optimization scheme where it generates different mappings of the actors and channels to the resources specified in the architecture description (Lukasiewycz et al., 2011). Evaluation is done in batches of different configurations. Typically, the DSE does not generate concrete configurations, but only abstract *constraint graphs* that specify the types of resources an actor is mapped to, and the maximum number of hops that may be taken to get to its communication partners. In this way, the search space is reduced significantly, as evaluating configurations that only differ in their location on the system, but are otherwise identical is avoided (Schwarzer et al., 2018). Each configuration is evaluated separately. Depend-

```
val cg = new ConstraintGraph();
val t0 = cg.addTaskCluster(2, Type.iCore);
val t1 = cg.addTaskCluster(3, Type.RISC);
val t2 = cg.addTaskCluster(1, Type.TCPA);
val m0 = cg.addMessageCluster(t1, t0, 3, 7);
val m1 = cg.addMessageCluster(t0, t1, 3, 4);
val m2 = cg.addMessageCluster(t1, t2, 2, 7);
val m3 = cg.addMessageCluster(t2, t1, 2, 2);

OperatingPoint op1 = new OperatingPoint(cg);

val q1 = new PowerConsumption(1.2, 2.0, "W");
val q2 = new PFH(0.0001, 0.000001);

op1.setQualityNumber(q1);
op1.setQualityNumber(q2);

operatingPoints.add(op1);
// Create and add other operating points...

val claim = Claim.invade(operatingPoints);
```

(a) Operating Point Creation (Wildermann et al., 2016)

```
// Bind actors onto claim
// according to selected
// operating point
if (claim.getSelection() == op1)
↪ {
  val r0 =
  ↪ claim.getResource(t0);
  val r1 =
  ↪ claim.getResource(t1);
  val r2 =
  ↪ claim.getResource(t2);
  ag.moveActor(v1, r1);
  ag.moveActor(v2, r0);
  ag.moveActor(v3, r0);
  ag.moveActor(v4, r2);
  ag.moveActor(v5, r1);
  ag.moveActor(v6, r1);
} else if (claim.getSelection()
↪  == op2) {
  // ...
}

ag.start();
```

(b) Claim Binding (Wildermann et al., 2016)

Figure 8.13.: Operating Point back-annotation into the invasive actor-based application. Usually performed through source-to-source compilation using the program and the set of operating points as input. (Wildermann et al., 2016)

ing on the configuration, this may be done using formal models (e.g. for worst-case execution time (WCET) analysis), using a simulator, or on the actual hardware. In all cases, *quality numbers* that describe the nonfunctional properties of the execution are collected and stored with the respective application configuration. In the course of the optimization, the optimizer maintains a set of *pareto-optimal* configurations. A pareto-optimal configuration is not equal to or worse regarding every quality number compared to the other pareto-optimal configurations. As the optimization progresses, the set of pareto-optimal configurations usually converges. The final set of pareto-optimal configurations, annotated with their quality number is the result of the DSE. Such an annotated configuration is also referred to as *operating point*. This set may contain a very large number of very similar operating points, and is therefore condensed into a smaller, more manageable set through operating point distillation (Pourmohseni, Glaß, and Teich, 2017).

The set is then embedded into the application for use in production through source-to-source compilation. For each operating point, the mapping onto the resources is embedded in the form of constraint graphs. A sample operating point for the code sample above is depicted in Figure 8.13a.

This constraint set is then handed over to *i*RTSS to invade an appropriate resource set. The run-time system then selects an appropriate operating point that fits the current conditions of the system, such as currently available resource or energy use. Once the operating point is selected, a suitable mapping onto the concrete resources is selected using a back-tracking algorithm (Weichslgartner et al., 2014). Finally, the claim is created, the actors may be moved to the indicated X10 places, and the computation started.

Setting the Stage

In the previous chapters, I gave an overview of both the environment of parallel computing in general, and the specific environment of my thesis project. In the next part of this thesis, I will formally introduce the FunState variant of the actor model. This incarnation follows the principles of the actor model as outlined in chapter 7, specifically the encapsulation of data and the message-driven behavior. However, unlike with the traditional actor models, the FunState model also formalizes the actor's behavior using a finite state machine and communication between actors through communication channels.

In the following, I will discuss two different actor libraries: ActorX10 and Actor-UPC++. ActorX10 was developed in the context of the invasive computing project, using X10. It serves an important role in the invasive design flow by making the application's structure explicit. This enables the generation of mappings for a given target architecture that are optimal regarding quality numbers such as throughput and latency. Actor-UPC++ was developed for use in large-scale distributed applications. The library is built on top of UPC++, and makes use of that framework's high performance asynchronous communication mechanisms.

In the third part of the thesis, I apply the actor model to the field of tsunami simulation. This part specifically will demonstrate both the feasibility and the benefits of using the actor model for block-structured HPC applications. I will demonstrate that the application developer can benefit from improved programmability through decoupling of the individual software components while retaining performance that is competitive, or even better than the conventional approach using MPI and OpenMP.

Part II.

The Actor Model

9. The FunState Actor Model

The actor model used in this thesis is based on the *FunState* model as defined in Roloff et al. (2016). It is based on the original version of Strehl et al. (2001), and was used in the collaboration to create ActorX10, and in Actor-UPC++. I will start by formally introducing *actors* and the structure connecting them, the *actor graph*. This general model forms the basis of the two actor libraries, and may be adapted for different models of computation as needed.

Definition 1 (Actor graph) *An actor graph is a directed graph $G_a = (A, C)$ containing a set of actors A and a set of channels $C \subseteq A.O \times A.I$ connecting actor output ports $A.O$ with actor input ports $A.I$. Each channel has a buffer size determined by $n : C \to \mathbb{N}^+$, and a possibly empty sequence of initial tokens denoted by $d : C \to D^*$ where D^* denotes the set of all possible finite sequences of tokens.*

Actors may only communicate externally using its ports. A port may only process tokens of a specific type. The model distinguishes two types of ports, *InPorts* as endpoint for receiving data, and *OutPorts* for sending data to other actors. Ports are connected through *channels*. Channels are buffers of finite size that may contain up to $n(c), c \in C$ tokens of type D. Communication between actors connected in this way occurs in First In First Out (FIFO) order, i.e. the ordering between messages in the same channel is always preserved. This is the natural mode of operation for many applications, and facilitates the use of widely used patterns in scientific computing, such as a halo layer exchange, as the application may rely on the fact that the data is always read in sequence. As indicated in Definition 1, the sole means of communication between actors is through channels. Implicit communication of any kind, for example through shared data structures or shared memory locations, is prohibited. An actor is therefore constrained to only consume data items (*tokens*) from channels connected to its input ports and to produce tokens on channels connected to its output ports. It is also possible to place initial tokens in FIFO channels, e.g., $d(c) = \langle 1, 2 \rangle$ to denote that two initial tokens with the values 1 and 2 are on the channel c.

I will use Cannon's algorithm for distributed and parallel matrix multiplication throughout this chapter as an example to illustrate the FunState actor formalism (Cannon, 1969). The algorithm performs a matrix-matrix multiplication in parallel on $p \times p$ processors, and is given in Algorithm 1 (cf. Bader (2019)). In the beginning, each processor is assigned three matrix blocks: $c_{i,j}$, which will eventually contain the result, and $a_{i,k}$ and $b_{k,j}$ (with $k \leftarrow i + j \bmod p$), which are blocks from the input matrices A and B. All matrix blocks contain $\frac{m}{p} \times \frac{m}{p}$ elements. Once the matrix data is distributed, a partial result is computed and added onto the result matrix. Then, each processor sends its data on to another processor. Blocks from matrix A are sent from rank $P_{i,j}$ to $P_{i,(j-1) \bmod p}$, i.e. to the processor "above", and blocks from matrix B are sent from $P_{i,j}$ to $P_{(i-1) \bmod p,j}$, or one rank "to the left". At the same time, the actor receives blocks from the ranks

Algorithm 1 Cannon's Algorithm

procedure MatMul(a,b,c) ▷ Perform $C = A \cdot B$ in parallel
  $P_{i,j} \leftarrow a_{i,j}, b_{i,j}, c_{i,j}$ with $1 \geq i, j \geq p$ ▷ on $p \times p$ processors
  $k \leftarrow i + j \bmod p$
  $a \leftarrow a_{i,k}, b \leftarrow b_{k,j}$
  for l in $1..p$ **do**
    $c_{i,j} \leftarrow c_{i,j} + a \cdot b$ ▷ Perform matrix multiplication on local blocks.
    Concurrently ▷ Send and receive new matrix blocks
      $a \ggg P_{i,(j-1) \bmod p} : a'$
      $b \ggg P_{(i-1) \bmod p,j} : b'$
      $a' \lll P_{i,(j+1) \bmod p} : a$
      $b' \lll P_{(i+1) \bmod p,j} : b$
    EndConcurrently
    Synchronize
    $b \leftarrow b', a \leftarrow a'$
  end for
end procedure

"below" and "to the right". Once communication has terminated on all ranks, computation of the next partial result may start. This is repeated until all partial matrices have been computed and the final result has been obtained.

The parallel execution of the matrix multiplication from matrices of size $m \times m$ using the Actor model may therefore be modelled as follows:

$$
\begin{aligned}
G_{\text{Cannon}} &= (A_{\text{Cannon}}, C_{\text{Cannon}}) \\
A_{\text{Cannon}} &= \left\{ P_{i,j} | i, j \in [1, p] \right\} \\
C_{\text{Cannon}} &= \left\{ \left(P_{a,b}.O_{\text{h}}, P_{c,d}.I_{\text{h}} \right) | a, b, c, d \in [1, p], c = a - 1 \bmod p, d = b \right\} \\
&\quad \cup \left\{ \left(P_{a,b}.O_{\text{v}}, P_{c,d}.I_{\text{v}} \right) | a, b, c, d \in [1, p], c = a, d = b - 1 \bmod p \right\}
\end{aligned}
\tag{9.1}
$$

Each processor in the original algorithm is replaced by an actor named $P_{i,j}$. For a computation with $p \times p$ actors, this leads to p^2 actors. In the algorithm above, each processor sends its matrix block a to the processor above, and each matrix block b to the processor to its left. The channel structure of C_{Cannon} reflects this communication flow. Each actor has two output ports $O_{P_{i,j}} = \{O_{\text{h}}, O_{\text{v}}\}$ and two input ports $I_{P_{i,j}} = \{I_{\text{h}}, I_{\text{v}}\}$, and channels connecting the vertical output port of an actor with its top neighbor, and its horizontal output port with its left neighbor. For actors at the boundary of the grid, the channels are wrapped around to the other side. This creates connection structure in the shape of a two-dimensional torus. The properties of the channels may be modelled as shown in Equation 9.2. The capacity n of each channel is set to two, and initially the matrix blocks pertaining to each actor's coordinates are placed in the channel connected to its input ports. A sample actor graph for $p = 4$ is displayed in Figure 9.1.

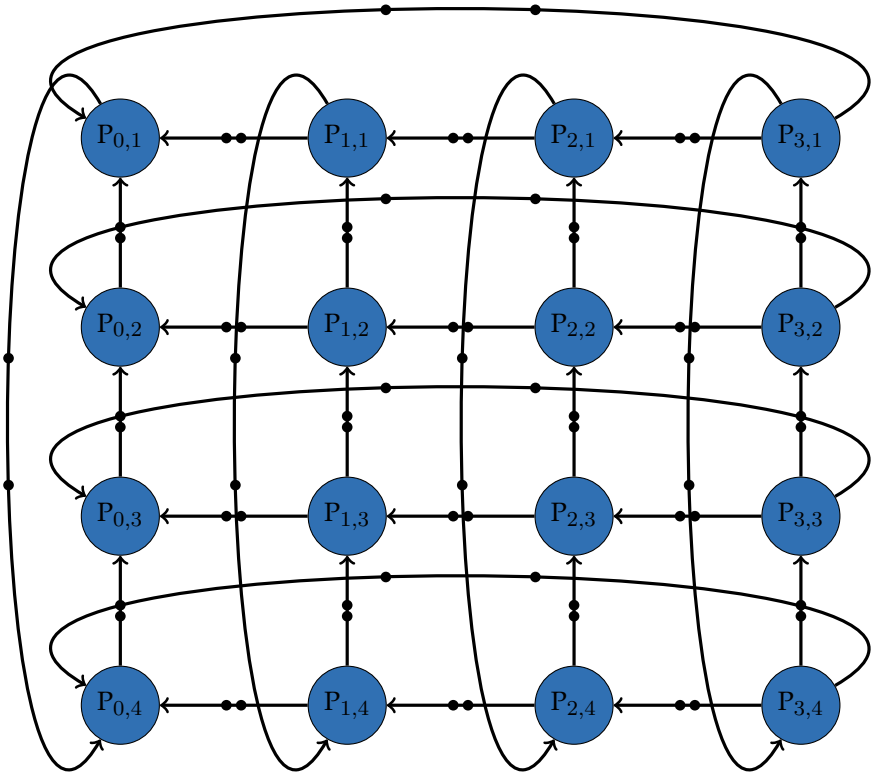


Figure 9.1.: Actor graph example. The actor graph displays an actor graph instance for the parallel execution of Cannon's algorithm.

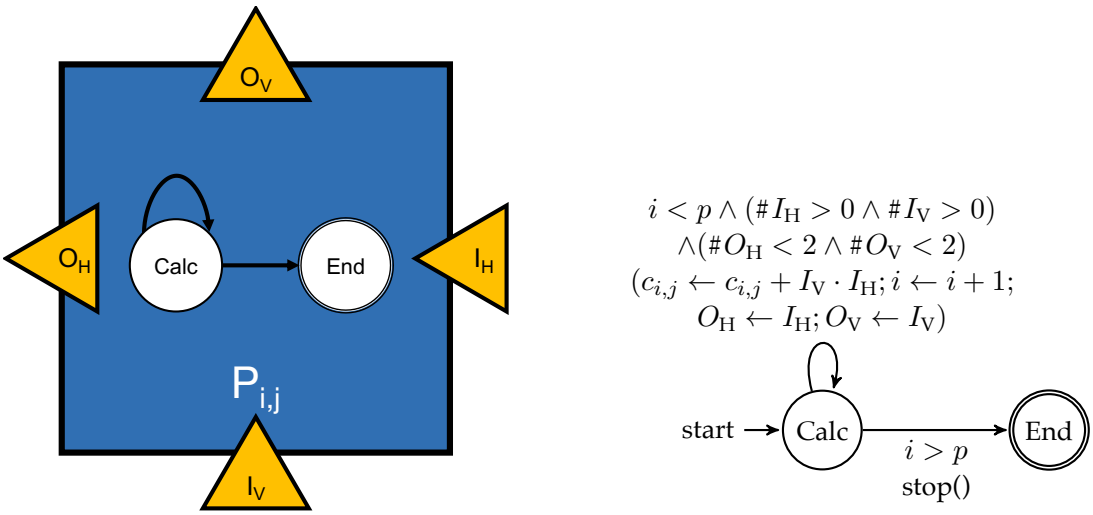
$$
\begin{aligned}
n_{\text{Cannon}} &: C \to \mathbb{N}^+ \\
n_{\text{Cannon}}(c) &= 2 \\
d_{\text{Cannon}} &: C \to \left(\mathbb{R}^{\frac{m}{p} \times \frac{m}{p}}\right)^* \\
d_{\text{Cannon}}((P_{i,j}.O_{\text{v}}, P_{i',j'}.I_{\text{v}}) &= a_{i',j'} \\
d_{\text{Cannon}}((P_{i,j}.O_{\text{h}}, P_{i',j'}.I_{\text{h}}) &= b_{i',j'}
\end{aligned}
\tag{9.2}
$$

The behavior of the actors is formally specified as well. The specification defines rules governing the actions of the actor, if and when it is activated to perform them, and on which data. In our model of the actor, we specify the behavior through an Finite State Machine (FSM). Execution of a state transition may depend on a specific configuration of tokens on the actor's ports, as well as the actor's internal state. The execution of a state transition is also referred to as *firing*. For each transition, we define the number of tokens consumed for each input port, the tokens produced for each output port, and the functions executed during the transition. In our incarnation of the model, we also require the separation of the functions performed in the transition from the communication and the state machine. Formally, we describe actors and their FSMs following Roloff et al. (2016).

Definition 2 (Actor) *An* actor *is a tuple $a = (I, O, F, R)$ containing actor ports partitioned into a set of actor input ports I and a set of actor output ports O, a set of functions F, and an FSM R called* firing finite state machine. *The functions encapsulated in an actor are partitioned into so-called* actions *$F_{Action} \subseteq F$ and* guards *$F_{Guard} \subseteq F$. Functions are activated during a so-called* firing transition *of the FSM R, which unambiguously formalizes the communication behavior of the actor (i.e., the number of tokens consumed and produced in each actor firing). Actions may produce results in the form of output tokens residing in the FIFO channels connected to the actor output ports. Using guards, more complex models of computation may be modeled. In particular, the activation of actors is based not only on the availability of a minimal number of tokens on the input ports, but also on their values. Guards return a Boolean value and may be assigned to each transition of the FSM of an actor.*

Definition 3 (Actor (Firing) Finite State Machine) *The* firing FSM *of an actor $a \in A$ is a tuple $R = (Q, q_0, T)$ containing a finite set of states Q, an initial state $q_0 \in Q$, and a finite set of transitions T. Moreover, a* transition *of an FSM is a tuple $t = (q, k, f, q') \in T$ containing the current state $q \in Q$, an activation pattern k, the respective action $f \in a.F_{Action}$, and the next firing state $q' \in Q$. The activation pattern k is a Boolean function which decides if transition t can be taken (*true*) or not (*false*) based on: (1) a minimum number of available tokens on the input ports $a.I$, (2) a set of guard functions $F' \subset F_{Guard}$, and (3) a minimum number of free places in the channels connected to respective output ports.*

Using these definitions, I model the Cannon actors $P_{i,j} \in A_{\text{Cannon}}$ as $P_{i,j} = (I, O, F_{\text{Cannon}}, R_{\text{Cannon}})$. The port structure follows the definitions above: $I = \{I_{\text{V}}, I_{\text{H}}\}$ and $O = \{O_{\text{V}}, O_{\text{H}}\}$. The set of Functions is modelled as the union of the set of guards F_{Guard}



(a) Cannon Actor $P_{i,j}$

(b) Cannon Actor FSM.

Figure 9.2.: Actor and FSM for actor-based modelling of Cannon's Algorithm. The figure on the left depicts the Cannon actor $P_{i,j}$. The yellow triangles denote its ports. The picture on the right depicts the actor's FSM. In each of the transitions, the upper half denotes the activation pattern, and the lower half the actions.

and the set of actions F_{Action}:

$$
\begin{aligned}
F_{\text{Cannon}} &= F_{\text{Guard}} \cup F_{\text{Action}} \\
F_{\text{Guard}} &= \left\{ \left[i < p \wedge \#I_{\mathrm{H}} > 0 \wedge \#I_{\mathrm{V}} > 0 \wedge \#O_{\mathrm{H}} < 2 \wedge \#O_{\mathrm{V}} < 2 \right], \left[i > p \right] \right\} \\
F_{\text{Action}} &= \left\{ \left[c_{i,j} \leftarrow c_{i,j} + I_{\mathrm{V}} \cdot I_{\mathrm{H}}; i \leftarrow i + 1; O_{\mathrm{H}} \leftarrow I_{\mathrm{H}}; O_{\mathrm{V}} \leftarrow I_{\mathrm{V}} \right], \left[\text{stop}() \right] \right\}.
\end{aligned}
\tag{9.3}
$$

The actor's FSM R_{Cannon} is given graphically in Figure 9.2b. In the initial state, "Calc", the actor waits until each of its two input ports have at least one token available $(\#I_{\mathrm{H}} > 0 \wedge \#I_{\mathrm{V}} > 0)$, and until there is at least one free spot in each of its output ports $(\#O_{\mathrm{H}} < 2 \wedge \#O_{\mathrm{V}} < 2)$. Once that happens, the state transition from "Calc" to "Calc" may be taken, and its action $\left(c_{i,j} \leftarrow c_{i,j} + I_{\mathrm{V}} \cdot I_{\mathrm{H}}; i \leftarrow i + 1; O_{\mathrm{H}} \leftarrow I_{\mathrm{H}}; O_{\mathrm{V}} \leftarrow I_{\mathrm{V}}\right)$ performed. The action takes the tokens from the input ports, multiplies and adds them onto the local partial result $c_{i,j}$, and enqueues them on their corresponding output ports. Finally, the internal counter is incremented to keep track of the overall completion of the computation. The other transition, from "Calc" to "End", is taken once the actor's internal counter is equal to the number of processors in one dimension, i.e. once every matrix block needed for $c_{i,j}$ has been processed. Its action serves to signal the termination of the actor.

An important aspect of the FunState model is the clear separation of concerns that occurs between the different parts of the computation. Actors do not need to possess knowledge about their neighbors (or global state). Instead, they only depend on their own internal state, and the information provided in the tokens received through the ports. There is also a clear separation between the execution behavior, the communication, and the performed functionality through the separation into the FSM R and the set of functions that may be performed. Finally, the

formal model does not prescribe any particular model of computation, but only requires the basic conditions to be met, e.g. that values are not consumed before a state transition actually takes place. This allows for great flexibility when implementing the model in practice, and leads to a better separation of concerns, where the application developer may focus on the actors and their interactions, and the library developer takes care of the execution of the actors efficiently, e.g. on a distributed machine. In the following chapters, I present two incarnations of the actor model, one written in X10 within the context of the Invasive Computing project, and one written using more widely available standard tools, such as modern C++ and the UPC++ communication library.

10. ActorX10, an X10 Actor Library

In chapter 5, I introduced X10 as an example for a programming language built on the APGAS computational model. X10 is the prevalent programming language used in InvasIC, and was therefore the first choice for our actor library to facilitate the best possible integration into the project. Our implementation of the actor model is an improvement on the more simple actor implementation[1] described by Roloff, Hannig, and Teich (2014).

The APGAS paradigm and X10 introduce the notion of places, i.e. shared memory domains that code may be executed on. When X10 is used on a distributed memory system, one would typically choose the place granularity to be one place per NUMA domain, with the whole computation potentially spanning the entire cluster. Mapping the actor model onto the APGAS paradigm also enables us to perform large-scale computations using the model. In this chapter, I will describe the ActorX10 library that maps the FunState actor model as described in chapter 9 onto the APGAS programming model as realized in X10. The work described in this chapter is based on a collaborative effort that has been presented previously in Roloff et al. (2016).

10.1. System Design

The ActorX10 system design closely follows the structure mandated by the FunState model, and results in an object graph that is distributed amongst the X10 computing domain. There is a single `ActorGraph` instance that typically resides on the root place, and multiple actors spread across the domain. The choice for a single graph instance was based on the architecture of the invasive platform prototype. Having an actor graph instance on all ranks would have meant a large amount of replicated information without much benefit, as the ActorGraph graph instance is not needed for the computation once the actors are started. On HPC systems, this choice becomes a bottleneck during the initialization of the computation.

The application developer implements their functionality through subclassing. The application will provide an `ActorGraph` subclass to implement the creation and setup of the actor-based computation. An overview of the class structure of ActorX10 is depicted in Figure 10.1. The actor's functionality is provided by subclassing the `Actor` class. In the following, we will provide a closer look at the individual components: actors, ports and channels, and the actor graph.

[1] Referred to as task model in the publication

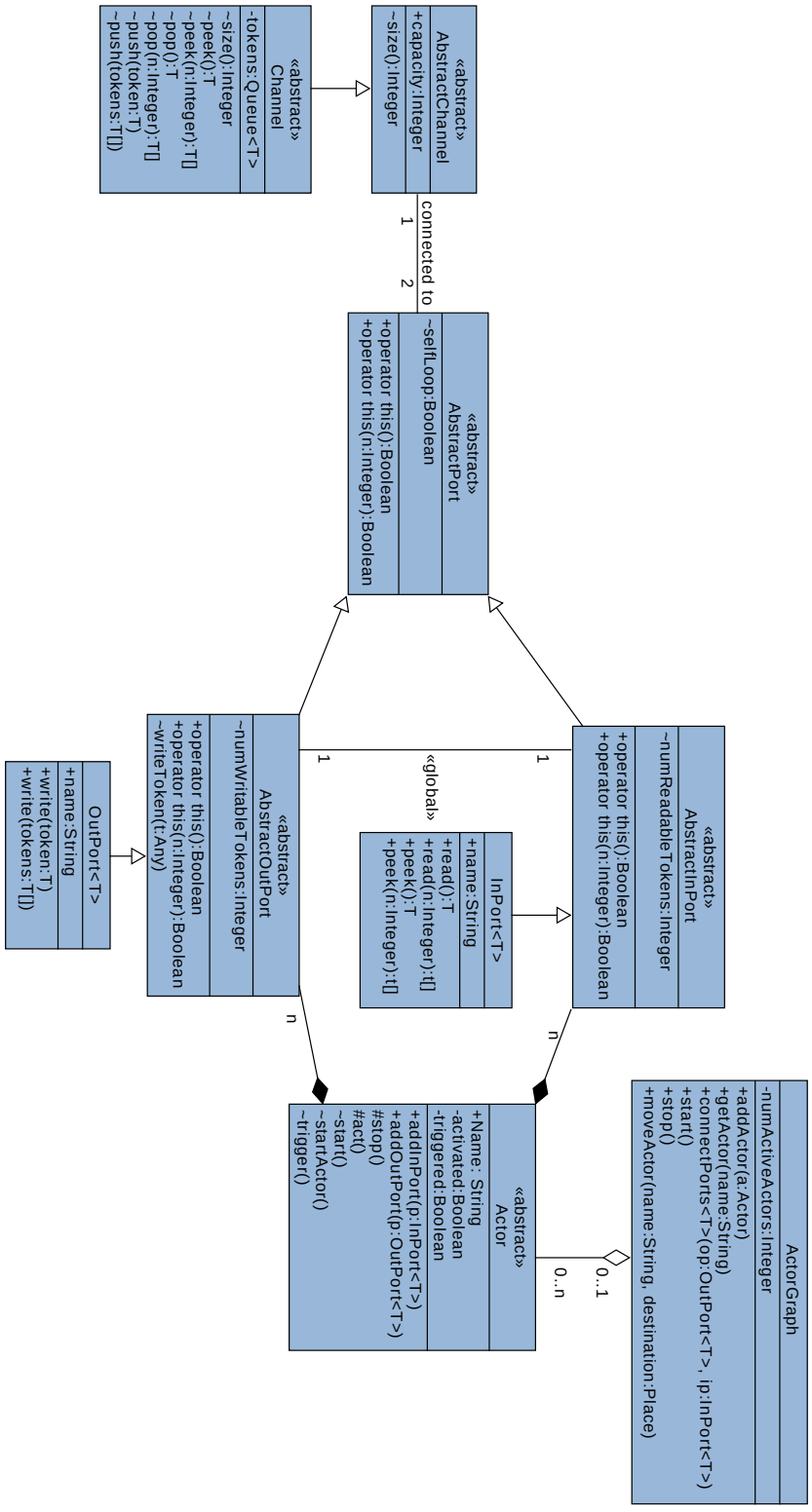


Figure 10.1.: ActorX10 System Design UML Class Diagram

10.2. Actors

In accordance with the formal definition of the actor in the FunState model, the abstract class `Actor` contains two lists representing the set of incoming ports I and the set of outgoing ports O. Ports are added to the actor using the corresponding factory methods `addInPort(port)` and `addOutPort(port)`. The actor class provides the abstract method `act()` which is called whenever the state of the connected channels of the actor changes. Ports and actor need to have a unique name. The actor class is subclassed by the application developer. It is possible to provide an arbitrary constructor, the only requirement is the implementation of the `act()` method. The subclass will typically keep instances of its ports as attributes.

On a functional level, actor execution is driven through X10 activities. Upon the start of the actor graph execution, the actor is set to active, and its X10 activity is spawned on the place that contains it. The activity enters an event loop. In the loop, first the `act()` method is invoked. Subsequently, all ports are checked for updates. If there are new tokens available, or spots in the queue have newly been consumed by connected actors, the actor is activated again, the actor keeps track of the event by invoking the `trigger()` method. If the actor has been triggered, the FSM is invoked once more, and may perform another state transition. Otherwise, the activity is suspended, and remains so until there is a change in the connected channels. This loop is executed until the actor's FSM signals the arrival in an accepting state, through invocation of the actor's `stop()` method.

10.3. Ports

An actor's only means of communication with its environment is through ports. This enabled the application developer to precisely describe the type of data that is consumed by the actor, as well as what data is created. We implemented this in ActorX10 with the generic classes `InPort[T]` and `OutPort[T]`. This ensures at compile-time that actors only receive tokens they are able to consume: First, the generated code only allows for tokens of a specific type T, and second, it is only allowed connect ports of the same token type through a channel. Actors may `read()` from `InPorts`, and `write()` to `OutPorts`. For some state machine transitions, it is necessary to evaluate one or more tokens before a transition is taken. The tokens are read from the channel only when a feasible state transition has been identified[2]. New tokens are written once all actions associated with the transition have been performed. Obtaining tokens from channels is possible non-destructively through invocation of the `peek()` method. When no tokens are available to read for `InPorts`, or no free space to write tokens for the `OutPorts`, all the methods to access channels will fail with an exception. To avoid this, the application developer has to insert checks into the code to make sure that channels are only read from or written to if it is actually possible to do so. These checks are provided by both types of ports using the overloaded `operator()` (e.g. for `InPort` p, `p()` would check whether a token is available, `p(5)` would return `true` if there are at least five). For `InPorts`, the method returns whether one (for no parameter provided) or a specific number (when that

[2] Note that whereas the formal actor model introduced in chapter 9 assumes a non-deterministic choice of transition in case multiple transitions should become simultaneously activated, our X10 implementation implements a priority given by the order in which the code checks the activation conditions.

number is provided as parameter) of tokens are available. `OutPorts` function the same way, but instead of available tokens, the number of unused spaces are checked.

To provide this functionality, ports hold references to the channel that connects them to their counterpart, and to the port on the connected actors. The former is used to insert, access and remove tokens, and the latter is used to signal the tokens' insertion or deletion. Global references are used since actors and channels are potentially distributed between multiple places. When a transfer of tokens is necessary, we start a new asynchronous activity on the place containing the channel to insert the token. After doing so, that activity asynchronously spawns another that notifies the connected `OutPort`. For the transfer of tokens, we benefit from the sophisticated object serialization in X10. When tokens are transferred across place boundaries, they are serialized on the source place, sent over the interconnection network, and then reassembled on the destination place. This enables the use of nearly any X10 type as tokens[3].

10.4. Channels

Channels are implemented by the generic class `Channel[T]`. Like the two port classes, it has a type parameter `T` to ensure that only tokens of compatible types may be enqueued. Tokens are stored in a thread-safe queue that allows for read and write operations without race conditions. In addition to methods for adding and removing elements, there is a method to access the first element in the queue without removing it (`peek()`).

10.5. ActorGraph

The `ActorGraph` bundles functionality for managing the components of the actor-based computation. It offers methods to add new actors to the graph (`addActor(a)`), to connect ports (`connectPort(i, o, n)`), and to distribute it (`distributeActors()` and `moveActor()`). Users of the library are expected to perform the set-up of the actor graph by creating their own subclass of `ActorGraph`, with an implementation of the `build()` method. To start the computation, the `start()` method is invoked. The method call returns only once the computation has finished, and all actors are stopped.

Internally, the actor graph class tracks existence and location of all its actors through a list of global references. Channels are tracked only implicitly. When two actors' ports are connected, the `ActorGraph` instance takes care of setting the references correctly by allocating global references, and a channel object. The default behavior of the actor graph allocates the channel object on the same place as the actor containing the `InPort`. Distribution of actors may happen before the actors have been activated, or during the actor graph execution. When the graph is inactive, actors, ports and channels may simply be moved to another place, and the X10 runtime takes care of the data serialization. If an actor is to be moved after the computation has started, the actor and all its

[3] Excluding, amongst others, types containing operating system level resources, unmanaged C++ data, or file handles.

neighbors[4] first have to be stopped. This is done by signalling a temporary termination signal. Once all involved actors finish their currently running computation, actors, ports and channels may be moved safely. Thereafter, the computation is restarted. Currently, the actor library only allows for the migration of a single actor at the same time.

10.6. ActorX10 Application Example: Cannon's Algorithm

In chapter 9, I formally described Cannon's algorithm in terms of the FunState model. In this section I will demonstrate its parallel and distributed implementation using ActorX10. For the operation on matrices, I use the `Matrix` class introduced in section 5.1 (Figure 5.2). In the example, I implemented the classes `CannonActorGraph` and `CannonActor`, which inherit from `ActorGraph` and `Actor`, respectively. In the following, I will take a closer look at the implementation[5] of these classes.

Following the formal definition, the class `CannonActor`, depicted in Figure 10.2 has two incoming and two outgoing ports that accept tokens of type `Matrix`, one for each spacial direction in the actor grid. Furthermore, there is an attribute to track the state the computation is currently in. The FSM is implemented in the `act()` method (line 11-25). There are two states, `STATE_COMPUTE` and `STATE_FINISHED`. In the compute state, as long as the actor has not yet received all the matrices from its row (and column), the actor may perform a state transition if there is sufficient space to place a token in `above` and `left`, and there is one token available in each of the ports `right` and `below`. If that transition is taken, the actor takes the partial matrices from the incoming ports, and multiplies them onto the result matrix `c`. Finally, the input matrices are placed on the outgoing ports conforming to the direction they originated from. The state remains the same in this transition, however, I increment a counter to keep track of the number of matrices that still need to be processed to obtain the final result. The other transition, from `STATE_COMPUTE` to `STATE_FINISHED`, is taken once the final result is obtained. Here, I set the actor's state accordingly, and end the actor's execution through invocation of the `stop()` method.

In `CannonActorGraph`, shown in Figure 10.3, the `build()` method is implemented to build the actor graph incrementally. In the beginning, $p \times p$ `CannonActor` instances are created and stored in a two-dimensional array *(lines 12-17)*. The actors are created each with their position within the overall computation, the size of their working set and the number of intermediate steps they need to compute. The newly created actor is added to the actor graph, and its ports are registered internally. Once all actors are created, they are connected *(lines 20-26)*. For each actor, I connect its `left` outgoing port to its left neighbor's `right` incoming port, and its `above` outgoing port to its top neighbor's `below` incoming port. For actors at the boundary of the computation, connections are created by wrapping around and connecting to the actor on the other side of the domain. This may be done using the modulo operator. Then, I distribute the actors *(line 29-35)*. First, I create a two-dimensional block-block-distribution[6] over the point cloud of the computation, and then I

[4] i.e. all actors that share a channel with it.

[5] The code shown in this section is shortened and lightly edited for readability. For the full sample code, see section A.1 in the appendix.

[6] X10 uses the Dist class to map array iteration spaces to Places for distributed arrays.

```
1  class CannonActor extends Actor {
2      val above : OutPort[Matrix];
3      val left : OutPort[Matrix];
4      val right : InPort[Matrix];
5      val below : InPort[Matrix];
6      private var state:Int = STATE_COMPUTE;
7
8      // Other attribute definitions, constructor,
9      // and initialization methods are omitted.
10
11     protected def act() {
12         if (state == STATE_COMPUTE && operationsPerformed < numOperations
13             && left() && above() && below() && right()) {
14             val a = below.read();
15             val b = right.read();
16             this.c = this.c + a * b;
17             operationsPerformed++;
18             left.write(b);
19             above.write(a);
20         } else if (state == STATE_COMPUTE && operationsPerformed ==
           ↪  numOperations) {
21             state = STATE_FINISHED;
22             stop();
23         } else if (state == STATE_FINISHED) {}
24     }
25 }
```

Figure 10.2.: ActorX10 sample actor implementing Cannon's Algorithm

```
class CannonActorGraph extends ActorGraph {
    val n : Int;
    val p : Int;
    val actorRegion : Region{rect, zeroBased, rank==2};

    // Constructor omitted.
    //  - Sets instance variables.
    //  - Ensures that n mod p = 0

    def build() {
        // Create the actors and register their ports
        val actors = new Array[CannonActor](actorRegion, (pt:Point(2)) => {
            val a = new CannonActor(pt(0), pt(1), n/p, p);
            addActor(a);
            a.initPorts();
            return a;
        });

        // Connect ports to neighbors
        for ([i,j] in actorRegion) {
            val a = actors(i,j);
            val left = actors((i + p - 1) % p, j);
            val upper = actors(i, (j + p - 1) % p);
            connectPorts(a.left, left.right, 2);
            connectPorts(a.above, upper.below, 2);
        }

        // Distribute the Actors using a 2D-Block-Distribution
        val distPlan = Dist.makeBlockBlock(actorRegion, 0,1);
        val distribution = new HashMap[String, Place]();
        for (p in actorRegion) {
            val plc = distPlan(p);
            distribution.put(actors(p).name, plc);
        }
        distributeActors(distribution);

        // Place initial matrix tokens in the incoming ports
        finish for ([i,j] in actorRegion) async {
            val aRef = getActor(actors(i,j).name);
            aRef.evalAtHome((a:Actor) => {
                (a as CannonActor).placeInitialTokens();
                return 0;
            });
        }
    }
}
```

Figure 10.3.: ActorX10 sample actor graph for Cannon's Algorithm

use it to create a Map that maps each actor name to the place it was assigned by the distribution object. Finally, I pass the map to the superclass, which takes care of the actor distribution. The last step is to place the initial matrix tokens in the incoming ports of the actors *(lines 38-44)*. This task may be performed asynchronously for all actors on all places involved in a computation.

11. An Actor Library for UPC++

In addition to the actor library written in X10, I also implemented a second version of the library using tools more generally available on current systems, namely modern C++ and the UPC++ communication library. UPC++ implements the PGAS paradigm on top of modern interconnection networks (for details, see chapter 4). UPC++ is available on some clusters (such a Cori[1] at National Energy Research Scientific Computing Center (NERSC)), and may be installed without administrative privileges on others (such as LRZ's CoolMUC 2 [2]). The combination of more simple interoperability with standard HPC environments made it an appealing target for another implementation of the actor model. I previously presented this library in Pöppl, Bader, and Baden (2019), the description in this chapter is based on this publication.

The system design of Actor-UPC++ follows a design similar to ActorX10, with a few notable differences. Whereas in ActorX10, there is only one central actor graph instance for all places, in Actor-UPC++, the actor graph is implemented as a distributed object. Application developers may interact with the actor graph instance instead of subclassing it. Furthermore, the inheritance hierarchy has been flattened by removing the `AbstractPort` class. In the following, I will look at the implementation of Actor-UPC++ in more detail, with a focus on the implementation choices taken as a consequence of the change in the implementation environment.

11.1. Actor Graph

As in ActorX10, the coordination of the graph setup is performed in the `ActorGraph` class. Similarly, the class holds a dictionary of all actors involved in the computation, and as in the other implementation, operations to add actors (`addActor(a)`), to connect ports (`connectPorts<type,capacity>(a1, p1, a2, p2)`) and to start the computation (`run()`) are offered. The implementation, however, is different. Following the SPMD process model of UPC++, the actor graph is collectively created on all ranks instead of only one instance on one place in X10. This allows every rank to add its own actors. When a new actor is added, it is added to a list of local actors, and to the global dictionary. Finally, the new entry is broadcast to all other ranks involved in the computation. This means that each instance will eventually possess a full copy of the dictionary. The main use of this dictionary is the discovery of remote actors for the purpose of connecting their ports. The method `connectPorts()` may be called from any rank. Depending on the location of the source and the destination actor, different connection strategies are used. When the source is local, I first schedule an RPC on the destination actor's rank, and attach a

[1] `https://docs.nersc.gov/systems/cori/`

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

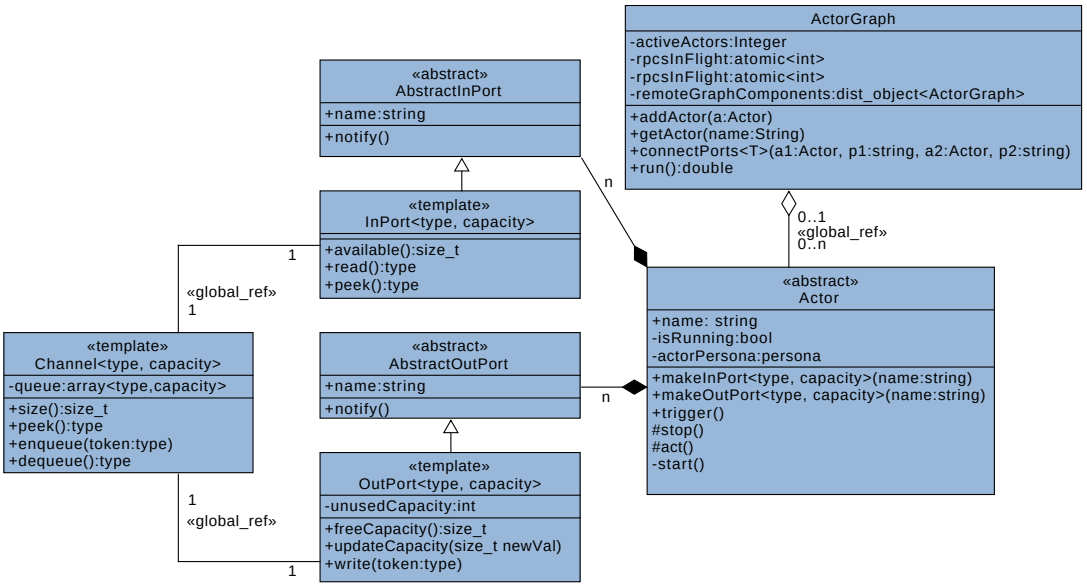


Figure 11.1.: Actor-UPC++ System Design UML Class Diagram

closure that connects the source side as an asynchronous completion. If the destination actor is local, I create the channel locally, and then asynchronously perform the source side using an RPC. Finally, if neither source nor destination actor are local, I simply invoke the operation to connect from the destination actor using an RPC to the rank containing it. In all three cases, I wait for completion of the operation before returning from the method call to `connectPorts()`. Finally, to start the computation, the ranks involved in the computation must collectively invoke the `run()` method. As in ActorX10, this method call will only return once all actors have been terminated.

Scalability Issues In my evaluation of the actor library, the performance of the current implementation of the actor graph dictionary was sufficient. However, I anticipate the dictionary to become a problem for computations with thousands of nodes. The amount of memory used per rank may be computed as

$$m_{\text{total}} = m_{\text{entry}} n_{\text{Actors}} \frac{n_{\text{Ranks}}}{n_{\text{Node}}} \tag{11.1}$$

and the network traffic is computed as

$$\begin{aligned} t_{\text{Actor}} &= m_{\text{entry}} n_{\text{Ranks}} \\ t_{\text{Total}} &= t_{\text{Actor}} n_{\text{Actor}}. \end{aligned} \tag{11.2}$$

The largest job I performed in the course of the evaluation of my sample application was executed on 128 Xeon Phi nodes on NERSC Cori. The job encompassed 65535 actors on 16384 ranks. Each dictionary entry contains the name of the actor as a `std::string` object (with a length of about 50 characters) and the `upcxx::global_ptr` object referencing the actor (with a total size of 16B) leading to a total message size of $\approx$ 100B. Following Equation 11.1, the dictionaries on a single

node would take a total of $100\text{B} \cdot 65536 \cdot \frac{16384}{128} = 800\text{MB}$. With a ten-fold increase of nodes in the computation, the memory requirements would become prohibitively large. Another problem is the increase in network traffic due to the broadcast of new entries. Following Equation 11.2 and assuming a constant number of actors per node with an increasing number of nodes in the computation, the number of messages and, therefore, the total network traffic increases quadratically with the number of nodes involved in the computation.

However, a global broadcast of all actors to all ranks will typically not be necessary. In most cases, each actor will only have a limited number of direct neighbors. Furthermore, once the actors' ports are connected, the dictionary is not used anymore, as actors communicate directly, using their ports. A solution to the Scalability problem may also be a distributed hash table such as the ones proposed in Bachan et al. (2019), or the one by Monnerat and Amorim (2015). Here, the values in the table are distributed amongst the participating ranks. When a new value is to be inserted, the hash value is computed on the initiating rank, along with the target rank. The value is then inserted using some kind of remote operation (Bachan et al., 2019). In the case of the actor library, I would store the references as the values in the global hash map, and use the actors' names as the key. When two actors' ports are to be connected, I may then perform a lookup to obtain the references and then go on as described above.

11.2. Actors and Execution Strategies

The interface and functionality of the `Actor` class from the application developer's view are very similar to the one in ActorX10. It contains dictionaries of its incoming and outgoing ports, and offers methods to create them (here using the template methods `makeInPort<type, capacity>(name)` and `makeOutPort<type,capacity>(name)`, respectively), and it provides an abstract method `act()` that is implemented by the application developer in order to provide the actor's FSM. The necessity of giving the FSM a chance to perform a state transition is tracked using the actor's trigger count. Whenever the state of the actor's ports change, the trigger count is increased by one, leading to an invocation of `act()`, which in turn decreases it by one. Within the library, I implemented and experimentally evaluated three different strategies of executing actors. They are presented below.

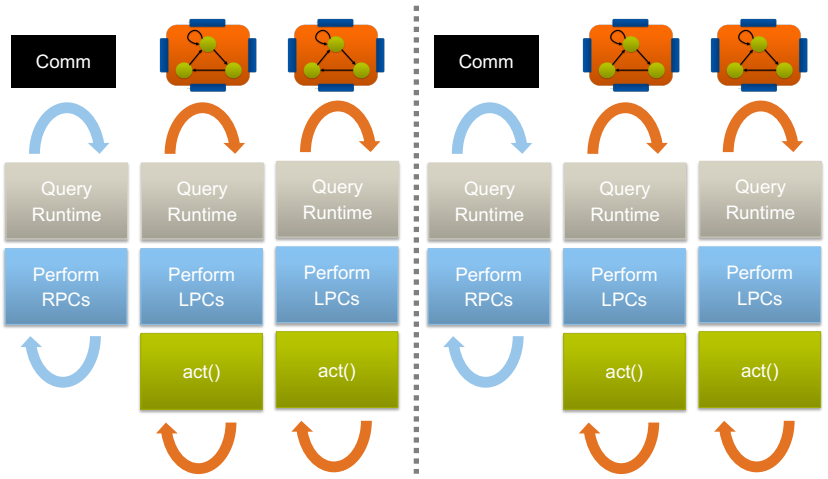
The scope of UPC++ is mostly based around inter-rank communication. In contrast to X10, which is a full parallel programming language, it does not prescribe a canonical style to parallelize an application. And while C++-14 offers basic parallelism in the shape of threads, application developers are free to use different parallelization solutions, such as OpenMP. Therefore, I use the UPC++ functionality mainly for communication, and parallelized the actor execution using three different parallelization strategies: The first strategy relies on UPC++ ranks, the second one on C++ threads and the third one on OpenMP tasks. Henceforth, I shall refer to them as *rank-based execution strategy*, *thread-based execution strategy* and *task-based execution strategy*, respectively. In all three cases, they implement the semantics of the FunState actor model as presented in chapter 9. To do so, ports need to be observed for changes in their respective connected channels. Once such an event takes place, the port's corresponding actor needs to be activated and its state machine given the chance to perform a state transition. For all three strategies, this leads to the

implementation of one or more *event loops* per rank. Within these loops, typically tokens are made available to the application, and the `act()` method is invoked on the affected actors. Figure 11.2 schematically depicts all three execution models.

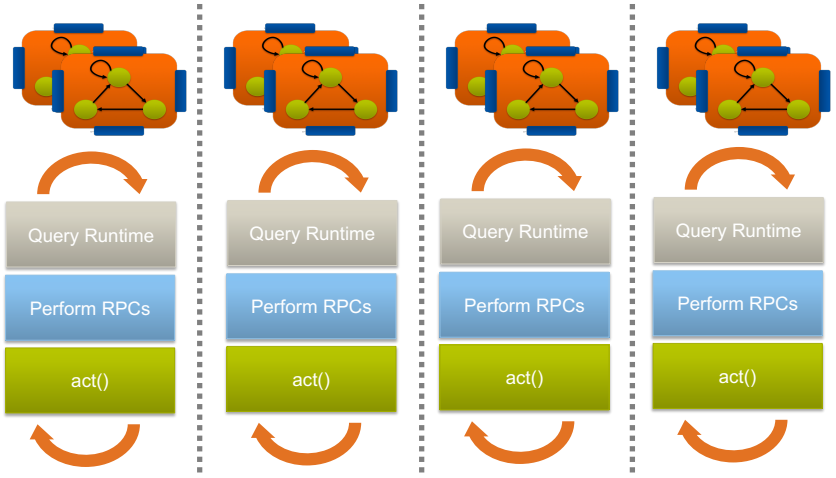
Thread-Based Execution Strategy This strategy relies on the threading functionalities provided by the C++ standard library. When it is used, each actor instance is mapped to its own operating system thread, while the actor graph instance is executed on the rank's main thread. This mapping is made apparent to UPC++ through the use of personas (as introduced in section 4.2). Each actor takes on its thread's persona, while the actor graph assumes the master persona of the process. All actors as well as the actor graph have their own event loop. The actor graph's event loop is responsible for inter-process communication. It continuously queries UPC++ for progress so that incoming RPCs that insert new tokens or update queue capacities from actors on other ranks may be processed in a timely manner. After each RPC, an LPC is sent to the persona of the actor affected by the change. The actors' event loops initially call for progress as well, to give the runtime the chance to execute the LPCs sent by the actor graph. If an actor was triggered, its `act()` method is invoked. The use of multiple threads per UPC++ rank mandates the use of the parallel UPC++ backend.

Rank-Based Execution Strategy When this strategy is used, the execution of each rank is performed sequentially, i.e. there is only a single thread per UPC++ rank. Parallelism is achieved instead through the use of many ranks per physical node, e.g. one rank for every logical core on the node. For this strategy, one would typically have a rather small number of actors per rank, as the actors are all processed sequentially. Now, only a single event loop within the `ActorGraph` instance of the rank is needed. In it, I first query the UPC++ runtime for progress, giving it the opportunity to process incoming RPCs. Afterwards, all actors that have been triggered by RPCs or by other local actors, get a chance to perform a state transition by having their `act()` method invoked. A notable advantage of having only a single thread per UPC++ rank is that one may use the sequential UPC++ backend, which forgoes synchronization and thus performs better compared to the thread-safe parallel UPC++ backend.

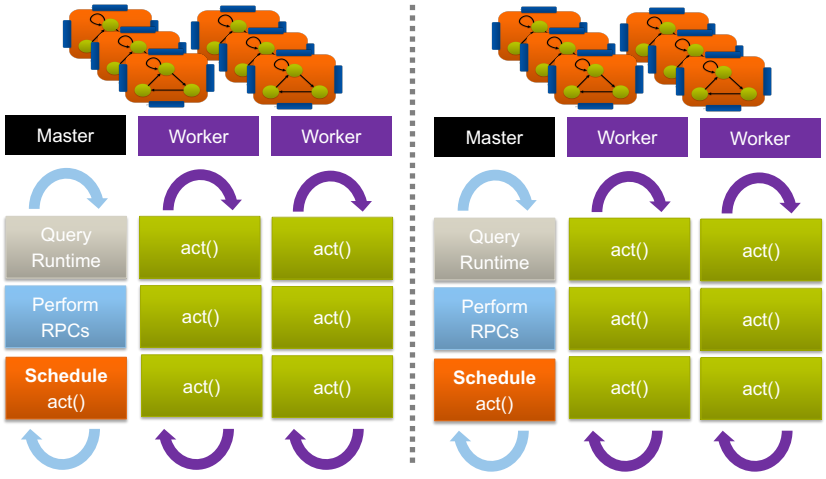
Task-Based Execution Strategy The third strategy parallelizes the actor execution using OpenMP tasks. It has the same basic approach as the rank-based execution strategy, with a single event loop for all actors. However, instead of executing the `act()` method directly, the invocations are scheduled onto an OpenMP task. The basic approach is the following: First, I allow the UPC++ runtime to execute incoming RPCs by calling `upcxx::progress()`. Then, I iterate over all actors on the rank, and schedule an OpenMP task that queries for progress and then executes `act()` for all actors that have a positive trigger count. These tasks are then scheduled by the OpenMP runtime onto worker threads, and executed concurrently to the application's main loop. Additionally, I need to communicate to the OpenMP runtime that tasks executing the `act()` method of the same actor need to be executed after each other. This may be done by adding the `depends(...)` clause to the OpenMP pragma that creates the task. In my case, the depends clause is a bi-directional dependency on the memory address of the actor that has its `act()` invoked. This allows the runtime to schedule tasks from different actors in parallel while keeping the execution within



(a) Thread-Based Execution Strategy

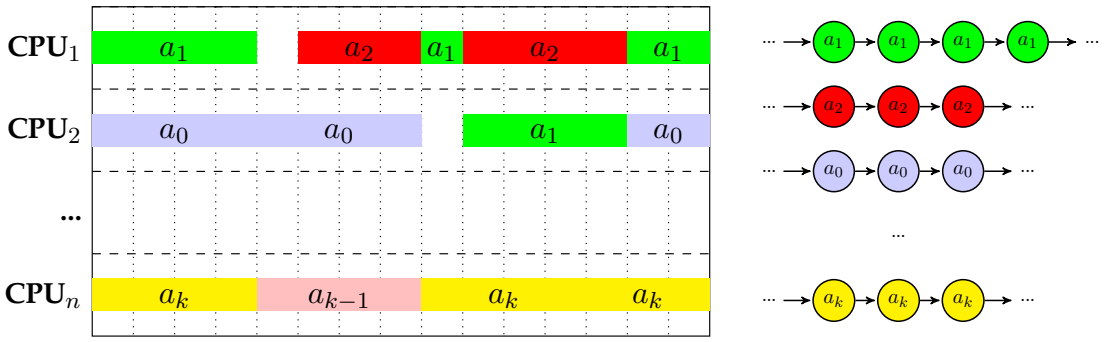


(b) Rank-Based Execution Strategy



(c) Task-Based Execution Strategy

Figure 11.2.: Parallel execution strategies for Actor-UPC++. (Pöppl, 2019)



(a) Sample Execution Trace. From (Pöppl, Bader, and Baden, 2019)

(b) Task Dependency graph

Figure 11.3.: Sample execution trace and corresponding task dependency graph for the task-based execution strategy. Tasks belonging to the same actors need to be executed sequentially.

an actor sequential. A possible execution trace is depicted in Figure 11.3. The number of worker threads for the OpenMP runtime may be set by the user invoking the application. Typically, the number of workers per node is similar to its number of cores.

Discussion The three different execution strategies are based on different approaches to parallelization, each with its distinct set of advantages and drawbacks. The thread-based execution strategy follows the approach taken also by ActorX10, and stays close to the original idea of actors being independently acting objects. As each actor has its own resource (an operating system thread), it is easier to analyze the resource use on a per-actor basis, and, as long as there are enough resources available, actors will be quick to respond to incoming information. However, this approach has significant drawbacks: First and foremost, using this strategy, actors consume resources whenever they are scheduled. Even when they do not have any work to perform, whenever their thread is scheduled to compute by the operating system, they will keep polling for updates, and therefore consume energy and take compute resources that other actors with incoming messages actually need. Furthermore, operating system level threads are expensive, and a context switch between threads involves storing all its data in memory, and therefore incurs a significant overhead. This makes this strategy the best if there are only few actors (or better just a single one) per compute resource. This, however, means that there is little to no overlapping of computation and computation possible, as there are not enough actors that may still have a positive trigger count. Finally, this strategy uses a fixed communication thread, which queries the runtime constantly for updates. This takes another compute core that might also at least be partially used for computations otherwise.

The rank-based execution strategy follows the structure of a classical SPMD application. Each rank's computations are performed sequentially using the master thread of the process. For intra-node parallelism, I then use multiple ranks, typically one per logical CPU core. Each rank will typically be responsible for a few actors. This has several benefits: The structure of the event loop means that all compute resources share some of the communication load, each for a small amount of actors, and while the loop also actively polls for updates, after each call to the

runtime, actors with a positive trigger count are given the chance to compute. This also means that actors without a positive trigger count do not utilize any resources. Furthermore, the fact that the execution within a rank is sequential has the added benefit that the implementation may be rather simple, as there are no shared resources, and no thread synchronization is necessary. Intra-rank communication can therefore be performed directly without the need for UPC++ communication operations such as LPCs. However, low number of actors per rank makes the library a bit less flexible when it comes to load-balancing. If the load of an application is not balanced well across the actors involved in the application, there may be ranks that do not have any work to perform, while other actors on ranks under greater load may hold up the computation. While this drawback is also present in the other strategies due to a lack of actor migration, using this strategy the problem is exacerbated, as the number of actors per rank is lower. Finally, the low number of actors per rank means that more RPCs need to be performed, as there is more communication across rank boundaries.

The task-based execution strategy follows the basic structure of the rank-based execution strategy, but instead of executing the actors itself, work is offloaded from the main thread onto worker threads. This allows one to use a more coarse-grained rank structure compared to the rank-based execution strategy, for example one rank for each NUMA-domain. Compared to the thread-based execution strategy, I use the available resources more efficiently, as the actor graph only schedules tasks for actors with a positive trigger count; therefore, idle actors consume no resources. While this strategy also has a main thread mostly responsible for communication (and scheduling tasks), this thread is able to execute the tasks it creates directly if there is a high load on the worker threads (using the "mergeable" clause of the OpenMP task pragma). Compared to the rank-based execution strategy, the higher number of actors helps to reduce the overall number of RPCs that the UPC++ runtime needs to handle. However, for the inter-rank communication that does occur, LPC completions are needed for book-keeping. A drawback of this strategy is the balancing of the rank-granularity. In some of my tests (see Figure 18.7 in section 18.2), the communication thread became a bottleneck of the computation. This can be alleviated by increasing the number of ranks per node to an appropriate value. This value is dependent on the hardware configuration, the compute-intensity of the actors' actions and the communication intensity of the problem to be solved.

11.3. Ports and Channels

Following the FunState model as introduced in chapter 9, communication between actors happens through ports and channels. In Actor-UPC++, ports and channels are each implemented using class templates (`InPort<type, capacity>`, `OutPort<type, capacity>` and `Channel<type, capacity>`) with a distinct `type` and fixed and finite `capacity`. They offer operations to handle the insertion of tokens into channels using `write()` and their extraction using `read()`. The channels use a ring buffer to store the tokens internally. As a limitation, I currently only support tokens that may be transferred using UPC++ RPCs. Simple types such as numbers (both integral and floating-point), character strings or `std::vector<T>` objects may be transferred directly; for others, the application developer has to specify serialization code. Furthermore, I restricted the placement of the channel objects insofar that they are always placed on the rank of the receiving actor, so

that messages need not be transferred or buffered when a read takes place. The communication scheme is shown in Figure 11.4.

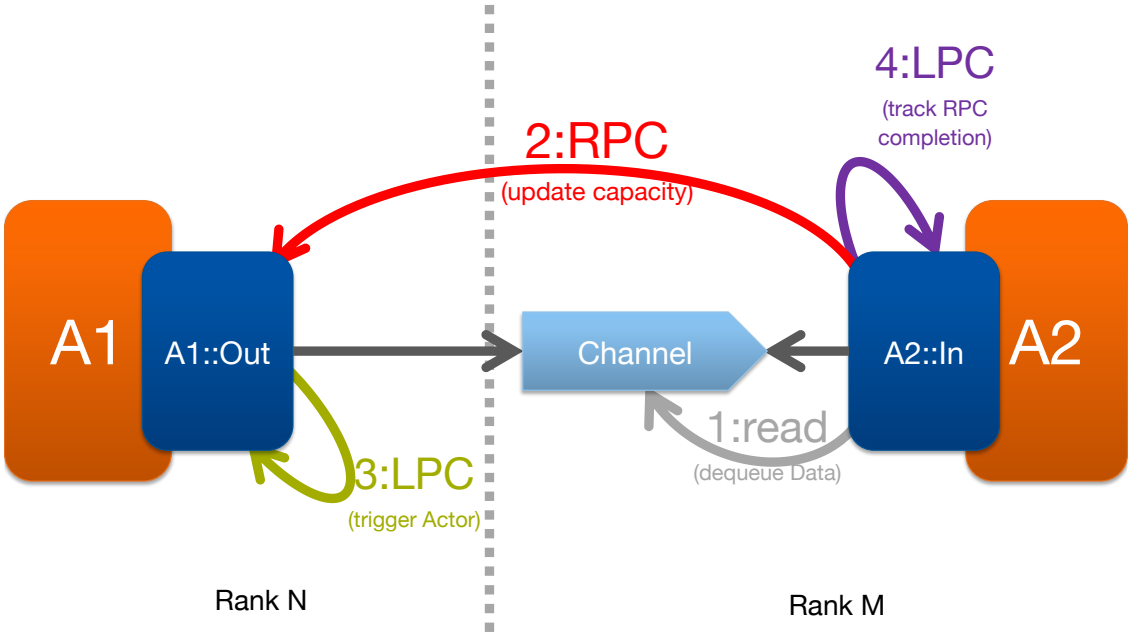
When an actor *A1* writes a token to one of its ports ("Out"), the port is responsible for adding the token to the channel. If the channel is on the same rank as the sending actor, the channel may be accessed directly and the token is simply inserted. Otherwise, the insertion is performed within an RPC to the receiving actor's rank. Once the insertion has been performed, the receiving actor *A2* is notified of the change. For the thread-based execution strategy, this is done using an LPC, otherwise, the actor may be notified directly. Read operations work similarly. The receiving actor *A2* initiates the operation by calling `read()` on an incoming port. First, the port removes the data from the channel. Once that is done, the corresponding outgoing port at the sending actor needs to be notified of the capacity change in the channel and the sending actor needs to be triggered. Again, this will be done using RPCs and LPCs, where necessary.

A Note on Quiescence The UPC++ runtime does not keep track of its communication operations. Instead, it is left to the application developer to make sure that all communication is received correctly. The UPC++ developers refer to a state where there are no more messages on the network as quiescent. If that state is not reached before the termination of the application, the behavior of the application is undefined (Bachan, 2019). In the case of Actor-UPC++, this pertains to the use of RPCs and LPCs, and application termination. In some cases, all actors may already be terminated while there are still unfinished communication operations, either on the network, or in the UPC++-internal queues. This causes problems during application termination. When the UPC++ runtime is deinitialized, the queues are drained and any pending operations executed. These operations then try to reference memory segments that have previously deallocated, which leads to segmentation faults and, consequently, to application crashes.

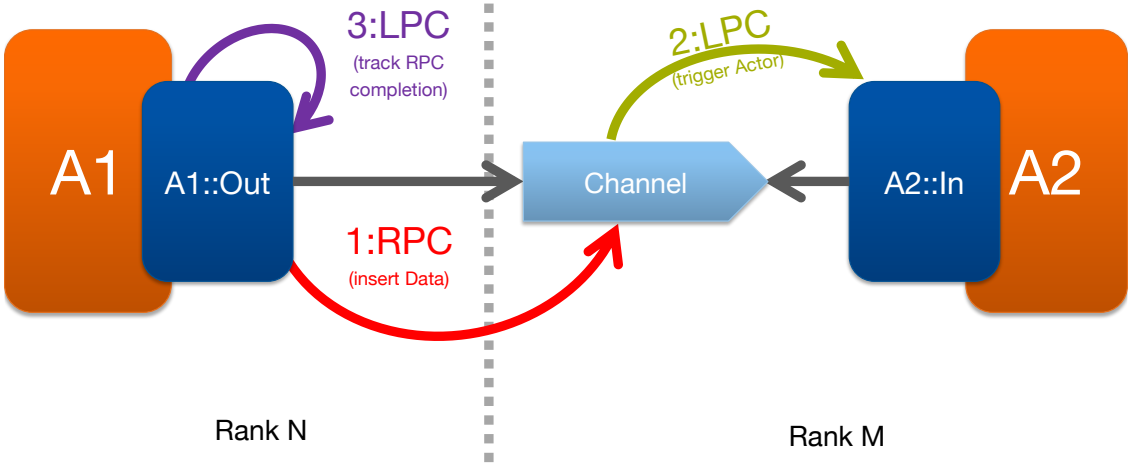
To avoid this, the actor library needs to track the initiation and completion of all RPCs and LPCs manually. I solved this using atomically accessible counters within the actor graph instances. Whenever a port has to perform a remote operation, it increases the RPC counter within the local actor graph instance, and attaches a completion, either a lambda closure or an LPC. LPCs may be tracked directly. Once all actors within the rank are terminated, the actor graph (and the actors for the thread-based execution strategy) call `upcxx::progress()` until the RPC and LPC counters reach zero. This is done on all ranks, and only once all ranks have no more communication operations in progress can the actor graph be torn down.

11.4. Actor-UPC++ Application Example: Cannon's Algorithm

As with the ActorX10 library, its use is best demonstrated using a code example. Therein, as before, I implemented an actor-based version of Cannon's Algorithm. In this section, I will demonstrate it with a focus on the differences in the implementation compared to the X10 version. The code samples in this section are lightly edited and shortened for readability, a full version of the code is shown in section A.2 in the appendix. In the ActorX10 example, I closely followed the theoretical definition, and used a `Matrix` class as the token type for the channels. I did the



(a) Read Operation. (Pöppl, Bader, and Baden, 2019)



(b) Write Operation. (Pöppl, Bader, and Baden, 2019)

Figure 11.4.: Inter-Actor Communication in Actor-UPC++.

```
struct Matrix {
    size_t rows;
    size_t cols;
    std::vector<float> data;

    UPCXX_SERIALIZED_FIELDS(rows, cols, data)

    Matrix();
    Matrix(size_t rows, size_t cols, std::function<float(size_t, size_t)> init);
    Matrix operator* (Matrix &other);
    Matrix operator+ (Matrix &other);

    std::string to_string();
};
```

Figure 11.5.: Actor-UPC++ Matrix token with UPC++ serialization enabled through the use of a macro.

same in the Actor-UPC++ version, through the implementation of a simple `Matrix` class in C++. However, unlike in X10, the UPC++ runtime is not able to handle the serialization and transfer of arbitrary object graphs over the network directly. Without additional annotations, only simple classes[3] may be serialized without user intervention. For most[4] other classes, serialization is still possible if the application developer provides the runtime with hints on how to serialize an object (Bachan, 2019). In the case of my matrix class (depicted in Figure 11.5), it is enough to provide the runtime with a list of fields to be serialized, using the `UPCXX_SERIALIZED_FIELDS(rows, cols, data)` macro. For more complicated classes, use of the macro may not be sufficient. In those cases, the serialization code may be provided manually. UPC++ provides the class template `upcxx::serialization<T>` for the serialization of type `T`. For classes that cannot be serialized using the macro, one can provide an explicit specialization of the template for the type to be serialized. Within that specialization, static methods for the serialization and deserialization of instances of the class need to be implemented. For the `Matrix` class, I implemented a custom serialization `upcxx::serialization<Matrix>`, shown in Figure 11.6. Serialization is performed by first writing the dimensions of the matrix, and then its contents. This allows for the deserializion of the object. First I read the dimensions, then I allocate a buffer of sufficient size and deserialize the matrix data into it. Finally, I can re-create the matrix object in the storage space provided by the UPC++ runtime.

With `Matrix` objects available as tokens, I can implement the `CannonActor` class. Its signature is given in Figure 11.7. The main difference to the ActorX10 version (see Figure 10.2) is that the number of possible tokens for each port is explicit directly in the port's type. Furthermore, the guards and actions are added explicitly as private methods. These methods are called in the actor's `act()` method, given in Figure 11.8. Each state is reflected by one of the cases in the `switch`-statement, and each state transition by one of the nested `if`-statements. The guard

[3] Fulfilling either one of the type traits `std::is_trivially_copyable` or `std::is_trivially_serializable`

[4] Except classes that depend on objects that are not serializable, such as operating system handles, or database instances

```
namespace upcxx {
    template <>
    struct serialization<Matrix> {
        template<typename Reader>
        static Matrix* deserialize(Reader &r, void *storage) {
            size_t cols = r.template read<size_t>();
            size_t rows = r.template read<size_t>();
            std::vector<float> data(rows*cols);
            r.template read_sequence_into<float>((void *)data.data(), rows*cols);
            Matrix *m = new (storage) Matrix();
            m->data = data;
            m->rows = rows;
            m->cols = cols;
            return m;
        }

        template<typename Writer>
        static void serialize(Writer &w, Matrix const &m) {
            w.write(m.rows);
            w.write(m.cols);
            w.write_sequence(m.data.begin(), m.data.end());
        }
    };
}
```

Figure 11.6.: Actor-UPC++ custom `Matrix` class serialization.

```
class CannonActor : public Actor {
    private:
        InPort<Matrix, 4> *right;
        InPort<Matrix, 4> *down;
        OutPort<Matrix, 4> *left;
        OutPort<Matrix, 4> *up;
        Matrix result;
        CannonActorState currentState;
        // Omitted: attributes i, j, size, numOperations, operationsPerformed

    public:
        CannonActor(size_t i, size_t j, size_t size, size_t numOperations);
        void act();
        void placeInitialTokens();
    private:
        void performPartialComputation();
        void performShutdown();
        bool mayRead();
        bool mayWrite();
};
```

Figure 11.7.: Actor-UPC++ Class signature of the `CannonActor` class.

function `mayRead()` checks if there is at least one free token available in each of the channels connected to ports `right` and `down`, and `mayRead()` checks if there is sufficient space to write at least one token in the channels connected to ports `left` and `up`.

The setup of the actor-based computation is performed by the `CannonActorGraph` class. In contrast to the ActorX10 version, there is no need to inherit from the `ActorGraph` class. Initialization is performed in three steps: first, all actors are created, then they are connected, and finally, the initial tokens are written into the channels. Unlike in the X10 version, the actors are created directly on the destination rank using the local instance of the actor graph. The precise mapping of the actor to a rank is left to the application developer. In my case, I chose an approximate Block-Block-Distribution. In between the three steps of the computation's setup, a synchronization step is necessary to ensure that the previous step has been completed on all ranks. The code for the setup is depicted in Figure 11.9. To determine the actors that need to be created on the local instance, I added a method `forallLocalActors(std::function<void(size_t,size_t)>)` that executes a given function on all the local actor coordinates. Once the creation of the graph is finished, the computation simply needs to be started on all ranks.

```
1  void CannonActor::act() {
2      switch (currentState) {
3          case CannonActorState::COMPUTE:
4              if (operationsPerformed < numOperations && mayRead() && mayWrite()) {
5                  performPartialComputation();
6              } else if (operationsPerformed == numOperations) {
7                  performShutdown();
8              }
9              break;
10         case CannonActorState::FINISHED:
11             break;
12         default:
13             abort();
14             break;
15     }
16 }
```

Figure 11.8.: Actor-UPC++-FSM for the `CannonActor` class.

```
CannonActorGraph::CannonActorGraph(size_t n, size_t p) : n(n), p(p) {
    upcxx::init();

    // Create all local actors
    forallLocalActors([&](size_t x, size_t y) {
        CannonActor *ca = new CannonActor(x,y,n/p, p);
        localActors.push_back(ca);
        graph.addActor(ca);
    });

    upcxx::barrier();

    // Connect the ports
    forallLocalActors([&](size_t x, size_t y) {
        auto yTop = (y + p - 1) % p;
        auto xLeft = ( x + p - 1) % p;
        GlobalActorRef a = graph.getActor("Cannon_"s + std::to_string(xLeft) +
        ↪  "_"s + std::to_string(y));
        GlobalActorRef leftActor = graph.getActor("Cannon_"s
        ↪  +std::to_string(xLeft) + "_"s + std::to_string(y));
        GlobalActorRef topActor = graph.getActor("Cannon_"s + std::to_string(x) +
        ↪  "_"s + std::to_string(yTop));
        graph.connectPorts(a, "L", leftActor, "R");
        graph.connectPorts(a, "U", topActor, "D");
    });

    upcxx::barrier();

    // Place initial tokens
    for (auto ca : localActors) {
        ca->placeInitialTokens();
    }
}
```

Figure 11.9.: Actor-UPC++ actor graph construction form in the constructor of the `CannonActorGraph` class.

12. An Actor Library for MPI

Actor-UPC++ provides application developers with the opportunity to use existing tools and libraries while still being able to make use of the actor-based computational model. However, in some cases, the use of UPC++ may not be possible[1]. In his master's thesis, Macedo Miguel implemented an actor library using MPI (Macedo Miguel, 2019). Actor-MPI closely follows the approach taken by Actor-UPC++, but uses MPI for inter-rank communication. For the execution model, the library uses the task-based execution model introduced in section 11.2. For the MPI communication, two different approaches were implemented. The first one uses two-sided MPI operations, and the second one one-sided ones. In both cases, all communication operations are performed by the master thread of the rank.

For the point-to-point transfer, the master thread on the incoming port's side posts non-blocking receive requests, stores the handles in an array, and periodically checks if they have been fulfilled. When a token has been received, the actor is notified, and may use the token. Once the token is read, a new request is posted. On the side of the sender, a non-blocking, synchronous send operation is used. First, the token is copied to an output buffer by the outgoing port. The master thread periodically checks for changes in the buffers, and posts communication operations for the tokens that have been places in its buffer. The send operation is only finished once the receive operation on the incoming port has completed. Thus, the outgoing port knows when to update the capacity.

The one-sided transfer uses a similar mechanism. Initially, each channel is initiated by allocating and exposing a memory segment sufficiently sized to hold the maximum number of its tokens. On the sending side, a buffer is allocated that is used by the master thread of the rank as a source for the RMA operation. When a token is to be written, the sender looks up the appropriate spot and performs the remote access. Once the data is written, it sends a zero-byte message to the receiving rank to inform it of the newly arrived token. When a token is read by the receiving port, the sender is notified the same way. This allows both sides to maintain a coherent view of the channel.

In experimental tests using a shallow water proxy application, the version using the point-to-point operations performed better than the one using the one-sided communication primitives. Reasons for that may be the additional point-to-point messages that are still required to notify the remote ranks in the one-sided version. There may also have been less optimization effort expended by the MPI library implementation.

---

[1] At the time of writing, the low-level GASNet-EX runtime does not yet support the Intel OmniPath Interconnection Fabric, for example.

13. Discussion and Outlook

In this part of the thesis, I presented the actor model as a computational model for parallel and distributed computational tasks. After presenting the FunState actor model in chapter 9, I introduced three implementations of the actor model, two (ActorX10 and Actor-UPC++) in detail, and one (Actor-MPI by Macedo Miguel (2019)) briefly.

The actor model is a computational model that may be used to describe parallel applications. In itself, it prescribes neither a specific interface, a communication technology, or a process model. It does, however, describe the rules for the interaction of different, concurrently running parts of an application. The model describes the rules for communication (ports and channels), and behavior of the individual components of the application (actors and FSMs). This presents a sharp contrast to MPI, the prevalent standard for distributed applications. The standard describes a specific interface for inter-process communication. However, it does not prescribe a specific communication mode, nor a process model. An MPI rank may be sequential or concurrent, and the specification of a rank's behavior is entirely left to the application developer. However, there is a one-to-one mapping to an operating system process. In essence, the MPI model seeks to provide a computational model of the hardware, to be used by the application developer, whereas the actor model provides a way to formalize parallel applications. This formal model may then be mapped to hardware through a concrete library implementation that uses, e.g., MPI. On a practical note, this means that the application developer of an actor-based application does not need to know whether communication is performed locally or globally. Compared to MPI ranks, actors are less tightly coupled, as the channels act as a buffer for communication, and there is no need to perform handshakes or synchronization. They are not necessarily mapped directly to operating system resources, as shown with the different actor execution strategies of Actor-UPC++. Furthermore, they are represented explicitly as an object, and with a suitable library implementation, it is possible to move actors across node boundaries.

ActorX10, the first library described in this part of the thesis, was implemented in a collaboration for use in the InvasIC project. The main requirement for the actor library was to be able to implement programs that may be executed with little porting effort on the invasive prototype platform as well as on HPC architectures. Using the X10 language as a platform, we achieved that goal. The main benefit of X10 is the global object model. Using X10, one may simply move arbitrary object graphs around without manually specifying their serialization or the mode of transportation over the interconnection network. This allowed us to implement the migration of actors in ActorX10 in a way that is transparent to the application developer. The X10 serialization is powerful enough to capture and serialize any X10 object that is connected to the actor someway, and to reconstruct it on the receiving place. For actor migration to work in Actor-UPC++, one has to specify the serialization for the actor subclass and all the types of its instance attributes. This

cannot be done directly in the library, as these types are not yet known, therefore the application developer needs to provide this information.

However, there are downsides to X10 as a language. Unfortunately, it never managed to gather a large community of active users, and therefore remained mainly a research language. One of the consequences is the lack of third-party X10 libraries for HPC. This can be somewhat mitigated using the possibilities for interfacing with C++ code, but when it is done, the advantage of universal object serialization is lost. Instead, the X10 object interfacing with the C++ code needs to provide manual serialization. In our project, we use an older version of the language, which is incompatible with the current X10 version. Newer versions of the language changed the integer data type to a length of 64 bit (Saraswat et al., 2019) to increase the maximum number of places within a computation, and to utilize the default register length on modern HPC processors. However, the LEON 3 core used in the invasive platform has a register size of only 32 Bit. Using 64 Bit integers is possible, but operations on them are not natively supported by the processor and are therefore much slower. To avoid this performance bottleneck, the X10 compiler for the prototype platform remained on the version 2.3.1 of the X10 language, the last language standard to use 32 Bit integers as a default. Aside from incompatibilities with current software environments, there is no support for current HPC interconnection technologies, the only interconnection technology supported is IBM's PAMI interconnect that was used on the, now obsolete, BlueGene systems. For all other systems, the MPI connection backend has to be used. This adds another layer of indirection, and therefore generates additional overhead. Finally, X10's model for parallelism sacrifices expressiveness for simplicity and deadlock-freedom. For instance, without the help of synchronization objects such as mutexes and semaphores, one cannot express the dependencies between different invocations of the `act()` method within the same actor.

The environment used in Actor-UPC++ is quite different: Unlike X10, C++ is widely used in the field of HPC, and therefore tools and third-party libraries are readily available. C++ was not initially designed as a parallel language, and therefore there is no prescribed mode for parallel computations, and, furthermore, the programming language is not aware of the existence of distributed memory environments at all. Instead, this functionality is added using third-party libraries. This allowed me to pick and match the technologies suitable for implementing the actor model. UPC++ was a good fit thanks to its one-sided communication operations, and OpenMP was chosen due to its straightforward approach to tasks and its wide support in modern compilers. The flexibility gained this way enabled me to experiment with different ways to parallelize actor execution through different execution models.

The actor graph of the current version of Actor-UPC++ is static, which hampers load balancing, as actors are bound to the rank they were created on. Unlike in ActorX10, migration of actors may not be done transparently to the application developer, as C++ code is not aware of the structure of object graphs. Therefore, it is necessary to implement a custom migration for each class that is to be serialized, and to provide code that reassembles the object on the target rank. In the thesis of Budanaz (2020), a version of Actor-UPC++ with support for actor migration was implemented. The implementation periodically interrupts the actor execution in order to move actors between ranks to equalize the computational load. Currently, migration of actors involves a significant overhead. In tests with the shallow water proxy application (Budanaz, 2020), actor migration proved to be impractical. Experiments with the Charm++ runtime system and SWE suggest that patch migration is not useful with only minor load imbalances, such as the ones caused through

local time stepping. Further research should be done here to investigate and mitigate potential performance bottlenecks.

Actor-UPC++ may be viewed as the first step towards a larger framework for actor-based parallel computing. The vision would be to express the individual building blocks of an application in terms of actors. The actor framework would then take care of distributing the computation onto the available compute resources. Such a framework should offer a range of different communication backends. Depending on the desired characteristics of the actor execution and the underlying hardware, different modes of actor parallelization, and different communication runtimes might be used. With Actor-MPI (Macedo Miguel, 2019), we demonstrated that using a lower-level communication library is possible. It may thus be considered another step in that direction. The current implementation still exposes some characteristics of the communication backend, especially during object initialization. Furthermore, a generic way for the application developer to specify serialization of actors and tokens is still missing.

Once the base framework is in place, there are several ideas that may be worth further exploring: *Dynamic actor graphs* are useful to support more unpredictable application scenarios. With the current, static graphs, all different configurations need to be known at the time of the actor graph initialization. Dynamic behavior is possible by sending data conditionally to different actors. However, in some cases, it may be better to completely replace an actor by another, or to only create and connect actors when they are actually needed. In this case, the actor could be created only once it is needed, and in the place that enables the best possible performance, based on the overall system state and the communication requirements of the actor.

Fault tolerant actor execution Fault tolerant execution is expected to become an important concern as we approach the exascale era (Yang et al., 2012). The actor model offers some opportunities to address this through the more explicit communication structure of an actor-based application. First, it may be interesting to implement a form of journaling to the communication channels. Non-volatile memory may be an interesting ingredient here, as its performance exceeds the one of other persistent storage types (Patil et al., 2019). The communication channels could be extended to log all incoming tokens since the last checkpoint of an application. If an actor crashes, one would then only have to recreate the last checkpoint, and insert all tokens logged since then again into the channel. This would allow the actor to reconstruct its last state before crashing without affecting the global state of the computation. Another option would be the replication of actors on different ranks. An actor and its clone would be connected to its neighbors through a single channel that would deliver the same message to both actors. Their behavior would be identical, and therefore, they would produce the same results. Potentially, the clone could use a version of the code that is less computationally complex, and only provides an approximation of the result. Once both the original and the clone insert their result into a communication channel, the channel could act as an arbitrator, and decide whether the input is similar enough to accept it. Optionally it could involve a general check for plausibility (e.g. negative water height in a tsunami simulation).

Formalized State Machine with heterogeneous Actions Currently, the actor FSM is implemented by the application developer using a switch statement. The library is only aware of which actor is triggered, but not of the actions invoked, or even if a state machine transition is possible. It would, therefore, be interesting to encode the state machine of the actor in a form that is understandable

to the actor library. The library could then use that knowledge, e.g. to only create tasks for actors that are actually able to perform a state transition. Alternatively, it may also be possible to enable the application developer to provide actions targeting heterogeneous devices such as GPUs. Modern GPU frameworks offer support for asynchronous kernel execution. The actor graph runtime could enqueue actions targeting GPUs asynchronously, and use the GPU and the CPU concurrently.

Part III.

Tsunami Simulation

14. Tsunami Modelling Using the Shallow Water Equations

I chose the problem of modelling tsunami wave propagation to demonstrate the use of the actor model in the domain of scientific computing. Tsunamis (Japanese for Harbor Wave) are gravitational waves usually caused by large-scale water displacement, e.g. through shifts in the ocean floor that occur in earthquake events. These shifts may release large amounts of energy. For example, the upwards movement of the water column caused by an upwards shift of a tectonic plate yields:

$$E_{\text{pot}} = \frac{1}{2}\rho g \iint \eta^2(x, y)\,\mathrm{d}x\,\mathrm{d}y.$$

This energy is converted into kinetic energy as the displaced column collapses. For instance, take a tectonic event similar to the Aceh-Adaman earthquake. Assuming the gravity $g = 9.81\frac{m}{s^2}$, and the density of water $\rho = 997\frac{kg}{m^3}$, a displacement area of $\approx 225000km^2$ and an upwards displacement of 5m, the potential energy would be $\frac{1}{2} \cdot 225 \cdot 10^9 m^2 \cdot 997\frac{kg}{m^3} \cdot 9.81\frac{m}{s^2} = 1.1 \cdot 10^{18}$J. This would be equivalent to the energy released by exploding 263000 kilotons of TNT[1](LeVeque, George, and Berger, 2011; Tang et al., 2012).

Tsunamis may also be caused by landslides (both underwater and into the water), and occur both at sea and in lakes. In all cases, they are caused by large displacements of the entire water column. During wave propagation on the open water, tsunami wave lengths are significantly larger than the height of the water column, and one may therefore view the water as shallow, and ignore any effects on the vertical axis (LeVeque, George, and Berger, 2011). This allows the use of the two-dimensional shallow water equations to model the propagation of tsunami waves.

In this chapter, I will discuss the model (see section 14.1) and the computation of approximate solutions (see section 14.2). The model is implemented in the actor-based tsunami applications presented in this thesis. It follows the model of LeVeque, George, and Berger (2011). It is based on the SWE implementation by Breuer and Bader (2012).

[1] Or about five times the yield of the largest man-made explosion, the Tsar Bomba

14.1. The Two-dimensional Shallow Water Equations

The one-dimensional case of the shallow water equations with a bathymetry source term is given as a system of partial differential equtions (PDEs)

$$\begin{aligned} (h)_t + (hu)_x &= 0 \\ (hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x &= -gh\,(b)_x\,, \end{aligned} \tag{14.1}$$

where g is the gravitational constant, h is the height of the water column, u the vertically averaged velocity, and b the elevation of the floor, also referred to as *bathymetry*. Bathymetry values greater than zero signify parts of the domain initially above water, while values smaller than zero are initially submerged. The term $\left(\frac{1}{2}gh^2\right)_x$ denotes the force induced by gravity as a result of hydrostatic pressure. Furthermore, hu denotes the momentum, as h is directly proportional to the mass of the water column. A graphical description of the unknowns is given in Figure 14.1.

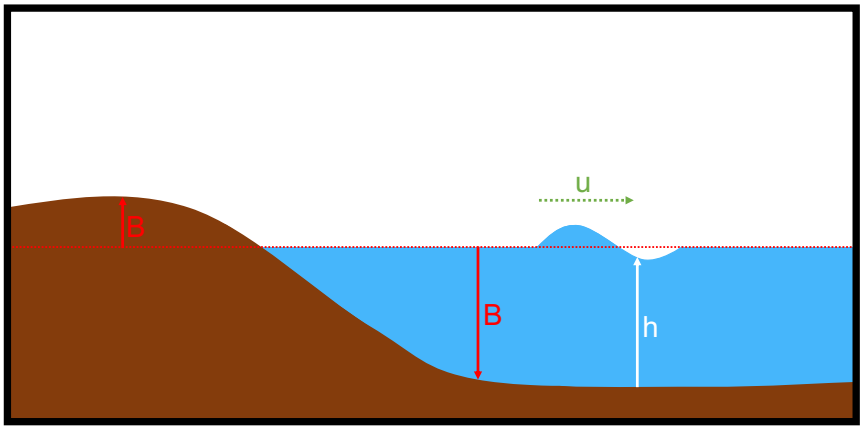


Figure 14.1.: Schematic overview of the shallow water equations' unknowns. The dashed line denotes water surface at lake-at-rest and is used as a reference point for the bathymetry, denoted in red. The water is modelled as the height of the water column on top of the bathymetry (shown in white). Finally, the vertically averaged velocity in the horizontal direction is depicted in green.

One can extend the system towards the two-dimensional case to model tsunamis. One obtains:

$$\begin{aligned} (h)_t + (hu)_x + (hv)_y &= 0 \\ (hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x + (huv)_y &= -gh\,(b)_x \\ (hv)_t + (huv)_x + \left(hv^2 + \frac{1}{2}gh^2\right)_y &= -gh\,(b)_y\,. \end{aligned} \tag{14.2}$$

Again, $h(x, y, t)$ denotes the water height at a given point and time within the domain. Consequently, $u(x, y, t)$ and $v(x, y, t)$ denote the velocities in the x and y directions, respectively.

The gravity forces are now present for the two spatial dimensions, as $\left(\frac{1}{2}gh^2\right)_x$ and $\left(\frac{1}{2}gh^2\right)_y$, furthermore, hu and hv denote the momenta in the two spatial directions. Equation 14.2 may be represented in the canonical form of a conservation law, as:

$$q_t + \big(f(q)\big)_x + \big(g(q)\big)_y = \Psi(q), \tag{14.3}$$

with:

$$q := \begin{pmatrix} h \\ hu \\ hv \end{pmatrix},\ f(q) := \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix},\ g(q) := \begin{pmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{pmatrix},\ \text{and } \Psi(q) := \begin{pmatrix} 0 \\ -gh\,(b)_x \\ -gh\,(b)_y \end{pmatrix}. \tag{14.4}$$

PDEs of this shape are a commonly occurring pattern, and there are well-established methods to approximately solve them (Einfeldt, 1988; Thomas, 1999; Bale et al., 2003; LeVeque, George, and Berger, 2011). However, in the general case, a direct solution is not possible, therefore, a numeric solution scheme is commonly used.

14.1.1. Hyperbolicity

The shallow water equations belong to the class of *hyperbolic* PDEs (LeVeque, George, and Berger, 2011). The Jacobian matrices of hyperbolic PDEs have only real eigenvalues ($\lambda \in \mathbb{R}$), and their eigenvectors are linearly independent. Following Meister (2016), we obtain the following Jacobian matrices for the two-dimensional shallow water equations:

$$q_t + f'(q)q_x + g'(q)q_y = \Psi \tag{14.5}$$

$$f'(q) = \begin{pmatrix} 0 & 1 & 0 \\ -u^2 + gh & 2u & 0 \\ -uv & v & u \end{pmatrix},\quad g'(q) = \begin{pmatrix} 0 & 0 & 1 \\ -uv & v & u \\ -v^2 + gh & 0 & 2v \end{pmatrix}$$

The Eigenvalues of the two matrices:

$$\lambda_f^1 = u - \sqrt{gh},\ \lambda_f^2 = u,\ \lambda_f^3 = u + \sqrt{gh},\ \lambda_g^1 = v - \sqrt{gh},\ \lambda_g^2 = v \text{ and } \lambda_g^3 = v + \sqrt{gh} \tag{14.6}$$

are real as long as $h > 0$. The corresponding eigenvectors:

$$v_f^1 = \begin{pmatrix} \frac{1}{v} \\ \frac{u-\sqrt{gh}}{v} \\ 1 \end{pmatrix},\quad v_f^3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},\quad v_f^2 = \begin{pmatrix} \frac{1}{v} \\ \frac{u+\sqrt{gh}}{v} \\ 1 \end{pmatrix},\tag{14.7}$$

$$v_g^1 = \begin{pmatrix} \frac{1}{v-\sqrt{gh}} \\ \frac{u}{v-\sqrt{gh}} \\ 1 \end{pmatrix},\quad v_g^2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix},\quad v_g^3 = \begin{pmatrix} \frac{1}{v+\sqrt{gh}} \\ \frac{u}{v+\sqrt{gh}} \\ 1 \end{pmatrix}$$

are linearly independent as long as $h > 0$. This condition holds for all physical phenomena according to the definition of the equations above, therefore, the PDEs are hyperbolic for the

value ranges that are to be simulated. Hyperbolicity brings a finite wave propagation speed that is determined by the Eigenvalues of the system. Together with the chosen discretization, discussed in section 14.2, it forms the basis for several implemented optimizations for my tsunami application.

14.2. Finite Volume Discretization

I use a finite volume scheme for the discretization in space, and an explicit Euler scheme for the discretization in time. The approach has been described before in Bader et al. (2020) and is identical to the one used in Breuer and Bader (2012). In contrast to the finite-difference method – which approximates the derivative on certain mesh points – the finite volume scheme averages the values of the unknowns over the entire grid cell. New time steps are computed by evaluating the transport of unknown quantities across cell boundaries (*fluxes*). A significant influence on the quality of the obtained solution scheme is the quality of the flux functions that are used to determine the solutions to these problems. For my application, I rely on different approximate Riemann solvers, e.g. the HLLE solver implemented by Schaffroth (2015) for his master's thesis. In the following, I will give a brief description of the discretization scheme used in my tsunami application.

The unknown vector q is discretized onto a Cartesian grid. Vector $Q_{i,j}^{(n)}$ describes the approximate solution of the integral of the unknown functions in cell (i,j): ($\mathcal{C}_{i,j} = \left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right] \times \left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right]$) at time step n as

$$Q_{i,j}^{(n)} \approx \frac{1}{\Delta x \Delta y} \iint_{\mathcal{C}_{i,j}} q(x,y,t_n)\,\mathrm{d}x\,\mathrm{d}y. \tag{14.8}$$

Here, Δx and Δy refer to the cell sizes in the two spacial directions of cell $\mathcal{C}_{i,j}$. Each cell contains the aforementioned unknown quantities $Q_{i,j} := \left(h_{i,j}, hu_{i,j}, hv_{i,j}, b_{i,j}\right)^T$. Thus, the system for $\mathcal{C}_{i,j}$ is

$$Q_t\Big|_{i,j} + \big(f(Q)\big)_x\Big|_{i,j} + \big(g(Q)\big)_y\Big|_{i,j} = \Psi(Q)\Big|_{i,j}. \tag{14.9}$$

This equation remains continuous in both space and time. To solve the system approximately, it is possible to compute only certain time steps[2] t_n:

$$Q_t\Big|_{i,j}(t_n) \approx \frac{Q_{i,j}^{(n+1)} - Q_{i,j}^{(n)}}{t_{n+1} - t_n} = \frac{Q_{i,j}^{(n+1)} - Q_{i,j}^{(n)}}{\Delta t} \tag{14.10}$$

Insertion into Equation 14.9 yields an iterative solution scheme (*explicit Euler time stepping*) that is solved by evaluating the fluxes between cell $\mathcal{C}_{i,j}$ and all its neighbors. Thus, one may compute the solution iteratively for each time step, starting with an initial state $Q_{i,j}^{(0)}$. The vector of unknowns

[2] the use of the subscript t_n does not denote a derivative here, but the n^{th} time step

for step $(n+1)$ is computed as

$$Q_{i,j}^{(n+1)} = Q_{i,j}^{(n)} - \frac{\Delta t}{\Delta x}\left(\mathcal{A}^{+}\Delta Q_{i-\frac{1}{2},j}^{(n)} + \mathcal{A}^{-}\Delta Q_{i+\frac{1}{2},j}^{(n)}\right) - \frac{\Delta t}{\Delta y}\left(\mathcal{B}^{+}\Delta Q_{i,j-\frac{1}{2}}^{(n)} - \mathcal{B}^{-}\Delta Q_{i,j+\frac{1}{2}}^{(n)}\right). \tag{14.11}$$

Here, $\mathcal{A}^{+}\Delta Q_{i-\frac{1}{2},j}^{(n)}$ represents the transfer of unknown quantities into $\mathcal{C}_{i,j}$ from its left cell boundary, $\mathcal{A}^{-}\Delta Q_{i+\frac{1}{2},j}^{(n)}$ the one from the right, $\mathcal{B}^{+}\Delta Q_{i,j-\frac{1}{2}}^{(n)}$ the one from the bottom and, finally, $\mathcal{B}^{-}\Delta Q_{i,j+\frac{1}{2}}^{(n)}$ represents the transfer from the top. The notation $\mathcal{A}^{\pm}+\Delta Q$ is derived from *Godunov's method* that may be used in the linear case of the shallow water equations (obtained through omission of the bathymetry). In the one-dimensional case, the flux F between two cells is computed as the matrix vector product

$$\mathcal{A}^{\pm}\Delta Q_{i-\frac{1}{2}} = A^{\pm}\left(Q_{i}^{(n)} - Q_{i-1}^{(n)}\right) \tag{14.12}$$

$$\text{with } A^{\pm} = R\Lambda R^{-1} \text{ and } \Lambda^{\pm} = \begin{pmatrix} \left(\lambda^{1}\right)^{\pm} & 1 \\ 1 & \left(\lambda^{2}\right)^{\pm} \end{pmatrix}.$$

R is the matrix of Eigenvectors of the Jacobian matrix $f'(q)$ of the flux function in the conservative form, and λ^{n} is its n^{th} eigenvalue. Furthermore, $\lambda^{+} = \max(0,\lambda)$ and $\lambda^{-} = \min(0,\lambda)$ denote maximum and minimum-limited eigenvalues.

Cells are typically updated in three passes. In the first pass over the domain, the fluxes between all horizontal neighbors are computed. In the second pass, the fluxes between vertical neighbors are computed. During the first two steps, the greatest wave speed is collected, and used to determine an appropriate Δt. Finally, the unknown cells of the domain are updated in a third pass, and the simulation time is advanced according to the determined Δt. Thus, it is possible to view the computation of fluxes as a one-dimensional problem, and therefore one may use the one-dimensional flux solvers described in section 14.3.

14.2.1. The CFL Condition

The method introduced above only considers the direct neighbors of a cell to compute its state in the next time step, e.g. the value $Q_{i,j}^{(n+1)}$ at time step $n+1$ only depends on the values $Q_{i,j}^{(n)}$, $Q_{i-1,j}^{(n)}$, $Q_{i+1,j}^{(n)}$, $Q_{i,j-1}^{(n)}$, and, $Q_{i,j+1}^{(n)}$ in time step n. For this computation to accurately capture the physical phenomena it aims to simulate, the phenomena must not travel faster than the numerical approximation permits. Figure 14.2 depicts a finite volume grid. Only phenomena in the green cells will be considered for the blue cell's update to the next time step. Therefore, there may not be any information in the physical domain that would propagate from the space occupied by the orange cells into the blue cell. Any such information would be lost, as only the values from the green cells are used to compute the update. This condition, named *Cauchy-Friedrichs-Lewy Condition (CFL condition)*, was recognized by Courant, Friedrichs, and Lewy (1928), who described it as:

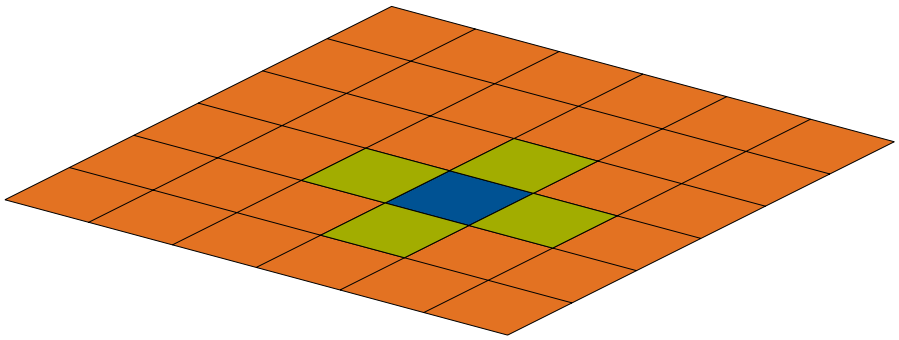


Figure 14.2.: Illustration of the numerical constraints on the size of the time step. The CFL condition requires the time step to be sufficiently small so that for a time step update of the blue cell only information from the green cells is required. Any information transmitted from the orange cells is lost.

> *Während aber beim elliptischen Falle einfache und weitgehend von der Wahl des Gitters unabhängige Konvergenzverhältnisse herrschen, werden wir bei dem Anfangswertproblem hyperbolischer Gleichungen erkennen, daß die Konvergenz allgemein nur dann vorhanden ist, wenn die Verhältnisse der Gittermaschen in verschiedenen Richtungen gewissen Ungleichungen genügen, die durch die Lage der Charakteristiken zum Gitter bestimmt werden. (Courant, Friedrichs, and Lewy (1928))*

The statement is succinctly summarized by LeVeque (2002): *"A numerical method can be convergent only if its numerical domain of dependence contains the true domain of dependence of the PDE, at least in the limit as Δt and Δx go to zero." (LeVeque (2002))*

In the common case, the CFL condition is satisfied by choosing a sufficiently small time step. In the case of hyperbolic PDEs, the time step size is determined by the finite propagation speed of the waves. For solving the shallow water equations, it should be chosen such that the influences from both side of the grid cell do not interact directly (LeVeque, George, and Berger, 2011):

$$\Delta t \leq \nu \frac{\Delta x}{\max_p \lambda^p} \tag{14.13}$$

This holds if the maximum wave speed, $\max_p \lambda^p$, only suffices to traverse half a cell within a time step, i.e. for a *CFL number* $\nu < \frac{1}{2}$.

14.3. Approximate Riemann solvers

In the presence of uneven bathymetry, $\mathcal{A}^{\pm}\Delta Q^{(n)}_{i\pm\frac{1}{2},j}$ and $\mathcal{B}^{\pm}\Delta Q^{(n)}_{i,j\pm\frac{1}{2}}$ may be interpreted as *Riemann Problems*, due to the fact that the unknowns on both sides of the cell boundaries are constant, with a discontinuity in between. The problem consists of the PDE that is to be solved and specific initial data at a given time $t := \bar{t}$ and data that is piece-wise constant, with a single

jump at a given position $\bar{x}$:

$$q(x, \bar{t}) = \begin{cases} Q_l & (x < \bar{x}) \\ Q_r & (x \geq \bar{x}) \end{cases} \tag{14.14}$$

This is precisely the case for the cell boundaries of the grid cells in the finite volume model. When bathymetry is considered, an exact solution of the Riemann Problem is often not possible, hence, approximate Riemann solvers are used. In the tsunami applications I developed, I use three different approximate Riemann solvers: the *f-Wave* solver implemented by (Breuer and Bader, 2012), the augmented Riemann solver implemented by Bader et al. (2014) and the *HLLE* solver implemented by (Schaffroth, 2015). The solvers are developed in the *SWE solver*[3] package at my chair. When possible, I used them directly, where necessary, I ported them to X10.

14.3.1. The f-Wave Solver

The f-Wave Solver is a basic solver for the linearized Riemann problem that nevertheless yields reasonably accurate results for cells away from dry sections of the domain. Its ansatz relies on a matrix A that approximates the derivative $f'(Q)$, and, therefore, satisfies $f(Q_r) - f(Q_l) = A(Q_r - Q_l)$. Matrix A is referred to as the *Roe matrix* (Roe, 1981). It is defined as

$$A := \begin{pmatrix} 0 & 1 \\ -\hat{u}^2 + g\bar{h} & 2\hat{u} \end{pmatrix} \tag{14.15}$$

and uses the arithmetic average $\bar{h}$ and the *Roe average* $\hat{u}$

$$\bar{h} = \frac{1}{2}(h_l + h_r), \; \hat{u} = \frac{\sqrt{h_l}u_l + \sqrt{h_r}u_r}{\sqrt{h_l} + \sqrt{h_r}}. \tag{14.16}$$

Using the corresponding Eigenvalues λ_r^1 and λ_r^2, and Eigenvectors v_1 and v_2:

$$\begin{aligned} \lambda_r^1 &= \hat{u} - \sqrt{g\bar{h}}, \quad v_1 = \begin{pmatrix} 1 \\ \hat{u} - \sqrt{g\bar{h}} \end{pmatrix} \\ \lambda_r^2 &= \hat{u} + \sqrt{g\bar{h}}, \quad v_2 = \begin{pmatrix} 1 \\ \hat{u} + \sqrt{g\bar{h}} \end{pmatrix} \end{aligned} \tag{14.17}$$

the solution is decomposed into *flux waves* (LeVeque, 2002; LeVeque, George, and Berger, 2011; Meister, 2016). The eigenvalues λ_r^1 and λ_r^2 represent the wave speeds of the corresponding waves. Next, $Q_r - Q_l$ is decomposed into two waves:

$$f(Q_r) - f(Q_l) - \Delta x \Psi_{l,r} = \alpha^1_{i-\frac{1}{2}} \cdot v_1 - \alpha^2_{i-\frac{1}{2}} \cdot v_2 \equiv \mathcal{W}^1_{i-\frac{1}{2}} - \mathcal{W}^2_{i-\frac{1}{2}}. \tag{14.18}$$

Direct solution of the system for $\alpha^1_{i-\frac{1}{2}}$ and $\alpha^2_{i-\frac{1}{2}}$ with δ the as right-hand side:

$$\delta = f(Q_r) - f(Q_l) - \Delta x \Psi_{l,r} = \begin{pmatrix} (hu)_r - (hu)_l \\ \left(h_r u_r^2 + \frac{1}{2} g h_r^2\right) - \left(h_l u_l^2 + \frac{1}{2} g h_l^2\right) + \frac{1}{2} g (h_r - h_l)(b_r - b_l) \end{pmatrix}. \tag{14.19}$$

[3] Available on GitHub: `https://github.com/TUM-I5/swe_solvers`

yields:

$$\alpha^1_{i-\frac{1}{2}} = \frac{\left(\hat{u} + \sqrt{g\bar{h}}\right)\delta^1 - \delta^2}{2\sqrt{g\bar{h}}} \quad \text{and} \quad \alpha^2_{i-\frac{1}{2}} = \frac{-\left(\hat{u} - \sqrt{g\bar{h}}\right)\delta^1 + \delta^2}{2\sqrt{g\bar{h}}} \tag{14.20}$$

Finally, one may compute the fluxes

$$\mathcal{A}^-\Delta Q_{i-\frac{1}{2}} = \sum_{w=1}^{2} (\lambda^w_r)^- \mathcal{W}^w_{i-\frac{1}{2}} \text{ and } \mathcal{A}^+\Delta Q_{i-\frac{1}{2}} = \sum_{w=1}^{2} (\lambda^w_r)^+ \mathcal{W}^w_{i-\frac{1}{2}} \tag{14.21}$$

$$\text{where } (\lambda^w_r)^- = \begin{cases} 0 & (\lambda^w_r > 0) \\ \frac{1}{2}\lambda^w_r & (\lambda^w_r = 0) \\ \lambda^w_r & (\lambda^w_r < 0) \end{cases} \text{ and } (\lambda^w_r)^+ = \begin{cases} \lambda^w_r & (\lambda^w_r > 0) \\ \frac{1}{2}\lambda^w_r & (\lambda^w_r = 0) \\ 0 & (\lambda^w_r < 0) \end{cases}. \tag{14.22}$$

The big disadvantage of the solver is that it may return physically unsound results for some input configurations, especially when considering interactions of wet and dry cells. To prevent this from happening, the interaction of wet and dry cells is instead short-circuited to have the dry cell react like a wall, and to reflect the incoming wave. Nevertheless, it performs sufficiently well to make it useful when utilized in conjunctions with more sophisticated solvers to save time when its result is predicted to be sufficiently accurate, or when inundation is irrelevant (Breuer and Bader, 2012).

14.3.2. The HLLE solver

The HLLE solver offers a compromise between the precision of the more computationally complex Augmented Riemann Solver, and the more easily computable, but less precise f-Wave solver. It is based on the work of Harten, Lax, and Leer (1983), with further improvements suggested by Einfeldt (1988). The solver was adapted for use with the SWE software package by Schaffroth (2015) for his thesis. As with the f-Wave solver, the wave composition is computed, but unlike it, the HLLE solver resolves thee waves, and the momentum flux $\phi = hu^2 + \frac{1}{2}gh^2$ is also considered. The system is formulated as:

$$\begin{pmatrix} H_r - H_l \\ HU_r - HU_l \\ \phi\left(Q_r - Q_l\right) \end{pmatrix} = \sum_{w=1}^{3} \alpha^w_{i-\frac{1}{2}} v_w \equiv \sum_{w=1}^{3} \mathcal{W}_p. \tag{14.23}$$

For the Eigenvectors, the so-called *Einfeldt speeds* $\breve{s}^{\pm}$ are used. Using the Eigenvalues of the Jacobian of the quasi-linear shallow water equations:

$$f'(q) = \begin{pmatrix} 0 & 1 \\ gh - u^2 & 2u \end{pmatrix}, \lambda^1_q = u - \sqrt{gh} \text{ and } \lambda^2_q = u + \sqrt{gh} \tag{14.24}$$

and the Eigenvalues of the Roe matrix (Equation 14.15), λ^1_r and λ^2_r are defined as the minimum (for $\breve{s}^-$) and the maximum (for $\breve{s}^+$):

$$\breve{s}^- := \min\left(\lambda^1_r, \lambda^1_q\right) \text{ and } \breve{s}^+ := \max\left(\lambda^2_r, \lambda^2_q\right). \tag{14.25}$$

This is used to define the vectors v_w as:

$$v_1 = \begin{pmatrix} 0 \\ \breve{s}^- \\ \left(\breve{s}^-\right)^2 \end{pmatrix},\ v_2 = \begin{pmatrix} 0 \\ \breve{s}^+ \\ \left(\breve{s}^+\right)^2 \end{pmatrix} \text{ and } v_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \tag{14.26}$$

along with the wave speeds $s_1 = \breve{s}^-$, $s_2 = \breve{s}^+$ and $s_3 = \frac{1}{2}\left(\breve{s}^- + \breve{s}^+\right)$. Using these, and the approximate HLLE middle height[4] set as:

$$h_\epsilon^* = \frac{(hu)_l - (hu)_r + \breve{s}^+ h_r - \breve{s}^- H_l}{\breve{s}^+ - \breve{s}^-}, \tag{14.27}$$

a steady wave v_0 may be computed using $\Delta b = b_r - b_l$ such that:

$$\begin{aligned} v_0^1 &= \min\left(\max\left(-\Delta b, \left\lfloor v_0^1 \right\rfloor\right), \left\lceil v_0^1 \right\rceil\right) \\ v_0^2 &= \min\left(\max\left(-\frac{1}{2}\left(h_r + h_l\right)\Delta b, \left\lfloor v_0^2 \right\rfloor\right), \left\lceil v_0^2 \right\rceil\right) \end{aligned} \tag{14.28}$$

given the lower and upper bounds:

$$\begin{aligned} \left\lfloor v_0^1 \right\rfloor &= \begin{cases} h_\epsilon^* \frac{s_2 - s_1}{s_1} & (s_1 > 0) \\ -h_l & (s_1 < 0) \end{cases}, \\ \left\lceil v_0^1 \right\rceil &= \begin{cases} h_\epsilon^* \frac{s_2 - s_1}{s_2} & (s_2 > 0) \\ -h_r & (s_2 < 0) \end{cases}, \\ \left\lfloor v_0^2 \right\rfloor &= \min\left(-g h_l \Delta b, -g h_r \Delta b\right) \text{ and} \\ \left\lceil v_0^2 \right\rceil &= \max\left(-g h_l \Delta b, -g h_r \Delta b\right). \end{aligned} \tag{14.29}$$

Now, the linear system:

$$\begin{pmatrix} 1 & 1 & 0 \\ s_1 & s_2 & 0 \\ s_1^2 & s_2^2 & 1 \end{pmatrix} \begin{pmatrix} \alpha^1_{i-\frac{1}{2}} \\ \alpha^2_{i-\frac{1}{2}} \\ \alpha^3_{i-\frac{1}{2}} \end{pmatrix} = \underbrace{\begin{pmatrix} h_r - h_l \\ (hu)_r - (hu)_l \\ \left(h_r u_r^2 + \frac{1}{2} g h_r^2\right) - \left(h_l u_l^2 + \frac{1}{2} g h_l^2\right) \end{pmatrix} - \begin{pmatrix} v_0^1 \\ 0 \\ v_0^2 \end{pmatrix}}_{\delta} \tag{14.30}$$

is solved, yielding:

$$\alpha_1 = \frac{s_2 * \delta_1 - \delta_2}{s_2 - s_1}, \quad \alpha_3 = \frac{-s_1 \delta_1 + \delta_2}{s_2 - s_1} \text{ and } \quad \alpha_2 = \delta_2 - \left(s_1\right)^2 \alpha_1 - \left(s_2\right)^2 \alpha_3. \tag{14.31}$$

[4] Always positive, see Theorem 3.1 of George (2006)

This may be used—as in the f-Wave solver—to compute the fluxes, as a linear combination of the three f-Waves:

$$\mathcal{A}^{-}\Delta Q_{i-\frac{1}{2}} = \sum_{w=1}^{3} (s^w)^{-} \mathcal{Z}_w, \quad \mathcal{A}^{+}\Delta Q_{i-\frac{1}{2}} = \sum_{w=1}^{3} (s^w)^{+} \mathcal{Z}_w \tag{14.32}$$

$$\text{where } \mathcal{Z}_1 = \alpha_1 \begin{pmatrix} v_1^2 \\ v_1^3 \end{pmatrix}, \quad \mathcal{Z}_2 = \alpha_2 \begin{pmatrix} v_2^2 \\ v_2^3 \end{pmatrix}, \quad \mathcal{Z}_3 = \alpha_3 \begin{pmatrix} v_3^2 \\ v_3^3 \end{pmatrix},$$

$$(s^w)^{-} = \begin{cases} 0 & (s^w > 0) \\ \frac{1}{2} s^w & (s^w = 0) \\ \lambda_r^w & (s^w < 0) \end{cases} \text{ and } (s^w)^{+} = \begin{cases} s^w & (s^w > 0) \\ \frac{1}{2} s^w & (s^w = 0) \\ 0 & (s^w < 0) \end{cases}.$$

In contrast to the more simple f-Wave solver, the HLLE solver is able to correctly approximate the interaction of wet and dry cells. However, the HLLE solver needs the top of the water column to be above the top of the dry bathymetry for inundation to occur, in contrast to the more computationally complex Augmented Riemann solver, which is able to correctly resolve inundation based on the wave speed alone (LeVeque, George, and Berger, 2011; Schaffroth, 2015).

14.3.3. Augmented Riemann Solver

The Augmented Riemann solver has been proposed by George (2006, 2008) as a more accurate alternative to the aforementioned solvers. Its basic ansatz is similar to the one of the HLLE solver, but with a more complex formulation of the governing system. The computation of a solution may, depending on the input data, necessitate an iterative step, which increases the difficulty for an efficient solution. The Augmented Riemann solver is supported in both SWE-X10 and Pond, but I did not use it for any of the experiments in the subsequent chapters. It is described in more detail by LeVeque, George, and Berger (2011).

15. SWE—Experiments with Novel Runtime Systems

The model described in chapter 14 may be used to simulate wave propagation. However, the sequential compute performance of modern computer architectures is insufficient to simulate full tsunami scenarios. Instead, computations are typically performed in parallel, using all layers of abstraction available to the target computer architecture. The SWE[1] package is an example for a proxy application that implements the aforementioned model using distributed-memory, shared-memory, and instruction-level parallelism (Breuer and Bader, 2012; Bader et al., 2014). It was originally written as a teaching code. The original version of the code strictly follows the BSP approach. It uses a patch-based decomposition of the simulation grid, and implements an iteration scheme with global time stepping, as described in section 15.1. Each processing element is assigned one patch. MPI is used for the communication between the patches. The code was later extended to support hybrid parallelism with MPI and OpenMP, and to support NVIDIA GPUs with CUDA.

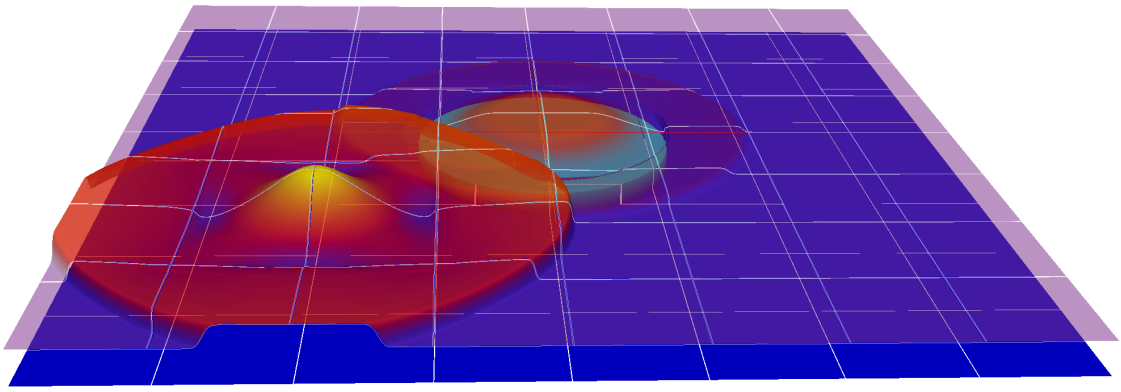


Figure 15.1.: Sample scenario simulated using SWE-PPM and local time stepping

The relatively compact size of the code base made SWE an ideal basis for my actor-based tsunami applications Pond and SWE-X10. Both applications take the patch concept of SWE. In Pond, I was able to reuse significant parts of the application directly, while in SWE-X10, I ported them to X10 and used them as a basis for further extensions. In both cases, it was possible to use the sequential parts of the application, such as the classes pertaining to the patches, or disk I/O. I then implemented the parallel aspects of the applications using the respective actor libraries. This lead to the idea to evaluate other emerging runtime systems and communication libraries using the SWE package. Thus, we collaboratively created SWE-PPM (Bogusz et al., 2020). The application suite implements a distributed tsunami simulation using different programming models, parallel runtime systems and libraries. It currently consists of a pure BSP-based MPI

[1] GitHub: `https://github.com/TUM-I5/SWE`

version based on the original SWE, a fork-join-based version using MPI and OpenMP, a UPC++ version, a Charm++ version and task-based variants based on Chameleon[2] and HPX. The initial versions were implemented in my preliminary work[3] and the bachelor theses of Olden (2018) and Bogusz (2019). Compared to these initial versions, we added support for local time stepping, a hybrid MPI and OpenMP version based on over-decomposition, and improved performance. For some frameworks, namely MPI and UPC++, only communication operations are supported, while for the others, namely HPX, Charm++ and Chameleon, it is possible to perform load balancing as well. The latter is useful in the presence of local time stepping, as the time step size depends on the wave speed, and may be different for different parts of the simulation domain, which may lead to load imbalances. In section 15.1, I will give an overview of the common techniques used in both SWE, SWE-PPM and the different actor-based versions used in the following chapters. Thereafter, I briefly discuss the implementation of SWE-PPM with the different frameworks. Finally, in section 15.3, I will compare the performance of the different implementations both with the original global time stepping scheme and with the local time stepping scheme.

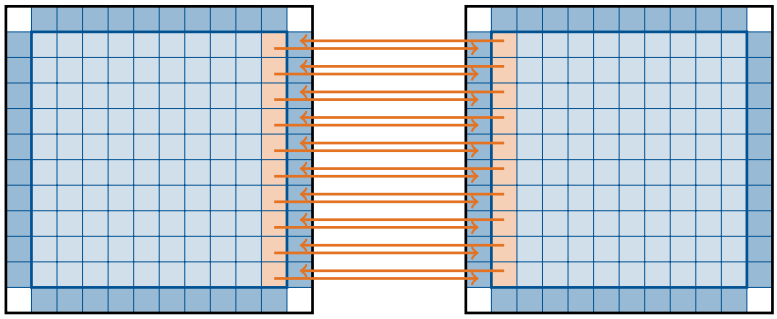
15.1. Patch-Based Tsunami Simulation

The numerical scheme described in chapter 14 only exhibits interactions between neighboring grid cells. To support distributed memory parallelism efficiently, it is therefore important to keep as many interactions between cells within the local memory of a process as possible. This may be achieved by decomposing the grid representing the simulation domain into patches of adjacent grid cells. A patch forms a coherent unit of computation that is always computed within the context of a single process, and the interactions between cells of a patch may be computed without external communication. However, for the cells at the boundaries of a patch, not all interaction partners are available: data from a cell of a neighboring patch is necessary. A common pattern for patch-based domain decomposition is the use of *ghost layers*. Here, an additional row or column of cells is added to the patch for each of its four boundaries. The cells act as a proxy for the adjacent patches. Before computing the updates on the extended patch, the unknown values of the cells have to be filled with data from the adjacent patch. For the scheme discussed here, this pertains to the outermost layers of the original patch, also called *copy layer*. Its contents are copied to the ghost layer of their respective adjacent neighbors, as depicted in Figure 15.2.

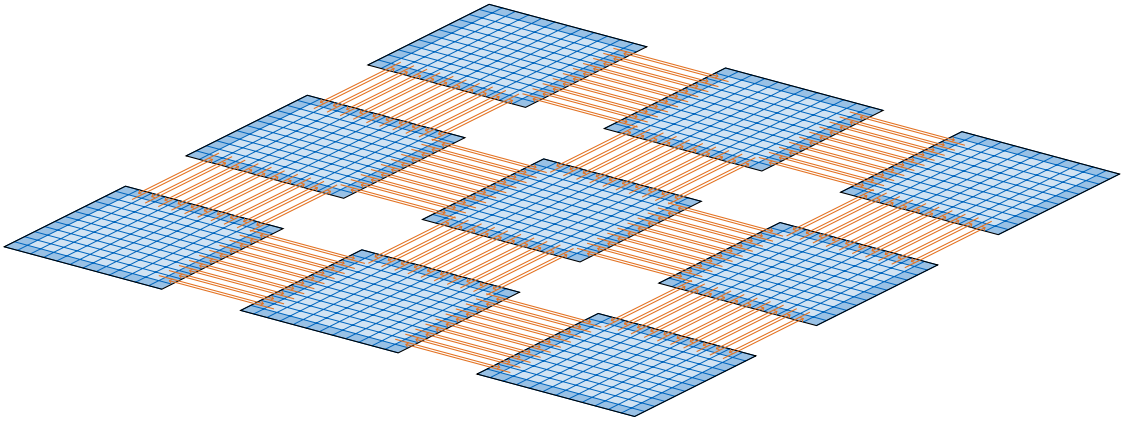
The simulation is performed iteratively, following the scheme outlined in section 14.2. In each iteration, one time step is computed. Here, for each patch, a sequence of steps needs to be performed. First, the patch *sets the ghost layer*. For the ghost layers at the boundaries of the simulation domain, the values are set according to the chosen boundary conditions based on the values of the second outermost layer of cells. For the *outflow* boundary condition, the inner values are simply copied. A *wall* boundary condition can be obtained by inverting the momentum perpendicular to the boundary. If the boundary is adjacent to another patch, the ghost layer needs to obtain values from the second innermost layer of the adjacent patch, as depicted in Figure 15.2. Once completed, the patch *computes the fluxes* between cell boundaries. As a side product of the flux computation between two cells, the solver also determines the maximum wave speed. Based

[2] Not discussed in this thesis.

[3] `https://bitbucket.org/apoeppl/upcxx-swe/src/master/`



(a) Ghost layer exchange between two adjacent patches



(b) Ghost layer exchanges between multiple patches

Figure 15.2.: Ghost layer exchange. Values are copied from the copy layer of a patch to the ghost layer of its neighbor. In Figure 15.2a, the copy layer is depicted in orange, and the copy layer is depicted in darker blue. Figure 15.2b depicts ghost layer exchanges for a simulation with nine patches. The exchange is only performed for the patch boundaries with adjacent neighbors, the other ghost layers are set according to artificial boundary conditions.

on this, the maximum time step for a patch may be computed (see subsection 14.2.1). The patch then has to determine the time step size Δt the simulation time is to be progressed in cooperation with the other patches. Discussed below, there are two strategies to do this. Finally, this time step is used to *compute the unknown values* of the grid cells for the next time step.

Global Time Stepping When global time stepping is used, the application follows the BSP approach. The patches follow the iteration scheme outlined above. As the strategy's name implies, the time step is negotiated globally across all patches. After the flux computation, each patch (i, j) determines its largest locally safe time step $\Delta t_{i,j}$ according to its maximum wave speed and the CFL condition (as defined in subsection 14.2.1). These may then be used to determine the global time step size such that the CFL condition is satisfied for all patches: $\Delta t = \min_{i,j \in \text{Domain}} \Delta t_{i,j}$Typically, this operation is performed as a global reduction, and its result distributed onto all patches. Finally, it is used by all patches to compute the unknowns for the next time step.

Local Time Stepping In many cases, it is not necessary to compute with the same small time step on the entire grid. Some patches may be able to use a larger time step and still satisfy the CFL condition. This would enable a significant savings potential compared to global time stepping, where the time step size is always set to the globally safe (and therefore smallest) increment. If a small time step is dictated by only a small part of the simulation domain, the other patches could use a larger time step, and therefore avoid performing unnecessary computations. A possible solution is *multi-rate local time stepping*. The basic iteration within a patch remains the same. However, the time step is determined locally, based on a *base time step* Δt_{max} that is negotiated in the beginning of the simulation across all patches. Time steps may be set from fixed multiples of the base time step, (e.g. Δt_{max}, $\frac{1}{2}\Delta t_{\text{max}}$, $\frac{1}{4}\Delta t_{\text{max}}$, …). Each patch may then set its time step size for each time interval $\left[k\Delta t_{\text{max}}, (k+1)\Delta t_{\text{max}}\right]$. When a patch is done computing all the fluxes between the cells at an even multiple of Δt_{max}, it uses the maximum wave speed calculated in the process to determine its largest safe time step multiple, and updates its cells using that value. That time step size is then used until the next even multiple of Δt_{max} is reached. However, using that scheme, the different patches no longer always have the same time step, as illustrated in Figure 15.3. Yet, they still depend on receiving fitting data from their neighbors. As depicted in Figure 15.3, a patch may only compute a new step if all its neighbors are at least at the same time step already. As with global time stepping, the values of the ghost layers need to be set at the beginning of the iteration. If the time step of the received values matches the one of the patch, the values may be set directly. Otherwise, they are interpolated linearly based on the values of the previously sent and the current time step. This scheme is based on the *approximate space-time interpolation* proposed by Gudu (2012). The old values are kept until the slower patch catches up to the faster one, and then discarded. If the values received by a patch are from an earlier time step rather than the one the patch is currently one, they are discarded immediately.

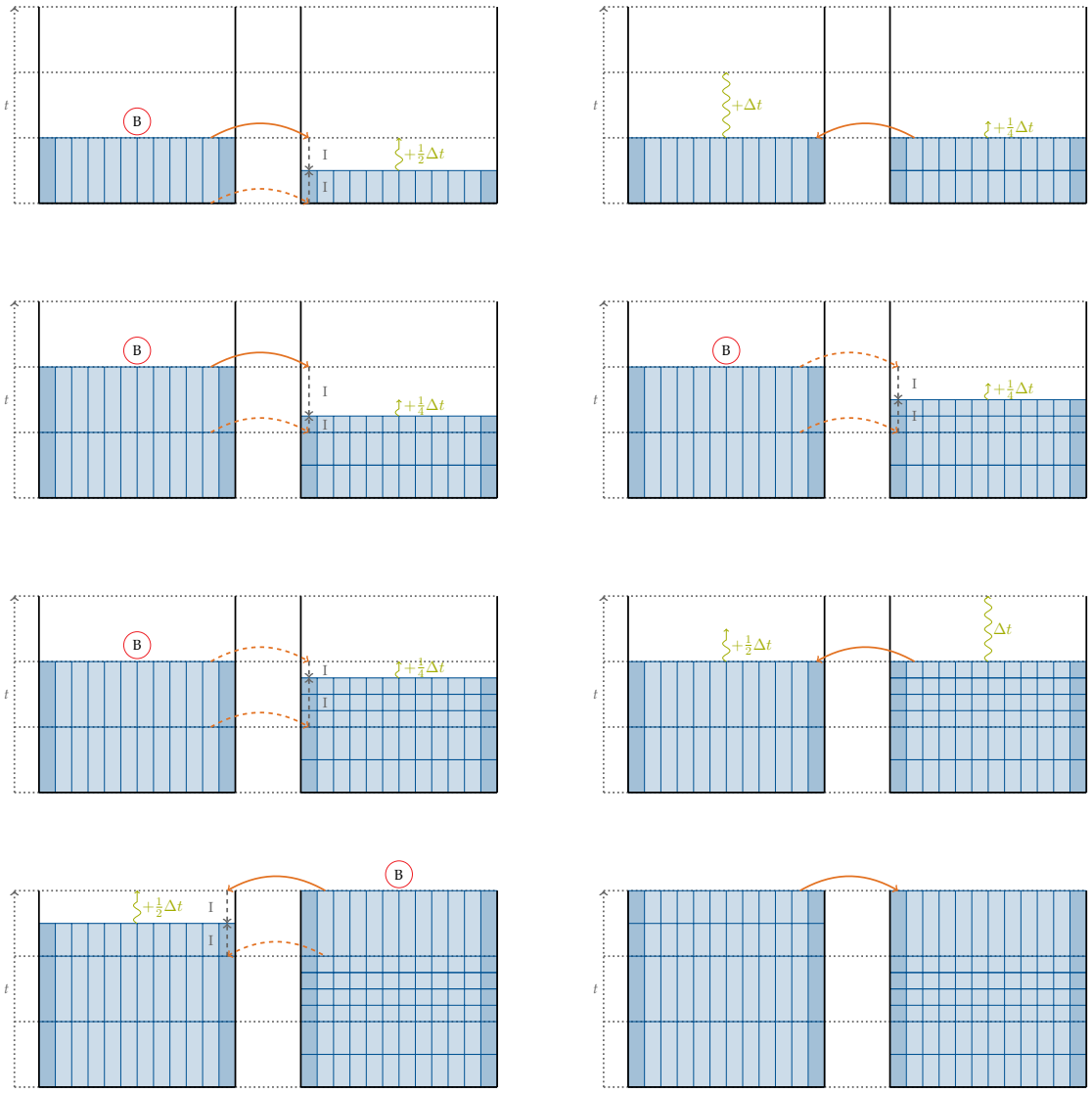


Figure 15.3.: Local Time Stepping Scheme for SWE, schematically depicted for a single patch boundary between two patches at multiple times t. The figure shows the progression of the patch iteration, from top left to bottom right. The horizontal gray dotted lines denote increments of Δt. Whenever a patch computed an update, it sends data to its neighbors. If the neighboring patch is already further on in the simulation, it is not able to use the information, and will discard it. Otherwise, if the time step of the received data matches the time of the patch, it may use it to receive an update. If the time step is greater than the patch's time step, the newly received data will be used together with the previously received data to create an interpolated time step. When either of these conditions is fulfilled for all the boundaries of the patch, it may compute the next time step. Within an increment of Δt, a patch may not change its time step size. Otherwise, it has to wait (and is marked with Ⓑ).

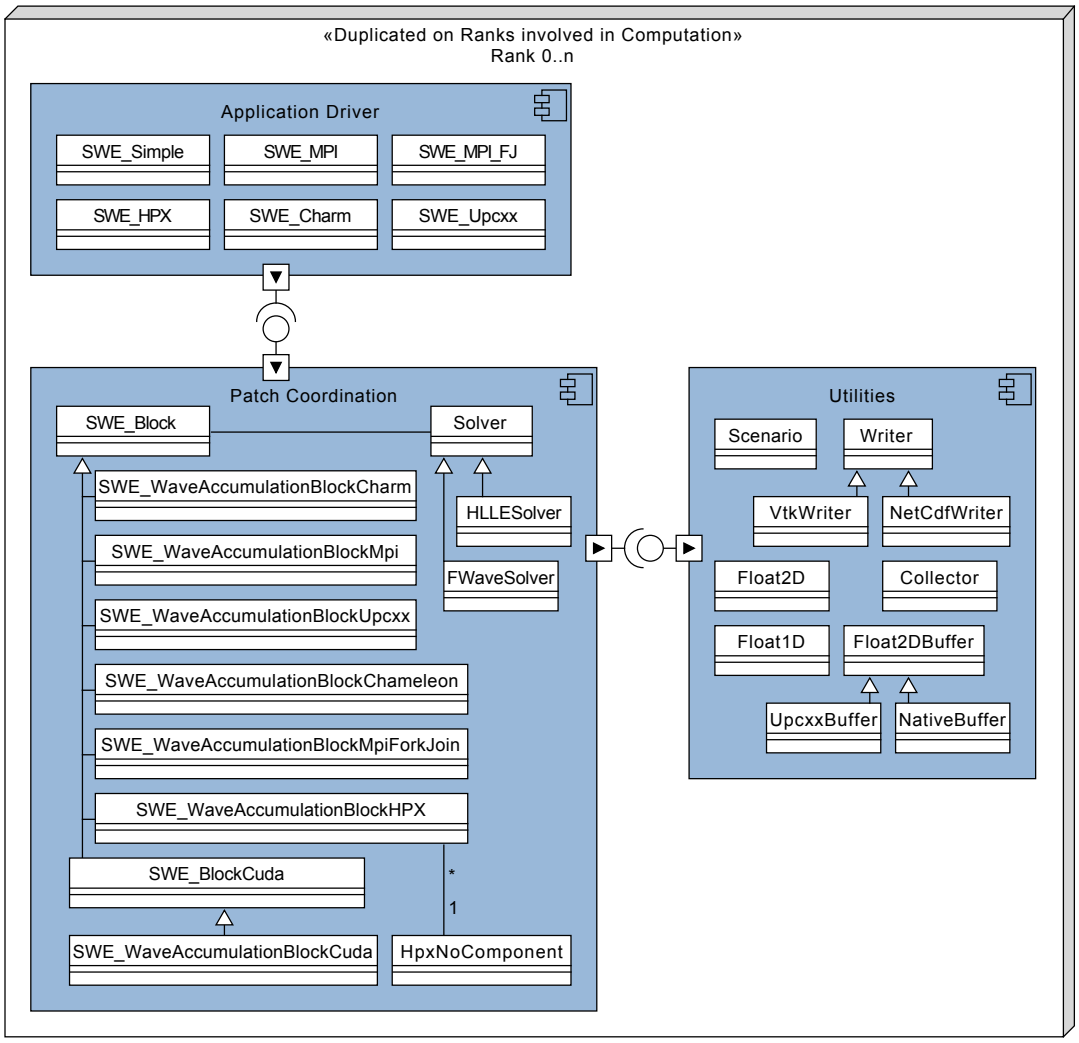


Figure 15.4.: UML Component Diagram for the SWE-PPM Software Package

15.2. Adapting SWE for Different Frameworks

SWE-PPM consists of three major components: *Utilities*, *Patch Coordination* and *Application Drivers*, as depicted in Figure 15.4. The Utilities component provides helper classes for argument parsing, file I/O, scenario handling, and data structures for two-dimensional arrays and array slices. The *Patch Coordination* component handles the patch, its data structures, the patch iteration, and the communication. At the core of the system is the `SWE_Block` class. It provides facilities common to all implementations, such as methods for setting the boundary conditions, the arrays for the unknown quantities or interfaces for the patch iterations. For each of the supported runtime systems—currently UPC++, Charm++, HPX, Chameleon and MPI—there is a corresponding subclass that implements the patch iteration and inter-patch communication according to the guidelines imposed by the parallelization framework. Depending on the framework, the patch iteration is performed sequentially (e.g. UPC++, MPI, MPI +OpenMP, Charm++), or using shared-memory parallelization (e.g. HPX). Finally, there are different application drivers, one for each framework. The drivers start the simulation, collect the parameters, and coordinate the execution of the simulation based on recommended control flow of the parallelization framework. In the following, "our publication" refers to the publication that forms the base for this chapter, Bogusz et al. (2020).

15.2.1. MPI

The MPI version of SWE-PPM uses a two-sided communication approach for the ghost layer exchange. It is based on the original SWE version as described in Breuer and Bader (2012) with extensions for local time stepping (unused) and metadata collection. The ghost layer exchange is performed using non-blocking `MPI_Isend` and `MPI_Irecv` operations. At the beginning of each iteration the application performs send and receive operations for its copy layer and its ghost layer, respectively. For the sending and receiving of the top and bottom boundaries (which are not non-contiguous in memory), an MPI datatype is used to enable a direct copy of the strided data from the buffer. When global time stepping is used, the time step is negotiated using the `MPI_Allreduce` operation. The MPI version of the code is implemented solely using MPI. Each core is assigned a single MPI rank, and each rank computes the updates of a single patch.

15.2.2. MPI and OpenMP

The hybrid MPI and OpenMP variant of SWE-PPM is based on over-decomposition into a number of patches that exceeds the number of processing elements on the rank. We developed this version collaboratively for our publication. In this version, there is a single MPI rank per compute node. Each rank manages a fixed number of patches. Computation of the individual steps in the iteration is parallelized using OpenMP parallel for-loops. Communication between different ranks is performed in a fashion similar to the one described in the pure MPI version, but on a per-patch level. If the received values for a local patch allow for the computation of a new time

step according to the criteria described above, the computation will be performed, otherwise the patch will be omitted, and considered again once new values are received.

15.2.3. UPC++

The UPC++ version of the code follows the same basic approach as the MPI version, but uses one-sided communication operations instead of two-sided ones. It is based on preliminary work performed during my research stay, was modified in the bachelor theses of Olden (2018) and Bogusz (2019) and extended with local time stepping (unused) and metadata collection for our publication. As with the MPI version, there is a one-to-one mapping of patches to processor cores and to UPC++ ranks. The patch data is allocated in the shared segment of each rank to enable a direct access using one-sided communication operations. The ghost layers are exchanged using Remote Memory Access (RMA) operations `upcxx::rget` and `upcxx::rput` for the left and the right ghost layers, and `upcxx::rput_strided` and `upcxx::rget_strided` for the top and bottom ghost layers, respectively. For the one-sided operations to be safely executed, it is important to ensure that the data to be read from or written to is in a state that permits it. This is accomplished using RPCs. Before a ghost layer exchange is started, each patch determines the state of all neighboring patches, and whether they have an update to be sent, or can use the update to be sent by the patch itself. Then, the ghost layer exchange is performed according to the exchanged information. Finally, another RPC is sent to each neighbor to let them know if data was submitted. This scheme avoids the use of a global barrier synchronization. If global time stepping is enabled, the time step computation is performed using `upcxx::reduce_all`.

15.2.4. Charm++

The first version of the Charm++ version of the code has been developed originally by Olden (2018). It has since been improved on by Bogusz (2019). For our publication, we added support for local time stepping and metadata collection. The control model of Charm++ differs from MPI and UPC++ in that there is no more one-to-one mapping of compute resource to patch. Instead, patches are mapped to the basic unit of computation in Charm++, the *chare*. The model resembles the one introduced in chapter 9, and is similar to the one that will be introduced in the forthcoming chapters, chapter 16 and chapter 18. Chares have event-queue-like execution semantics, and may communicate with other chares using specified entry methods. In our solution, we follow this model. There are entry methods in each chare that enable other chares to send ghost layer data, and, as with the other solutions, the chare is only allowed to compute once it has received all necessary data. For global time stepping, each chare sends its own patch-local time step maximum to the main chare. The main chare performs the reduction, and calls the appropriate entry method to register the global time step on all chares. It stores all simulating chares in a chare array. This enables the possibility of load balancing when the chares are sufficiently fine-granular to have multiple chares per processing element. The prospect is especially interesting when local time stepping is enabled, as the different time step sizes lead to different work loads across chares. There are two methods for load balancing available in Charm++, asynchronous and periodically scheduled balancing, and explicit balancing. Both require chares to be serializable,

but for the former version it is necessary that chares can always be transferred without creating inconsistencies. In our case, this is not possible, thus only the second choice for load balancing remains. For load balancing to happen, all chares need to enter a special load balancing state. Once that happens, they and their data may be migrated to different nodes. For this method to be useful, it is important to weigh the gains from balancing against the costs of doing so. In our case this means only performing the load balancing after a number of iterations have passed. There are two different load balancing strategies, *GreedyRefine*, which transfers chares from the most utilized rank to the least utilized one, and *Refine*, which transfers chares away from the most congested rank to average out the load more evenly. In both cases, migration is only performed if the expected benefits outweigh the potential costs of the migration.

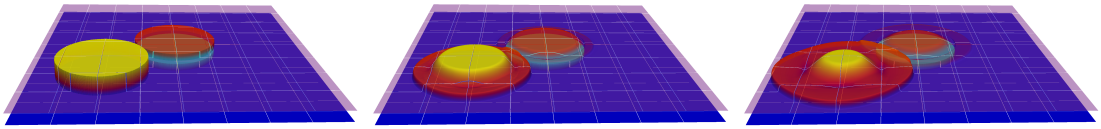
15.2.5. HPX

The HPX implementation of SWE-PPM has originally been developed in the context of the thesis of Bogusz (2019), and as with the other implementations, we added support for metadata collection and local time stepping for our publication. There variants with a single patch or multiple patches per locality. Both variants use one locality per node. Ghost layer exchange and the different stages of the computation are coordinated using asynchronous operations by the master thread of the locality following a fork-join-like pattern of parallelism. If there are multiple patches within the same locality, the individual steps of the iteration are executed concurrently among the patches. The computation of fluxes and computation of the new time step within a patch may optionally be parallelized using `hpx::forall` loop constructs. Communication between patches is implemented using two-sided, explicit communication channels. The time step reduction of in the global time stepping variant is similarly implemented.

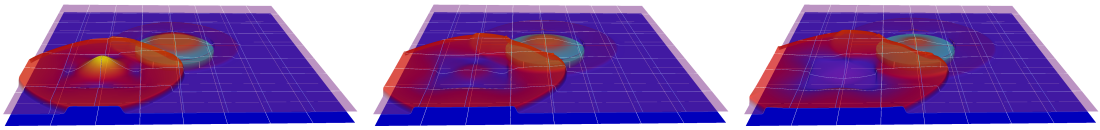
15.3. Evaluation

We evaluated the performance of SWE-PPM and its different implementations on the CoolMUC 2 cluster of the LRZ. CoolMUC 2 features 812 dual-socket nodes and a peak compute performance of 1.4 PFlop/s. Each node is equipped with two Intel Xeon E5-2690v3 “Haswell” CPUs with 14 cores each, and 64 GB of main memory. The nodes are connected using the Mellanox InfiniBand FDR14 interconnection fabric. We used the Intel Development environment featuring the Intel Compiler 2019 and Intel MPI 2019. The code was compiled with optimizations and Advanced Vector Extensions 2 (AVX-2) instructions enabled.

We chose a modified radial dam break scenario for the evaluation with an initial water displacement in the lower left corner of the simulation domain, and an elevation of the ocean floor in the middle of the domain. The scenario and the resulting wave propagation is depicted in Figure 15.5. The flooding of the central islands in the test scenario necessitates a smaller time step size on the affected patches compared to the one used for patches that are on the open water, or at rest. When local time stepping is used, this causes significant load imbalances, as shown in Figure 15.6. Smaller time steps cause comparatively more work than large ones, as a patch with a smaller time step will require multiple small steps to reach the same simulation time as the larger one.



(a) $t = 0\,\mathrm{s}$ (b) $t = 4\,\mathrm{s}$ (c) $t = 8\,\mathrm{s}$



(d) $t = 12\,\mathrm{s}$ (e) $t = 16\,\mathrm{s}$ (f) $t = 20\,\mathrm{s}$

Figure 15.5.: Scenario used for the Evaluation of SWE. In the center of the simulation domain there is a water-covered elevation of the ocean floor. Furthermore, there is an elevation of the water column in the lower left corner of the simulation domain. The water column as well as the water on top of the ocean floor elevation radiate outwards from their initial position as the simulation progresses.

The execution performance is obtained by recording the time from the beginning to the end of the simulation and the number of cell updates that have been performed. This allows us to compute the performance in Flop/s assuming a constant cost of 135 floating-point operations per call to the HLLE solver. Furthermore, we recorded time spent on computation, ghost layer exchange, and time step reduction (for global time stepping only). In the tests, we use the MPI versions as a baseline, and compare it to the aforementioned implementations. For each implementation, we empirically determined the best configuration (see Table 15.1).

15.3.1. Global Time Stepping

First, we evaluated the performance using global time stepping. To this end, we performed a strong scaling test with 8192×8192 grid cells with one to 32 nodes. For the UPC++ and the MPI solution, this led to patches containing about 2.4 million cells per core for the single-node configuration. For the configuration with 32 nodes, each core had a working set of 75 thousand cells. The patches are generated according to a block-block distribution of $n_x^P \times n_y^P$ patches with

$$n_x^P = \max\left(\underset{0<x<\sqrt{r}}{\operatorname{argmin}}\,(r \bmod x)\right) \tag{15.1}$$
$$n_y^P = \frac{r}{n_x^P}.$$

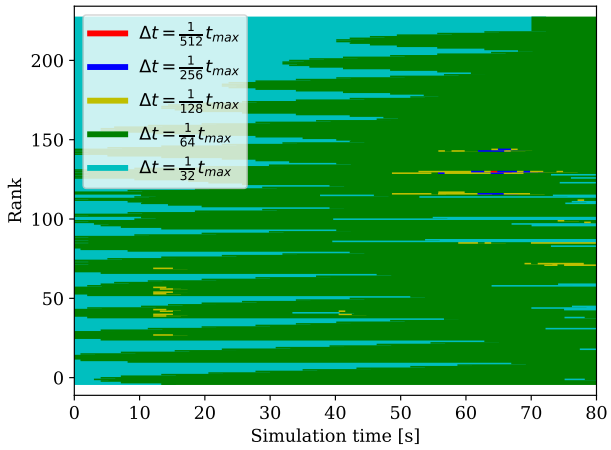


Figure 15.6.: Time step size per rank over the course of the simulation. Each row represents a rank. (Bogusz et al., 2020)

Table 15.1.: SWE-PPM evaluation configurations used in the scaling tests

Execution Type	Symbol	Description
MPI	△	**MPI** implementation. One rank per physical core, 28 ranks per node.
MPI-fj-64	△	**MPI and OpenMP** implementation with 64 Patches per rank. Patches are executed using parallel for-loops.
MPI-fj-128	△	**MPI and OpenMP** implementation with 128 patches per rank. Patches are executed using parallel for-loops.
UPC++	⊗	**UPC++** implementation. One Rank per physical core, 28 ranks per node.
Charm-64	◇	**Charm++** implementation[1] Each node initially contains 64 Chares. Load Balancing is disabled unless specified otherwise.
Charm-128	◇	**Charm++** implementation[1] Each node contains 128 Chares. Load Balancing is disabled.
HPX-1	∗	**HPX** implementation. One HPX locality per node, using `hpx::parallel::for_loop` on each locality
HPX-64	∗	**HPX** implementation. One HPX locality per node, using 64 patches on each locality and task-based Parallelization
HPX-128	∗	**HPX** implementation. One HPX locality per node, using 64 patches on each locality and task-based Parallelization

[1] Configuration with 32 nodes did not run successfully. An error occurred prior to the start of the application during the execution of the Charm++ startup script.

This results in a patch size of $p_x \times p_y$ with

$$p_{x,y} = \begin{cases} \frac{n_{x,y}}{n^P_{x,y}} & \left(q^P_{x,y} < n^P_{x,y} - 1\right) \\ n_{x,y} - \left(\left(n^P_{x,y} - 1\right) \frac{n_{x,y}}{n^P_{x,y}}\right) & \left(q^P_{x,y} = n^P_{x,y} - 1\right) \end{cases}. \tag{15.2}$$

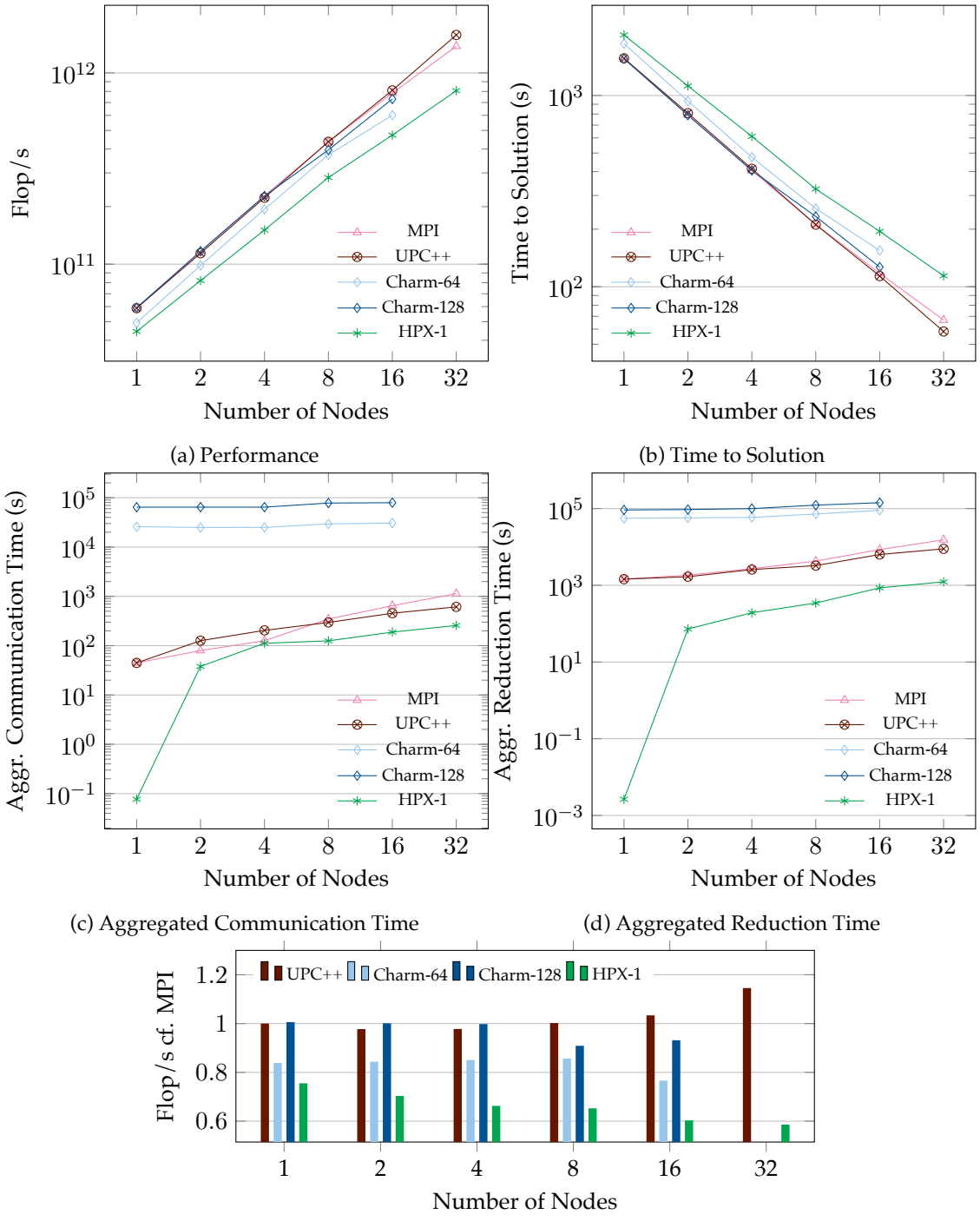
Here, $q^P_{x,y}$ refers to the index of the patch in x or y direction, respectively, and $n_{x,y}$ refers to the number of cells in each dimension. The Charm++ and the HPX variant are configured to take a fixed number of patches per node. Patch size is determined in the same way, but instead of the number of ranks per node, a fixed number of patches is used to split up the domain. Patches are then distributed to the nodes in sequence, i.e. the first 64 patches are distributed onto node 0, the next 64 onto node 1, and so on.

The results of these tests are depicted in Figure 15.7. The UPC++ performs best, along with the MPI and the Charm++ implementation with 128 patches per node. All three implementations perform identically in the runs with up to eight nodes. In the larger configurations UPC++ performs slightly better than MPI, and Charm++ performs slightly worse. HPX performed significantly worse than the other solutions. Depending on the number of nodes, the implementation was 25% to 40% slower. For the UPC++ solution, the better performance may be explained by the slightly better communication and reduction times visible in Figure 15.7c and Figure 15.7d. For the implementations that do not contain an overlapping of communication and computation (MPI, HPX-1 and UPC++), the results depicted there paint an accurate picture of the actual communication time. On the other hand, in the case of Charm++, the result is to be read differently due to the way it was measured. To measure communication, we recorded the time passed between the start of the communication and its termination once all four ghost layers have been received. The over-decomposition used in the Charm++ implementation makes a direct measure of the communication time infeasible, however, as it relies on overlapping communication steps with computation steps to obtain the best possible performance. It is therefore possible for a chare to be preempted if its communication requirements have not been met yet, and for another chare to utilize the computing resource. The timer may only be stopped once the original chare is scheduled again and its communication has concluded.

15.3.2. Local Time Stepping

We also evaluated the performance of our local time stepping solution in another strong scaling [4] test. In contrast to the solution with global time stepping, we only considered implementations that used over-decomposition to generate a solution, i.e. the Charm++ solution and the HPX solution with over-decomposition enabled. The BSP-like structure of the MPI implementation as well as the UPC++ and HPX-1 implementation limits their execution speed to the slowest encountered time step size, as they only use one patch per rank, and therefore do not have the opportunity to balance load locally. Instead of the pure MPI version, we use the hybrid MPI and

[4] This is to be taken with a grain of salt here. As with the classical strong scaling test, we keep the size of the domain constant as we scale up the number of nodes used for the application. However, the actual workload changes, as the patch sizes get smaller as the number of nodes increases. Time steps are set per patch, and therefore the number of cell updates performed is not identical between the different node sizes.



(a) Performance

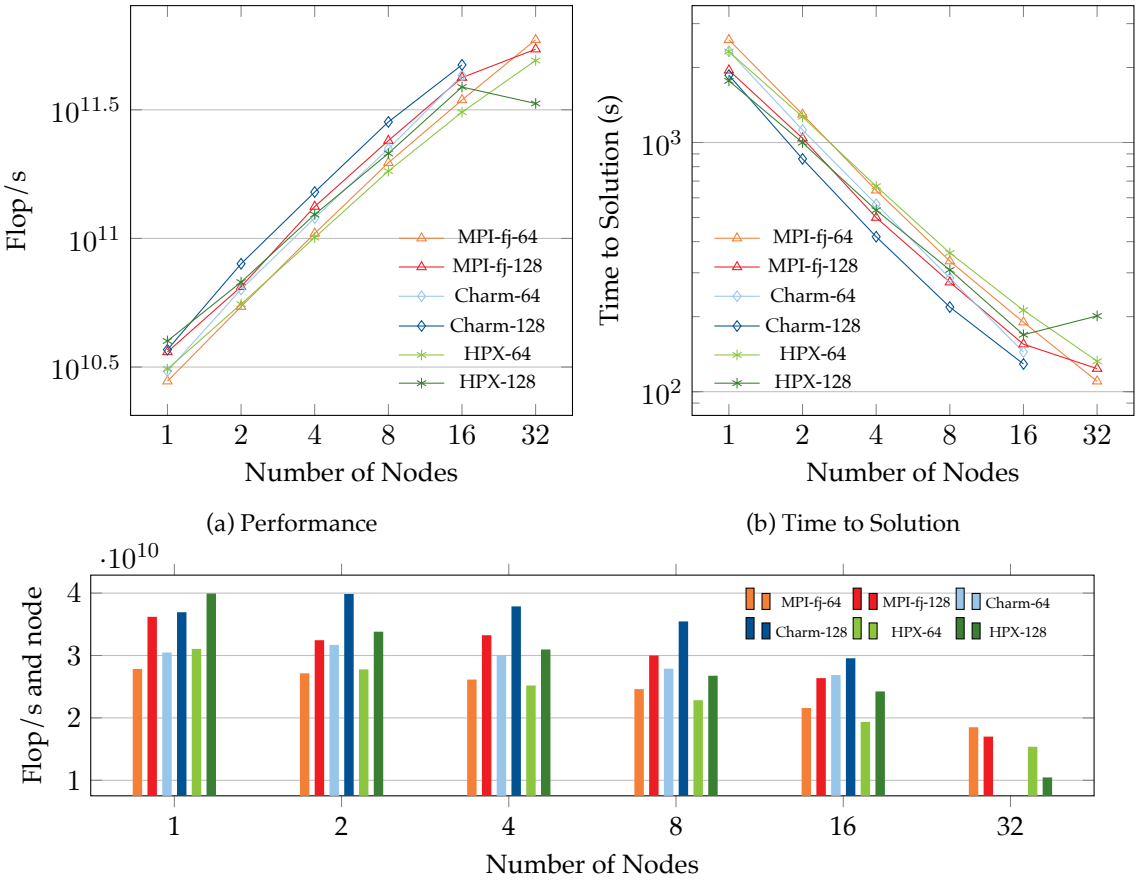
(b) Time to Solution

(c) Aggregated Communication Time

(d) Aggregated Reduction Time

(e) Performance: Detailed Comparison

Figure 15.7.: SWE-PPM Strong Scaling Test with Global Time Stepping. The grid size for the test was fixed at 8192×8192 grid cells. Figure 15.7c and Figure 15.7d depict the aggregated time spent in communication routines by all patches involved in the computation. Result from Bogusz et al. (2020)

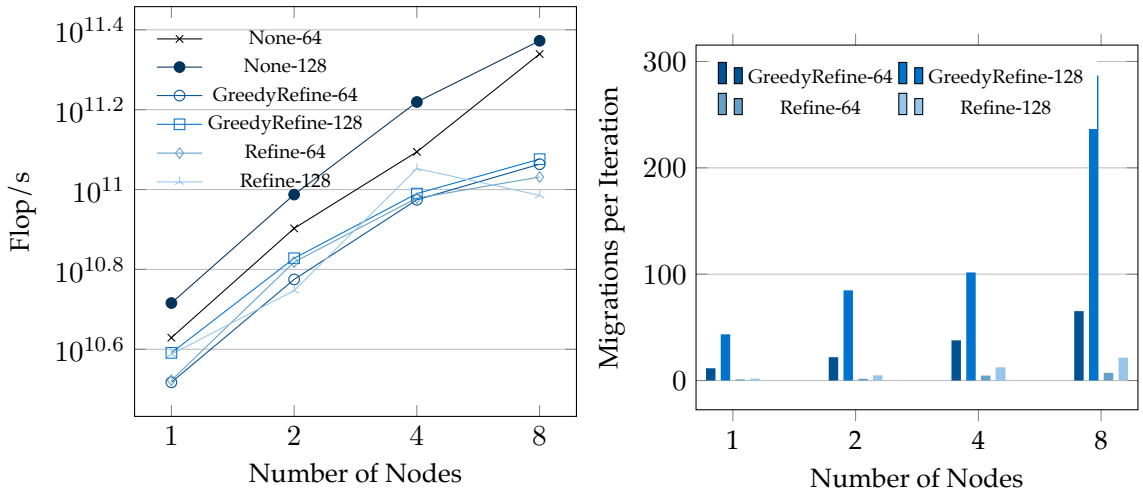


(a) Performance

(b) Time to Solution

(c) Performance: Detailed Comparison

Figure 15.8.: SWE-PPM Strong Scaling Test with local time stepping. The grid size for the test was fixed at 8192×8192 grid cells. Result from Bogusz et al. (2020)



(a) Performance using different inter-node load balancing strategies.

(b) Number of migrations comparison

Figure 15.9.: Performance comparison and comparison of number of chare migrations for different inter-node load balancing strategies for the Charm++ implementation of SWE. The test was performed as a strong scaling test with a fixed grid size of 4096×4096 grid cells and local time stepping. Result from Bogusz et al. (2020)

OpenMP implementation which uses the same over-decomposition as the other implementations. For all implementations, we compared variants with 64 and 128 patches per node. The results are depicted in Figure 15.8. Across the different runtime systems we evaluated, configurations with a higher degree of over-decompositions managed to yield a better performance, except at the scaling limits of the application. Charm++ generally outperformed the HPX solution, as in the previous test, except in the single-node configuration, where HPX performs the same. The hybrid MPI and OpenMP version performs the slowest. For the largest configuration, the HPX and the MPI implementation with 128 nodes are outperformed by the ones using 64 nodes, which suggests that the runtime overheads were too large compared to the computational load.

15.3.3. Detailed Comparison of Over-Decomposition Variations in Charm++ and HPX

In order to select the best configurations to use in the two previous scaling tests, we evaluated different implementation variants of Charm++ and HPX. For the Charm++ implementation we focused on load balancing. Charm++ offers two pre-supplied algorithms for inter-node load balancing: *GreedyRefine* and *Refine*. We compared variations with inter-node load balancing to the baseline solution without inter-node load balancing using over-decompositions with 64 and 128 patches per node and local time stepping (see Figure 15.9a). As before, configurations with a higher degree of over-decomposition perform better, and, notably, the baseline solutions perform better than either load balancing variant. This may be due to the nature of the load imbalance caused by local time stepping. Smaller time steps occur unpredictably, and may change quickly between different iterations. This seems to lead to many chare migrations that may be outdated already at the next iteration. Erratic chare migrations therefore become an obstacle to performance.

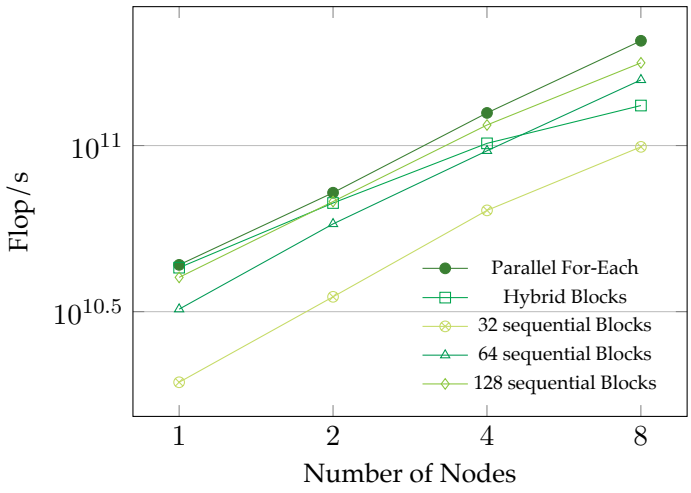


Figure 15.10.: Strong scaling test comparing different implementation strategies for the HPX implementation of SWE. The test was performed as a strong scaling test with a fixed grid size of 4096×4096 grid cells and global time stepping. Result from (Bogusz et al., 2020)

Indeed, the number of chare migrations increases with the number of nodes in the computation, as depicted in Figure 15.9b.

Finally, we compared different variants of the HPX implementation in Figure 15.10. There are two different paths available for parallelization of patches using HPX: one may use parallel algorithms to perform the computation of a single patch in parallel, or one can use tasks to compute multiple sequential patches concurrently. Alternatively, it is also possible to combine the two solutions. In our test the solution with a single patch per node and parallel for-each performed best. For the variant with sequential patches and tasks, the highest degree of over-decomposition performed best. The hybrid solution with 32 patches per node worked well for a low number of nodes, but fell behind successively as the number of nodes in the computation increased. Moreover, increasing the number of patches for the hybrid solution decreased the performance. This may be due to the increase in coordination overhead from the larger number of tasks in combination with the diminishing amount of computational load per patch.

16. SWE-X10, an Actor-Based Tsunami Simulation

SWE, the application introduced in the previous chapter, forms the basis for SWE-X10, my actor-based tsunami application. The BSP-based version works well for the global time stepping variant, as the regular communication structure allows for an easy implementation of the communication, with a direct exchange of the ghost layers. Here, patches are able to copy their data directly into the ghost layers of their neighbors, and it is therefore possible to avoid extra copies. However, this tight coupling of the application domain and the parallelization scheme is not without its disadvantages. The individual blocks need to have knowledge about the internal state of their neighbors, e.g., to determine when the neighbor's buffers are ready to receive updates.

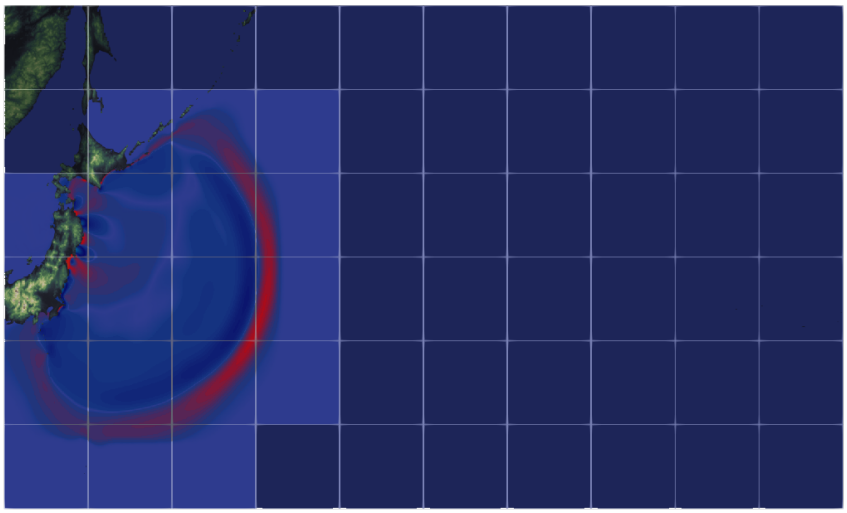


Figure 16.1.: Tohoku tsunami, simulated using SWE-X10. Each square represents a patch being simulated by an actor. The tsunami wave is depicted in red and dark blue. Darker patches are patches that have not been reached by the tsunami wave yet, and are therefore not enabled.

In SWE-X10, I use ActorX10 to decouple the application logic from the parallelization scheme. This facilitates more high-level optimizations, such as *lazy activation*, where actors are only enabled and start computing when they are actually reached by the tsunami wave. Furthermore, the event-queue semantics of ActorX10 enables an overlap of communication and computation without intervention from the application developer, as long as there is a sufficient amount of actors per node. In the remainder of this chapter, I will first discuss the system design of the application in section 16.1, and then move on to the actor-based coordination scheme in section 16.2. Thereafter, I will introduce lazy activation in section 16.3. Finally, I will discuss issues encountered during the implementation of SWE-X10 in section 16.4, and the performance results obtained using ActorX10

on a cluster in section 16.5. The content of this chapter has been presented previously in Pöppl and Bader (2016) and in Pöppl et al. (2016).

16.1. System Design

SWE-X10 shares the numerical properties of SWE. Its sequential core is similar to the one of SWE, but is written entirely[1] in X10. However, SWE-X10 is built around the actor model, and does therefore not utilize MPI or OpenMP for parallelization. Figure 16.2 depicts the basic system design of the application. The top layer, *Actors*, is responsible for the coordination of the computation using the actor model. I will discuss it in more detail in section 16.2. The middle layer, *Patch Coordination* is responsible for implementing the actors' actions on per-actor basis. This involves all computations on a single patch, such as computation of fluxes, application of the patch boundary conditions, or creation of a refined or coarsened patch, if requested. Some functionality, such as the patch iterations on the different available hardware targets, or file I/O, is delegated down to the bottom-layer components *Patch Iteration* or *Utility*, respectively. Finally, the *Setup & Tear-down* component is responsible to set up the actor graph, and to coordinate global I/O tasks such as the loading of complex scenarios, the parsing of command-line arguments or communication with the visualization client from the invasive platform prototype.

For the simulation of complex scenarios, SWE-X10 is able to load bathymetry files provided by General Bathymetric Chart of the Oceans (GEBCO)[2]. The `NetCDFImporter` class may be used to import these bathymetry files, provided that they conform with the NetCDF Climate and Forecast Conventions. Additional options for the simulation run, such as the number of grid cells, the patch size, or the type of time stepping may be selected using the application's command-line interface. When SWE-X10 is executed on the prototype, it is also possible to enable an in-situ visualization that uses a TCP/IP connection to connect to an external system with a running visualization client. If this option is enabled, the application serializes and sends the unknown data of the patch as well as metadata for the iteration to the visualization client.

16.2. Actor-Based Coordination

Similarly to SWE, SWE-X10 relies on a decomposition of the simulation domain into rectangular patches. Following the update scheme outlined in section 14.2, in order to update of cells to a new state $q_{i,j}^{(n+1)}$ it is required to know the values of the neighboring cells quantities at the current time step n. This, in turn, necessitates the transfer of the values of the outermost layer of the adjacent patches. These are typically stored in an extra row or column outside the actual patch, in the *ghost*

[1] Except for the `NativePatchCalculator`, which is a hybrid of X10 and hand-written C++. The class is available only on conventional systems, when the default IBM X10 compiler is used. The invasive X10 compiler does not support such inter-mixing of X10 and C++, as the compilation is performed in one step, unlike with the original compiler, which first generates C++, and then uses another compiler to generate machine code.

[2] Link: `https://www.gebco.net`. Datasets are available at: `https://www.gebco.net/data_and_products/gridded_bathymetry_data/gebco_2019/gebco_2019_info.html`

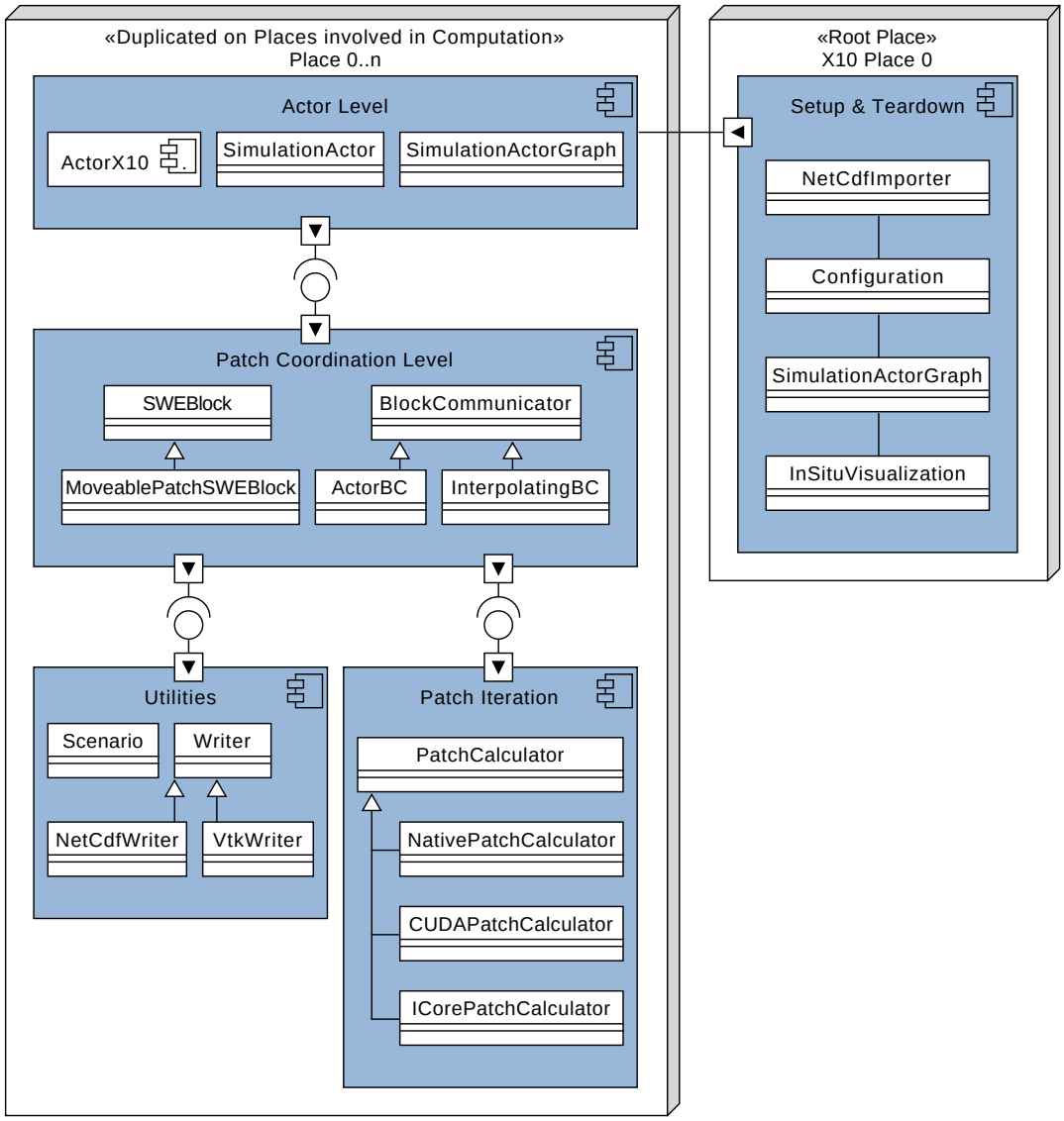


Figure 16.2.: SWE-X10 UML component diagram. The application features a layered approach. The top layer implements the actor-based coordination, and the lower layers contain the data structures to implement the actions. The instances of the classes contained in the components are distributed over all available X10 places. Setup and tear-down of the simulation are managed by another component that is located on place 0 of the X10 PGAS domain.

layer, and they have to be available before a new time step can be computed. In SWE-X10, the task of transferring the ghost layers, and the coordination of the computation falls to the actors. To this end, each patch is allocated a simulation actor. Each actor is connected to the actors controlling the patches adjacent to its own. An example for the resulting domain decomposition is depicted in Figure 16.3. It shows a grid subdivided into 3×3 patches, and 3×3 actors coordinating them. Here, the connection between each two neighboring actors is denoted by a single edge that stands for their bilateral connection. This connection is implemented through four channels, two in each direction (as depicted in Figure 16.4). In each direction there is one channel for control information, such as termination message, and one for the cell quantities of the ghost layer. The token capacity for each channel is set to two for all channel types in order to avoid deadlocks. Even though in the basic version, there may only ever be one token in the channel, the second slot is needed to meet the invariant of the actor state transition: a transition may only be taken if there is both a token available in the incoming channel, and one free slot available in the outgoing channel. As all channels start by sending their initial ghost layer, it is likely that two neighboring actor send each other messages concurrently, and therefore deadlock. Increasing the capacity to two sidesteps this issue. This capacity is sufficient unless different time step sizes are to be accommodated. The graph may be modelled in terms of the FunState model as:

$$
\begin{aligned}
G_{\text{SWE-X10}} &= (A_{\text{SWE-X10}}, C_{\text{SWE-X10}}) \qquad (16.1)\\
A_{\text{SWE-X10}} &= \left\{a_{i,j} |\ 0 \leqslant i < n_x \wedge 0 \leqslant j < n_y\right\}\\
C_{\text{SWE-X10}} &= \left\{c^{\text{D}}_{a_{i,j},a_{i',j'}} |\ \left(i = i' \wedge |j - j'| = 1\right) \vee \left(|i - i'| = 1 \wedge j = j'\right)\right\},\\
&\qquad \cup \left\{c^{\text{C}}_{a_{i,j},a_{i',j'}} |\ \left(i = i' \wedge |j - j'| = 1\right) \vee \left(|i - i'| = 1 \wedge j = j'\right)\right\}.
\end{aligned}
$$

The actor graph is distributed onto the available X10 places using a block-block distribution. For the actors, there are different levels of functionality that may be added successively. The basic level assumes a fixed time step size, and full activity across the entire simulation domain. In a second step, I add lazy activation of actors, so that only actors that have been reached by the wave perform computations In the basic model, actors trigger their neighbors through the ghost layer exchange. Whenever an actor is done with an iteration, it sends its ghost layer data to its neighbors. At the same time, each actor may compute an iteration once it has received ghost layer data from all its neighbors. In terms of the FunState model, actors are modelled as shown in Figure 16.5. For the connection to its neighbor, each actor has incoming and outgoing ports for data and control messages in each of the four directions (top, bottom, left and right). Actors start in the *initial* state. Once are able to write to their channels, they write the initial ghost layer data to their data channels, and set their state to *compute updates*. In that state, they will compute iterations whenever there is ghost layer data available from all connected neighbors (*mayRead()*), and there is sufficient space remaining in the channels connected to the outgoing ports (*mayWrite()*), as long as the current simulation time t_{cur} is smaller than the specified end time t_{end}. The ghost layer data tokens take the shape of one-dimensional arrays (`Rail[Float]{self.size == this.patchSize}`). In X10, arrays are conventional objects, and thus allocated on the heap (during inter-place transfer), and passed by reference. The references remain in the channel until the state transition is taken. Then, they are passed into the `SWEBlock`, which handles the local iteration. At the end of the state transition, the copy layers of the patch are exported into another token that is then inserted into the outgoing channel. The big advantage over the SWE implementation is that with that scheme, coordination is completely local, and there is no longer a global synchronization step. The individual patches do not need any knowledge about the

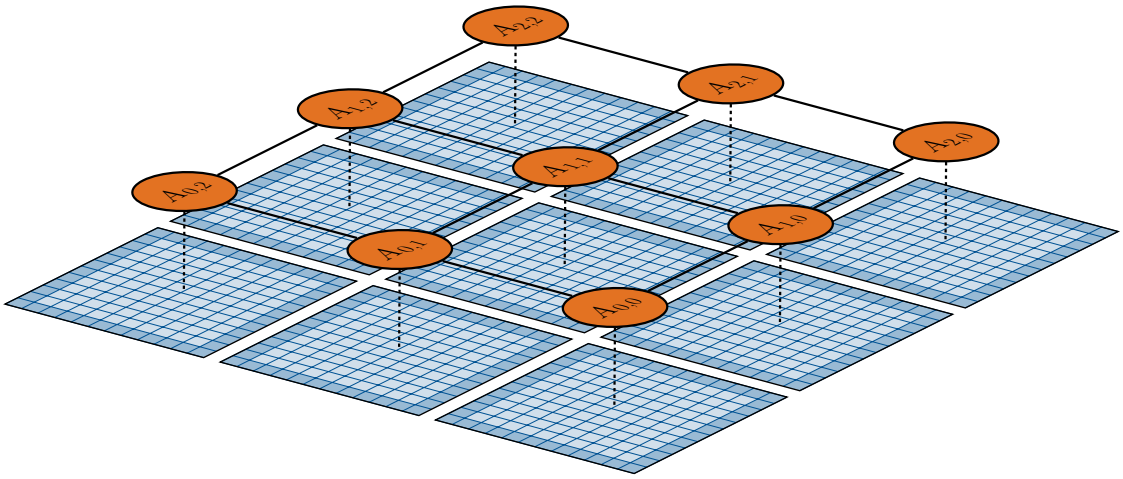


Figure 16.3.: Decomposition of the simulation domain into 3×3 patches. Each patch is assigned an actor $A_{i,j}$ that is connected with its direct neighbors.

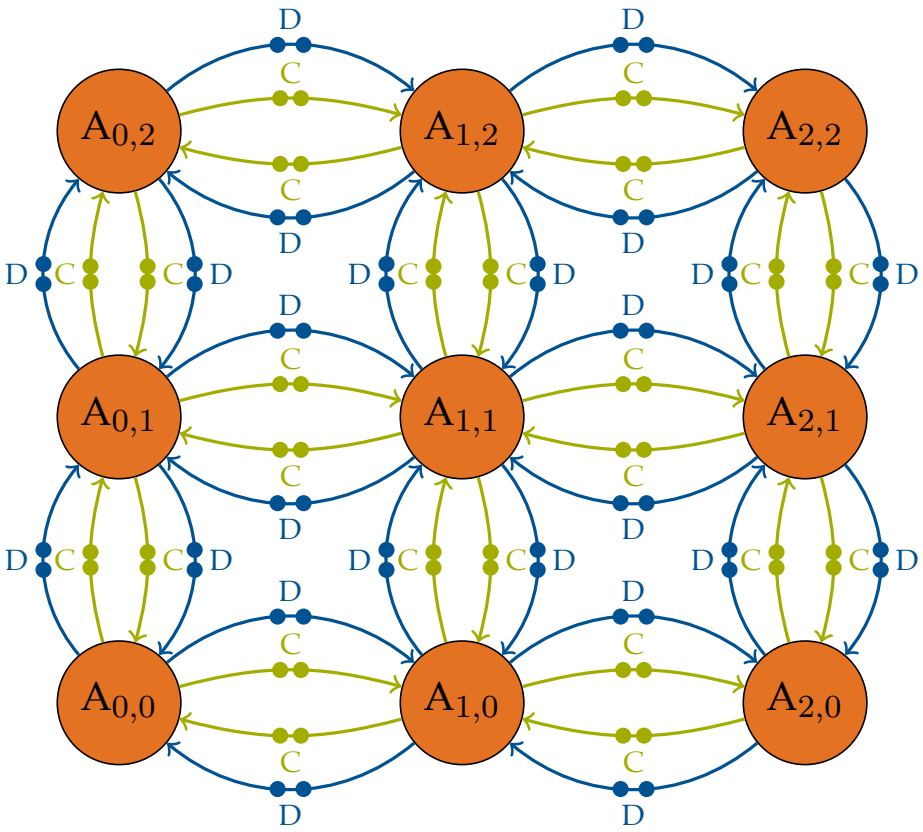
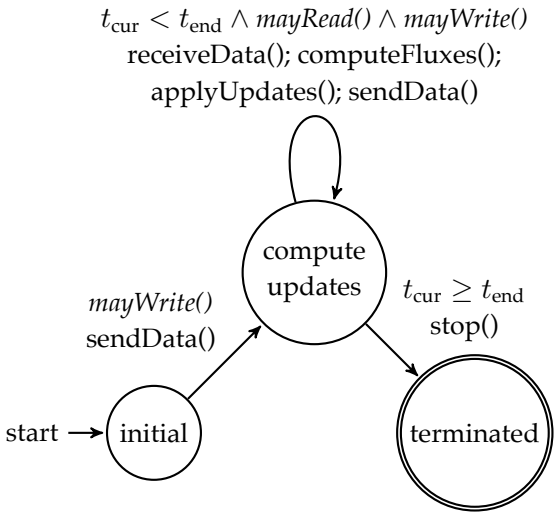


Figure 16.4.: Actor Graph with 3×3 actors. Each actor (orange) is connected to its neighbors through four channels, one channel for ghost layer data exchange (depicted in blue) and one for control messages (depicted in green). Token capacity is set to two for all channels.



(a) Simulation Actor FSM

$$
\begin{aligned}
a_{i,j} &= \left(I_{a_{i,j}}, O_{a_{i,j}}, F_a, R_a\right) \\
I_{a_{i,j}} &= \left\{ip^{\text{C,D}}_{i',j'} | c^{\text{C,D}}_{a_{i',j'},a_{i,j}} \in C_{\text{SWE-X10}}\right\} \\
O_{a_{i,j}} &= \left\{op^{\text{C,D}}_{i',j'} | c^{\text{C,D}}_{a_{i,j},a_{i',j'}} \in C_{\text{SWE-X10}}\right\} \\
F_{\text{SWE-X10}} &= F_{\text{Guard}} \cap F_{\text{Action}} \\
F_{\text{Guard}} &= \{mayRead(), mayWrite(), \\
&\qquad [t_{\text{cur}} < t_{\text{end}}], [t_{\text{cur}} \geq t_{\text{end}}]\} \\
F_{\text{Action}} &= \{\text{receiveData}(), \text{computeFluxes}(), \\
&\qquad \text{applyUpdates}(), \text{sendData}()\}
\end{aligned}
$$

(b) Simulation Actor

Figure 16.5.: Finite state machine for the simulation actor. Italicized functions are guard functions, the rest are actions. (Adapted from Roloff et al. (2016))

internal state of their neighbors, and may send their tokens whenever they are done with their local computations.

16.3. Lazy Activation of Actors

The basic model already introduces local coordination to the computation. Actors now only depend on their direct neighbors, and coordinate amongst themselves. The activation scheme introduced above may now be exploited in order to reduce the number of calculations performed by the application. In the basic implementation, an iteration is performed regardless of whether it is actually needed. As described in subsection 14.1.1, the shallow water equations are hyperbolic PDEs. This type of PDE exhibits "wave-like" behavior, i.e. there is a finite propagation speed of the tsunami wave across the simulation domain. Cells that have not yet been reached by the wave will be at rest, i.e. there is no change in the water height or momenta. An actor may find out whether it is initially needed from the scenario: if its patch contains part of the initial perturbation, it should start as active, otherwise start as dormant. The set of active actors propels the simulation forward, and as soon as the wave reaches the boundary of a dormant actor's patch, it is activated and joins the computation. In many cases, the initial perturbation will be limited to just a few patches. The remaining patches will only become active as the simulation progresses.

To enable this behavior, the structure of the actor graph does not need to be changed from the one described in section 16.2. It suffices to adapt the set of actions and to change the actors' FSM, shown in Figure 16.6 to model the new behavior. Initially, each actor determines its activity state, and sends it to its neighbors. Actors that contain part of the initial wave set their state to *propagating wave*, while those that do not set their state to *lake at rest*. Active actors perform simulation steps as

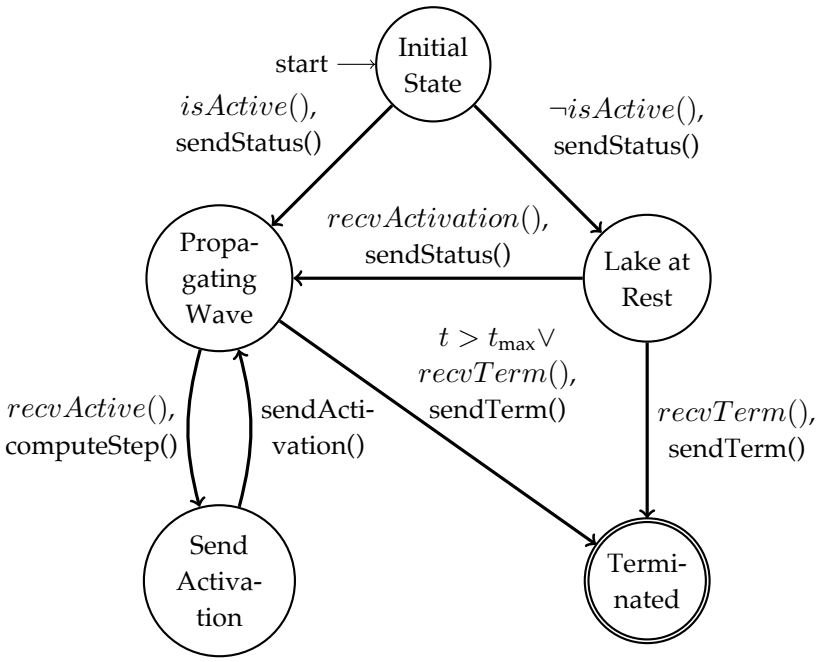


Figure 16.6.: Finite State machine for the simulation actor with lazy activation enabled. As before, italicized methods are guard functions, and functions written in normal letters are actions. The set of functions is omitted for brevity. (Pöppl et al., 2016)

described above. Unlike previously, ghost layer data is only needed from active neighbors now. The actor tests the activation conditions using the aggregate guard *recvActive()*. If an actor does not have any active neighbors, it may just keep computing updates until the wave reaches one of the boundaries of its patch by triggering itself at the end of its actions. During the computation of the update, the actor determines for each of its copy layers whether the update actually changes any values. If there are changes, or the neighbor is already active, the updated data will be sent at the end of the iteration. If the neighbor is still inactive and there is a non-zero update, the actor will also send a control message stating that it shall henceforth provide updates (sendActivation()). Once this message is received (*recvActivation()*), the neighbor in *lake at rest* state will change its state to *propagating wave*, and propagate its new status to all its neighbors, so that its other active neighbors know to expect updates from it, and may in turn send their updates. Once an actor reaches its termination condition, it will send a termination signal to all its neighbors, and terminate. The same happens if a termination signal is received (*recvTerm()*).

16.4. Patch-Level Calculations

The actions used within the FSM of the actors' are implemented in the lower layers of SWE-X10, specifically the `MoveablePatchSWEBlock` class. In order for the application to provide a good base for the evaluation of the actor model, its performance characteristics need to be comparable to the prior application, SWE. Just using X10 code for the computational hotspots proved insufficient to get a performance comparable to the SWE version. Nevertheless, obtaining a high performance

using X10 is possible if common pitfalls are avoided (Tardieu et al., 2016). In the following, I will highlight some measures taken to improve the performance in SWE-X10.

One potential problem for X10 may be the two-step compilation process of X10. For the IBM X10 compiler, the compilation of X10 to machine code is a two-step process using C++-98 as an intermediate language. This technique is a common approach for new programming languages, as it allows the compiler writer to benefit off the previous work done by the developers of the target language compiler. Some deficits, such as the lack of an effective available expression analysis may be traced back to this approach (Horie et al., 2015). The optimizations implemented in the IBM X10 compiler target mostly high-level optimization opportunities such as the lowering of range-based `for`-loops to low-level C-style loops that avoid heap allocations. The commonly implemented optimization steps are left to the C++ compiler. However, the generated C++ code may be too complicated for the C++ compiler to recognize as an optimization target. For example, X10 array access expression `val i = a(3);` is translated by the X10 compiler to:

```
x10_double b = (__extension__ ({
    ;
    x10_double ret6171;
    ret6171 = (a->FMGL(raw))->__apply(((x10_int)3));
    ret6171;
}))
```

To get to the raw memory value, two levels of indirection need to be resolved and a function called. Depending on the X10 compiler settings, the function call will also perform a bounds check to detect out-of-bounds accesses. As the C++ compiler may not have the body of the `__apply()` function available[3], it will not be able to rule out side effects, and will therefore reject the expression as a target for optimization. In an HPC context this is especially problematic, as the use of SIMD instructions is necessary to reach a CPU's peak performance. Auto-vectorization, however relies on the compiler to find code sections that may be safely executed in parallel. If the compiler encounters an eligible loop, it may generate code to compute several iterations in parallel using SIMD instructions. To do that, the compiler needs to prove that there are no adverse side effects to vectorization. If the compiler is unable to do so, or it does not have all necessary code available, only scalar code will be generated. The computational hotspot of SWE, the solution of the edge-local Riemann problems on the unknown arrays has been vectorized before by Bader et al. (2014) using the Intel C++ compiler. However, using plain X10, vectorization of that computation was not possible. Vectorization failed due to the loops' complexity.

A second pitfall is the prevalence of heap allocations. In C and C++, it is a common technique to declare small local arrays directly, such as `float arr[4]; arr[2] = 4.2f`. These arrays may be used as local variables with *automatic* storage duration, i.e. they will be allocated on the stack along with all other local variables. In X10, there is no such completely stack-allocated array type, and it is not possible to program one of arbitrary size. The most primitive type of array, the `Rail[T]`, a one-dimensional, zero-based, fixed-size storage container for instances of a single type is implemented as a class, and as such, it is implemented on the heap. As previously shown, there are two levels of indirection between the `Rail[T]` object and the raw memory. All types

[3] It is part of the X10 standard library, and therefore defined in a different compilation unit.

of X10 arrays use an instance of the `IndexedMemoryChunk[T]` as a backing storage. For `Rail[T]` instances, its functionality may be directly accessed, but for more complex array types, there needs to be some sort of address translation. The `IndexedMemoryChunk[T]` holds a pointer to the raw memory that is not exposed as an X10 object. All memory accesses need to pass through the `IndexedMemoryChunk[T]`'s instance methods. While it is possible to annotate objects with `@StackAllocate` to direct the compiler to generate the object on the stack, this is not helpful for arrays, as the annotation only works for constructors, and the `IndexedMemoryChunk[T]` may only be instantiated using a factory method. A first implementation of the Riemann solvers in X10 used `@StackAllocate`-annotated arrays. Replacing these with local scalar variables[4] sped up that early version of SWE-X10 by a factor of ten.

As introduced in chapter 5 and subsection 8.1.1, there are two different X10 compilers that are used to compile SWE-X10, namely the IBM compiler and the invasive X10 compiler developed within InvasIC. For the IBM compiler targeting HPC systems, it is possible to integrate native C++ for the computational hotspots of the simulation. The aforementioned two-step compilation process allows for the developer to insert their own code into previously generated C++ code. The `@NativeRep` annotation may be used to tell the compiler to skip the code generation for the annotated class. The X10 compiler will perform all other steps using the X10 class, therefore the class is indistinguishable from other X10 classes from the X10 perspective. During the second step of the compilation, the user has to compile and link the compilation unit containing the custom code manually. As long as the interface matches, there no issues will occur during compilation. However, the application developer is now responsible to extend the automatically generated serialization code to include any attributes that have been added to the class on the C++ level. I used this technique to implement the patch iterations needed to compute net updates, and to apply them using C++ (in the `NativePatchCalculator` class). Instead of using the functionality provided by the X10 array (`x10.regionarray.Array`) class, I directly accessed the underlying memory segment. The loops are annotated with the Intel Compiler directive `#pragma simd` to instruct the compiler to emit vectorized code. This is sufficient to increase the performance by a factor of four over the non-vectorized, pure X10 version. When this optimization is enabled, the more general, purely X10-based `MoveablePatchSWEBlock` delegates the patch iteration to the patch iterator class.

16.5. Performance of SWE-X10 on CPUs

The application performance of SWE-X10 was evaluated with respect to different layers of parallelism. First, I evaluated the performance on the single-core-level, comparing the pure X10 version against the vectorized C++ code. I then compared the performance of SWE-X10 on a single node and in a multi-node setting to SWE, the prior BSP approach. Finally, I compared the difference in time-to-solution and the CPU time used with and without the lazy activation technique.

[4] e.g., `@StackAllocate val a = @StackAllocate new Rail[Float](2);` would become `var a0:Float = 0.0f, a1:Float = 0.0f;`

All tests were performed on the, now-deconstructed, MAC Cluster[5], a small development cluster that was equipped with, amongst others, 28 nodes with two Intel Xeon E5-2670 CPUs (*Sandy Bridge micro-architecture (μArch)*) each. The peak single-precision floating-point performance for the CPU nodes is 332.8 GFlop/s, and the STREAM triad performance was measured as 108.9 GB/s per node. The nodes were connected using Mellanox InfiniBand QDR. For all the following measurements, I used the IBM X10 compiler as a front-end compiler, and the Intel Compiler 16.0 as the backend compiler.

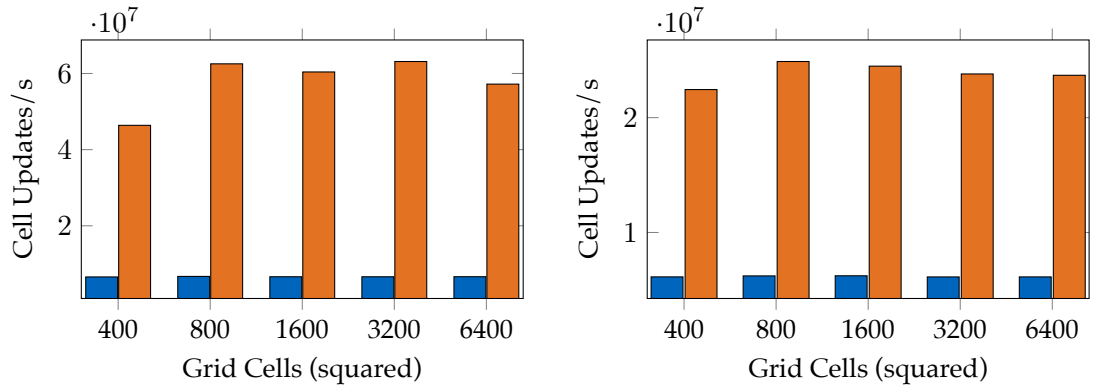
The `NativePatchCalculator` described in section 16.4 features loops that are annotated with the Intel-specific `#pragma simd`[6] annotation. The directive forces the Intel Compiler to generate vectorized versions of the annotated loops. I compared the purely X10-based version of both the HLLE solver and the f-Wave solver to their respective counterparts that use the vectorized C++ loops. Performance was measured based on the execution time of the entire actor graph execution with four actors on a single core. Therefore, measurements do not show the Riemann solver performance alone, but the performance of the entire application, including overhead from ActorX10, and the updating of the unknown arrays using the fluxes computed by the Riemann solvers. Performance results (see Figure 16.7, previously shown in Teich, Kleinöder, and Mattauch (2015)) show a speed-up of $5\times$ over the non-vectorized X10 version. These results mirror the ones reported earlier by Bader et al. (2014) for SWE, which allows for an easy and fair comparison of the actor-based SWE-X10 to the BSP-based SWE.

Next, the shared-memory performance of SWE-X10 was compared to the performance of SWE. These results were previously shown in Pöppl and Bader (2016) and Pöppl et al. (2016). The test was performed on a single node of the MAC cluster, using configurations ranging from one to sixteen cores. Both applications solved a radial dam break scenario using the HLLE solver. Each CPU core was assigned a region of 1024×1024 grid cells. In SWE-X10 these were distributed onto 2×2 actors with patch sizes of 512×512 grid cells each. The results of this test are shown in Figure 16.8. SWE-X10 starts with a single-core performance of about 25 GFlop/s. The highest performance is reached at about 75 GFlop/s using 10 cores. Adding more cores to the computation did not yield a higher performance. It is likely that the memory bandwidth is saturated at this point. SWE reaches a similar performance, however only at a higher core count of 14 cores. Both applications manage to reach about 23 % of the node's peak performance.

The multi-node performance of SWE-X10 was evaluated through another weak-scaling test. As with the single-node test, the result was previously reported in Pöppl and Bader (2016) and Pöppl et al. (2016). In SWE-X10, the place granularity was set to eight cores (or one CPU) per place. As with the single-node test, each core wa assigned 1024×1024 grid cells subdivided onto 2×2 actors, resulting in 32 actors per CPU. The test was performed with configurations ranging from one CPU to 16 nodes, the larges available configuration on the MAC Cluster. The performance results (see Figure 16.9) show a linear scaling from one to eight nodes both in SWE-X10 as in SWE. At sixteen nodes a performance degradation is observed. The degradation occurs with both applications, implying an issue with the hardware of the cluster. In all the configurations displayed here, SWE is outperformed by SWE-X10. This may be explained by the different

[5] `http://www.mac.tum.de/wiki/index.php/MAC_Cluster`

[6] This annotation has since been deprecated and superseeded by a number of OpenMP SIMD pragmas. The version of SWE-X10 used to measure the benefits of vectorization used the old annotations at the time of measurement.



(a) f-Wave Solver

(b) HLLE Solver

Figure 16.7.: Single-Core Performance of SWE-X10. The blue bars show the scalar version (using pure X10) of the solvers, the orange bars show the native, vectorized C++ version. (Result previously shown in Teich, Kleinöder, and Mattauch (2015))

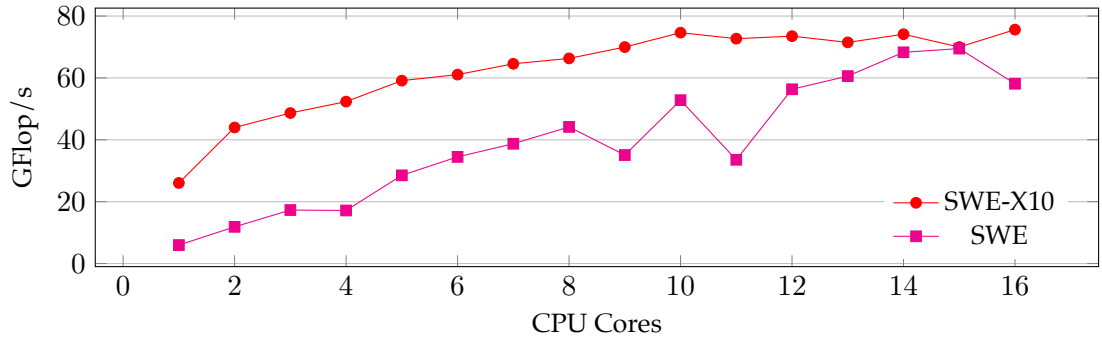


Figure 16.8.: Single-node performance of SWE-X10 vs. SWE. I assigned four actors with 512×512 cells to each core (weak scaling). The performance of SWE-X10 saturates earlier (10 cores) compared to SWE (14 cores), but both codes reach the same peak performance (75 GFlop/s). (Result from Pöppl and Bader (2016) and Pöppl et al. (2016))

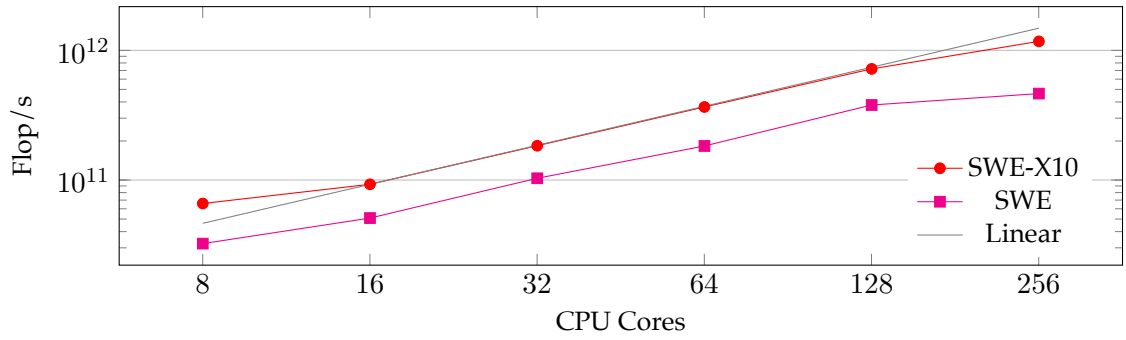


Figure 16.9.: Weak scaling of SWE-X10 vs. SWE from one CPU (8 cores) up to sixteen nodes (256 cores). 32 actors with 512×512 cells were placed on each CPU. The gray line illustrates perfect scaling from one node. (Result from Pöppl and Bader (2016) and Pöppl et al. (2016))

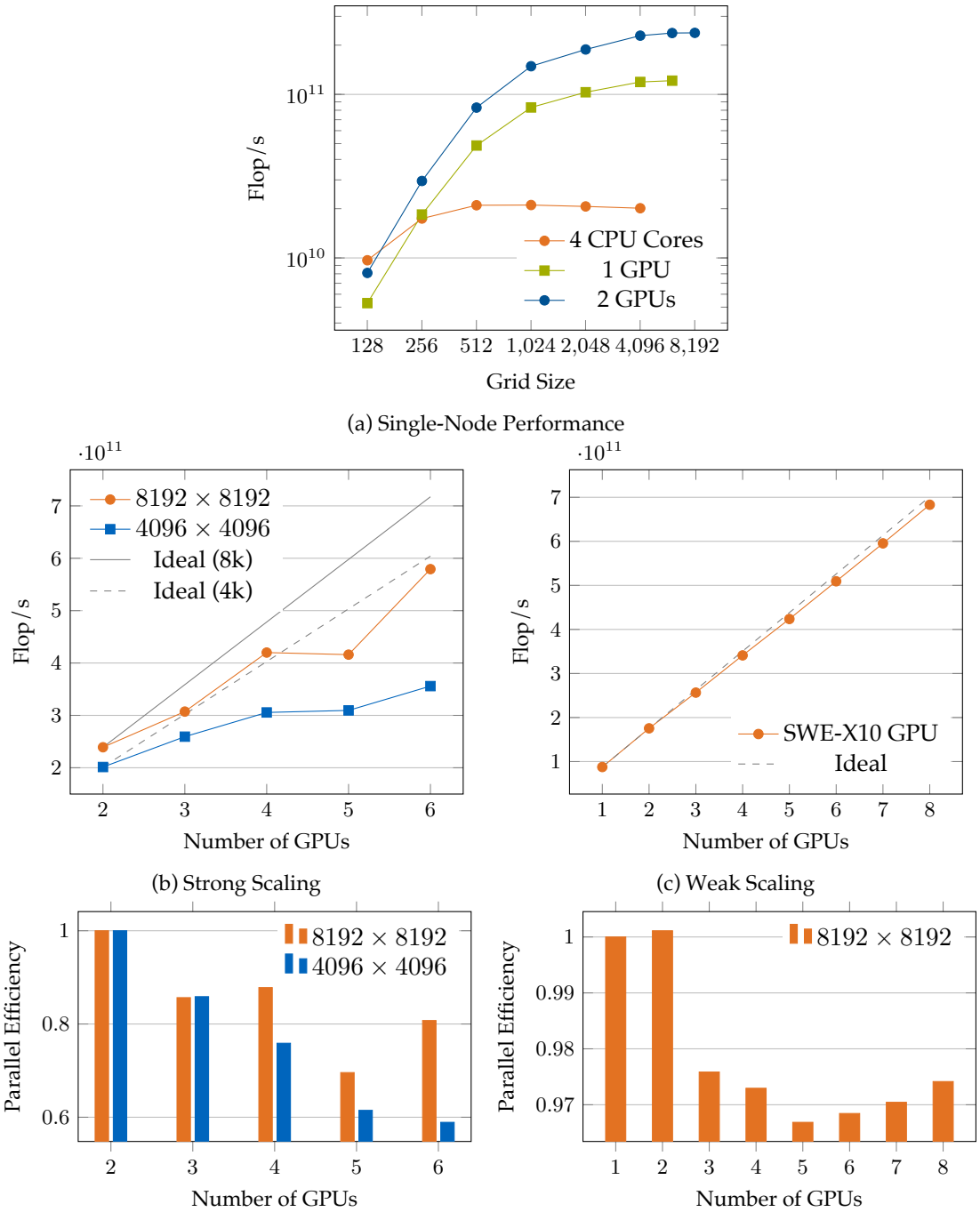
communication patterns used in the two applications. The actor model used in SWE-X10 enables an overlapping of communication and computations by virtue of having multiple patches per core. An actor that already received ghost layer data from its neighbor may start computing its next time step, while other actors still wait, or are sending updates from the previous time step. In SWE, such an overlap is not supported, as the application follows the BSP model, and has clearly separated, non-overlapping phases for computation, communication and synchronization.

These tests show that, given the right circumstances, the performance of SWE-X10 is comparable to, or even better than the one of SWE. However, these results are specific to the cluster used for the evaluation. The hardware features relatively low number of relatively strong cores (and a low floating-point performance, compared to more modern CPUs). This favors SWE-X10, as the coordination overhead imposed by ActorX10 is relatively high. For configurations with smaller patch sizes, or for hardware architectures with weaker CPU cores, such as the Intel Xeon Phi, this becomes a larger problem. The cores of the Xeon Phi feature, relative to the Sandy Bridge cores used here, a much more capable floating-point unit that uses 512 Bit wide vector registers, but the rest of the core has a much more simple and less capable architecture. This favors a more simple control flow with a greater focus on floating-point arithmetic, and therefore SWE-X10's performance degrades significantly. This was one of the motivations for the development of Actor-UPC++, and will be discussed in more detail in section 18.2.

16.6. Performance of SWE-X10 on GPUs

In two bachelor's theses, support for patch iterations on CUDA-capable GPUs was added (Gärtner, 2016; Molzer, 2017). Molzer and Gärtner added another patch iterator, the `CUDAPatchCalculator`, to enable the execution of the local path iteration on GPUs. The patch calculator uses CUDA streams to schedule different kernels that do not have data dependencies concurrently. For example, the calculation of the vertical and the horizontal fluxes may be performed in parallel. The same applies to cells at the boundaries of the patch that are treated separately in SWE-X10. The resulting dependency graph may be encoded using streams to enable the GPU to schedule kernels as efficiently as possible. Furthermore, the kernels themselves were optimized to use the memory hierarchy, including the shared memory, efficiently. For the parallel computation of the maximum wave speed, an efficient technique proposed by Harris (2010) was used. These techniques, together with ActorX10, enable the efficient use of multiple GPUs in the same calculation.

The performance of SWE-X10 was evaluated in the bachelor's thesis of Molzer (2017). He performed the evaluation on the MAC Cluster's NVIDIA partition, which contained four nodes with two NVIDIA Tesla M2090 GPUs (Fermi μArch) each. The Tesla M2090 is clocked at 1.3GHz and reaches a peak single-precision floating-point performance of 1332 GFlop/s It has 6 GB of on-device memory and a peak bandwidth of 178 GB/s (NVIDIA Corporation, 2012). Molzer evaluated the performance of his improvements of the CUDA patch iterations on the MAC cluster, performing strong and weak scaling test, as well as a comparison with the prior CPU solution. In the first test, depicted in Figure 16.10a, performance of configurations using four CPU cores, a single GPU and two GPUs was compared at different patch sizes. Compared to the CPU solution, the GPU requires larger patch sizes to perform efficiently. Performance on the GPU only levels off



(a) Single-Node Performance

(b) Strong Scaling

(c) Weak Scaling

(d) Parallel Efficiency – Strong Scaling

(e) Parallel Efficiency – Weak Scaling

Figure 16.10.: Performance of SWE-X10 on the GPU. The test in Figure 16.10a has been performed on a single node with different patch sizes and four actors. Strong scaling was performed using two different patch sizes, and up to 6 GPUs. For the weak scaling test, actors with 1024 × 1024 grid cells were used. (Experiment performed by Molzer (2017))

for configurations above a patch size of 1024×1024 grid cells. For smaller patch sizes, the amount of parallelism is not sufficient to saturate the entire GPU. The same holds for the configurations using two GPUs.

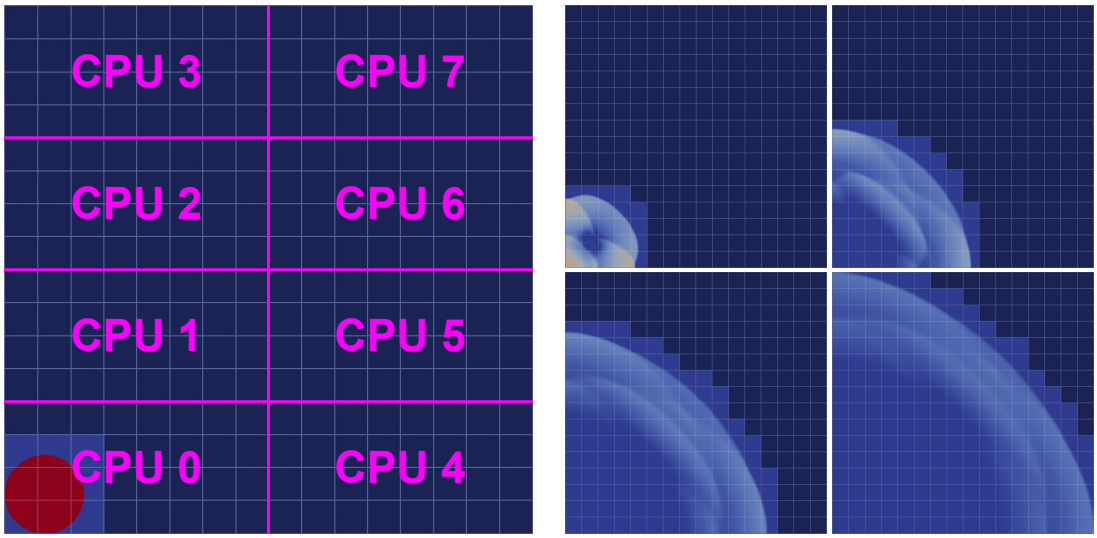
The strong scaling performance, depicted in Figure 16.10b and Figure 16.10d, reveals a similar picture. The comparatively lower performance observed at three and five GPUs used may be explained with imperfect load balancing given the odd number of GPUs. A weak scaling test performed using GPUs (results shown in Figure 16.10c and Figure 16.10e) revealed a result similar to the one observed in Figure 16.9. The actor model with its local coordination scheme enables good scalability. This is demonstrated here once more, with a parallel efficiency of over 97 % for an execution with eight GPUs.

With 10 % of the potential peak performance on the NVIDIA Tesla M2090, the obtained performance is relatively low. However, the structure of the computation limits the maximum performance attainable for the GPU. For each Riemann problem, three unknown values $(h, hu, b)\,T$ on two sides of the cell boundary are read, and two updates and the maximum wave speed are written. Assuming single precision floating-point numbers, the arithmetic intensity is at $\approx 3\,\mathrm{Flop/B}$. Thereafter, the fluxes are used to compute the next time step. This operation performs three loads and three stores per cell, and performs six operations, yielding an arithmetic intensity of $0.25\,\mathrm{Flop/B}$. This makes both operations memory bound. Furthermore, the solver exhibit a number of branches that may lead to thread divergence, further reducing the performance. Nevertheless, the performance of SWE-X10 is comparable to the one determined for SWE by Hölzl (2013). In future work, it may be interesting to evaluate the performance of SWE-X10 with patches executed both on the CPU and the GPU at the same time.

16.7. Evaluation: Lazy Activation of Actors

In the following, I will demonstrate the benefits of lazy activation implemented using the actor model. SWE-X10 currently does not support load-balancing of actors. Instead, actors are distributed to their place at the start of the application. Therefore, the full benefit of lazy activation in terms of time-to-solution cannot currently be demonstrated. Instead, I measured the CPU time used by the application by summing up over the time used by each CPU to compute the scenario. This metric assumes that CPUs of inactive actors otherwise remain idle, and thus provides a rough approximation of the required energy to compute the solution. In the context of the invasive computing environment, it may be possible to invade tiles gradually as actors activate, hence allowing for a more efficient use of the system. Alternatively, multiple operating points may be used for the different simulation states, e.g. one with a low amount of cores for the beginning of the simulation, and another for a later stage when all actors are active.

As the test scenario, I chose a modified radial dam break scenario. In this scenario, the initial water elevation is placed in the lower left part of the simulation domain, and the boundary condition are set to wall type. Simulation time was set to 90 simulated seconds, a sufficient time for the wave to reach the entire domain. The grid resolution was set to 8192×8192 grid cells, distributed onto actors containing patches of 512×512 grid cells each. Actors were distributed onto eight



(a) Initial Setup (b) Wave Propagation with Lazy Activation

Figure 16.11.: Setup to test lazy activation of actors. The red circle marks the initial water displacement. Throughout the simulation, it will collapse and radiate outwards as a propagating wave. Each little square denotes an actor; the initial assignment of actors to places (i.e., CPUs), is marked in magenta.

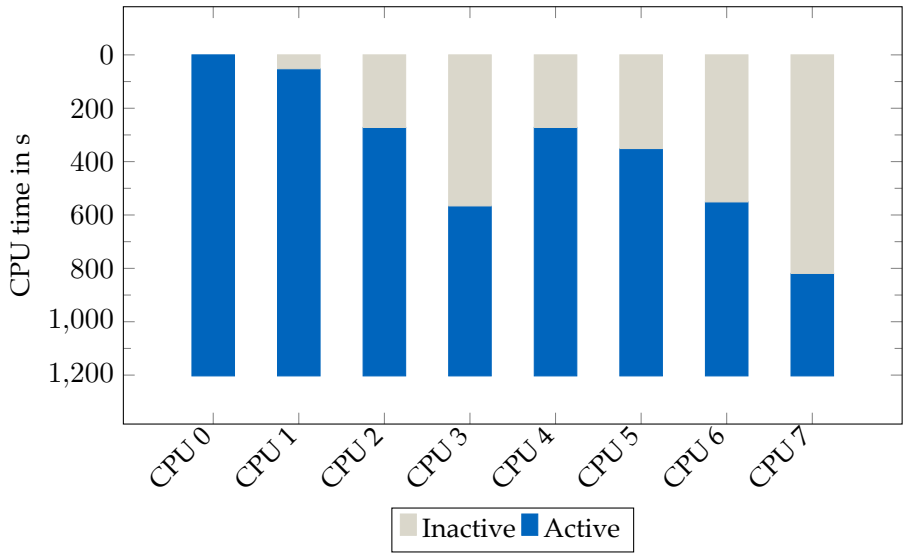


Figure 16.12.: CPU activity for the test run with lazy actor activation. Each bar represents the activity of one CPU; gray marks inactive time and blue marks time spent computing updates. (Result from Pöppl et al. (2016))

CPUs on four nodes of the MAC Cluster according to the scheme displayed in Figure 16.11a. The application recorded the activation time of the first actor for each X10 place. The time from first activation to the end of the simulation was then used for each actor to compute the overall CPU time. For the configuration without lazy activation, the overall execution time times the number of CPUs in the computation yields the CPU time, as all CPUs are active and computing updates from the start.

For the run without lazy activation, I measured an overall execution time of 1433 seconds. All actors are active from the start of the simulation, therefore, the overall CPU time utilized may be summed up to 12264 CPU seconds, or 3.41 CPU hours. In comparison, the version with lazy activation takes 1203 seconds to complete. Summing over the measured CPU activity (see Figure 16.12) yields a total number of 6741 CPU seconds, or 1.87 CPU hours. Hence, in terms of CPU time spent, lazy activation enables significant savings in terms of used system resources. Naturally, the actual resource utilization benefits of lazy activation heavily depend on the initial scenario. In cases where the initial disturbance is large and located in the center of the simulation domain, the gains will be significantly smaller. However, typical tsunami simulation setups usually contain a relatively localized initial disturbance. Large parts of the simulation domain will be at rest initially, especially if propagation across the entire ocean is considered.

17. Shallow Water on a Deep Technology Stack

In chapter 8, Invasive Computing was introduced. The project's novel programming model and hardware architecture form the environment for which SWE-X10 was developed. One of SWE-X10's design goals was support for heterogeneous architectures. In the invasive computing project, there are two types of accelerators available, the TCPA and the *i*-Core. Within a collaboration with several subprojects of InvasIC[1], we implemented support for the *i*-Core accelerator in SWE-X10. Its results were previously shown in Pöppl et al. (2018), and later used successfully to demonstrate the vertical integration of the invasive computing project as part of a large demonstration at the project's review for the third funding phase.

Use of the *i*-Core within SWE-X10 is particularly interesting, as it is a representative for the class of reconfigurable resources on one side, and has a very interesting and fine-granular approach to parallelism on the other side. In the wider field of HPC, the use of specialized hardware has become more common due to the slowing increase of performance in general-purpose hardware. The use of specialized hardware can greatly accelerate the specific applications, as previously demonstrated, e.g. with the Anton-2 system (Shaw et al., 2014), through the use of NVIDIA's tensor cores for performing LU decompositions (Abdelfattah, Tomov, and Dongarra, 2019), or Google's Tensor Processing Unit (Jouppi et al., 2017). Ours is not the only project to explore the use of reconfigurable fabric within the field of HPC, but others typically investigated loosely-coupled solutions (Altera, 2007; Flich et al., 2017), while we focus on tightly-coupled reconfigurable fabric. In the field of embedded computing, the other research domain involved with the invasive computing project, *reconfigurable processors* such as *i*-Core (see also section 8.2) are commonly utilized to accelerate specific parts of the application.

As a result of its tight integration into the general-purpose Processing Element (PE), the reconfigurable fabric has direct access to the PE's internal state, i.e. its internal registers, caches, and TLM. This allows for the implementation of fine-granular acceleration of specific hot-spots within the application. Within SWE-X10, the main task is the computation of the Riemann problems between cell boundaries, which is solved using appropriate Riemann solvers such as the f-Wave or the HLLE solver. In the following, I will present the results of accelerating SWE-X10 with an *i*-Core Custom Instruction (CI) for the f-Wave solver. The efficient implementation of the acceleration serves to demonstrate the benefits of controlling the entire compute stack. Aside from the acceleration of the computation, modifications to the operating system, the compiler and the application were required.

[1] Namely the following subprojects: A4, B1, C1 and C3. My project (A4 - Characterisation and Analysis of Invasive Algorithmic Patterns) was responsible for SWE-X10. Project B1 (Adaptive Application-Specific Invasive Micro-Architectures) was responsible for the *i*-Core CI. Project C1 (Invasive Run-Time Support System (*i*RTSS)) was responsible for the operating system support. Project C3 (Compilation and Code Generation for Invasive Programs) was responsible for the X10 compiler support. Work was performed for the most part in a close on-site collaboration.

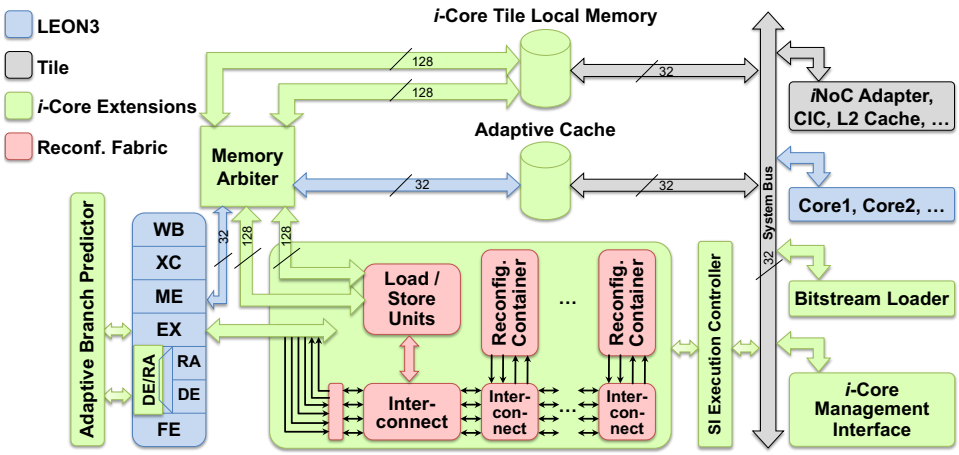


Figure 17.1.: *i*-Core Overview (Damschen et al., 2020)

17.1. Acceleration of Approximate Riemann Solvers using *i*-Core

i-Core, introduced in section 8.2, combines sequential execution following the von-Neumann model with reconfigurable fabric. We used it to provide acceleration to the computational hot-spot of SWE-X10, the computation of the appropriate solution of the Riemann problem using the f-Wave solver. The entire process, as described in subsection 14.3.1, has been implemented in a single CI. The CI is implemented as a μProgram that is invoked when the `fwave` instruction is brought to execution by the *i*-Core on the *CI Execution Controller*. μPrograms are represented by a data-flow graph that represents the computations performed by the CI, and their data dependencies. The computations are performed by the application-specific accelerators. Depending on the structure of the graph, multiple operations may be scheduled for execution concurrently on several accelerators loaded onto the available reconfigurable containers. Furthermore, data may be loaded and stored from the core's registers as well as the *i*-Core TLM. Access to the *i*-Core TLM is performed using a 128-bit-wide bus with a single-cycle latency. For the best possible performance, this memory segment should be used, however, it is also possible to access other memory types, such as the global memory, or the normal TLM. Access to the latter is provided using the tile-bus. The application isolation provided through invasive computing allows us to rely on the *i*-Core to have the CI ready as long as we need it, i.e. for as long as the tile containing the *i*-Core remains invaded. We therefore only need to load the CI and configure the fabric once at application startup, which takes about 5.5ms given a reconfiguration bandwidth of 100 MB/s.

The f-Wave solver consists of mostly floating-point instructions such as additions, multiplications and some divisions. Therefore, we configured the fabric with accelerators for these common operations. The accelerators provide these common operations in the form of micro-operations (μOps) to be used in the CI. The configuration of the accelerators is described in Table 17.1. We configured the fabric with accelerators for simple arithmetic operations (`FP_MAC`), i.e. additions, subtractions, multiplications and fused-multiply-add operations, for divisions (`FP_DIV`), for square root calculations (`FP_SQRT`), and for miscellaneous comparison operations (`FP_UTIL`). The `FP_MAC` accelerator is based on the one by Bauer et al. (2015). CIs may utilize multiple accelerators in parallel. Container space permitting, configuring highly-utilized accelerators multiple times

Table 17.1.: Pipelined floating-point accelerator types available for *i*-Core to be used in SWE-X10. (Pöppl et al., 2018)

Accelerator	Operations	Min. / Max. Latency[1]	Initiation Interval
`FP_MAC`	Add/subtract, multiply, multiply-accumulate	3 / 5	2
`FP_DIV`	Divide, reciprocal	6 / 6	2
`FP_SQRT`	Square root	5 / 5	2
`FP_UTIL`	Min/max, absolute and compare (<, >)	3 / 3	2

[1] Clock cycles on the reconfigurable fabric

can benefit a CI's latency. The current configuration of the *i*-Core contains five reconfigurable containers. We chose to configure them with two `FP_MAC` accelerators, and one each of the `FP_DIV`, the `FP_SQRT` and the `FP_UTIL` accelerator. The data-flow graph of the f-Wave solver contains 54 floating-point operations, and 43 other (address generation, memory access, communication between accelerators and execution of accelerators) operations, leading to 97 nodes in total. With the configuration described above, the micro-program may be executed in 41 steps. The accelerators perform their work in a pipelined fashion. Without this, the number of steps needed would almost double to 71 steps. Each program step takes two clock cycles, which leads to a total of 82 cycles to execute the μProgram.

17.2. Changes in the Middleware

Integration of the *i*-Core CI into SWE-X10 required the bridging of several layers of abstraction. X10 neither provides support for inline assembly instructions (to invoke the CI directly), nor is it aware of the underlying heterogeneous architecture. Therefore, the effective use of the CI requires support from the InvadeX10 compiler and runtime as well as the operating system OctoPOS.

OctoPOS features a light-weight and fine-grained parallelism model using *i*-lets. It relies on cooperative scheduling (see Figure 17.2). When an *i*-let is scheduled onto the core, it will typically stay there until it finishes its execution. There is no preemption, and all available compute resources are to be used by the scheduled *i*-let. This allows for a reuse of the *execution contexts* between the execution of different *i*-lets, as the call stack of an *i*-let will be empty once it terminates, resulting in a very low overhead for the scheduling and dispatch of *i*-lets. It is only necessary to bind an *i*-let to its context if it encounters a blocking operation. In that case, the *i*-let and its context are moved to the block list, another context is added to the core, and another *i*-let is executed on that context. Once the reason for a blocking operation has been resolved, the *i*-let is re-added to the ready list, and may be scheduled once again to a core.

This scheme works well for conventional processor architectures. However, an *i*-Core tile consists of an *i*-Core alongside with three conventional LEON 3 cores. The `fwave` instruction is implemented as a processor instruction, one that is only available on the *i*-Core. Essentially, the *i*-Core

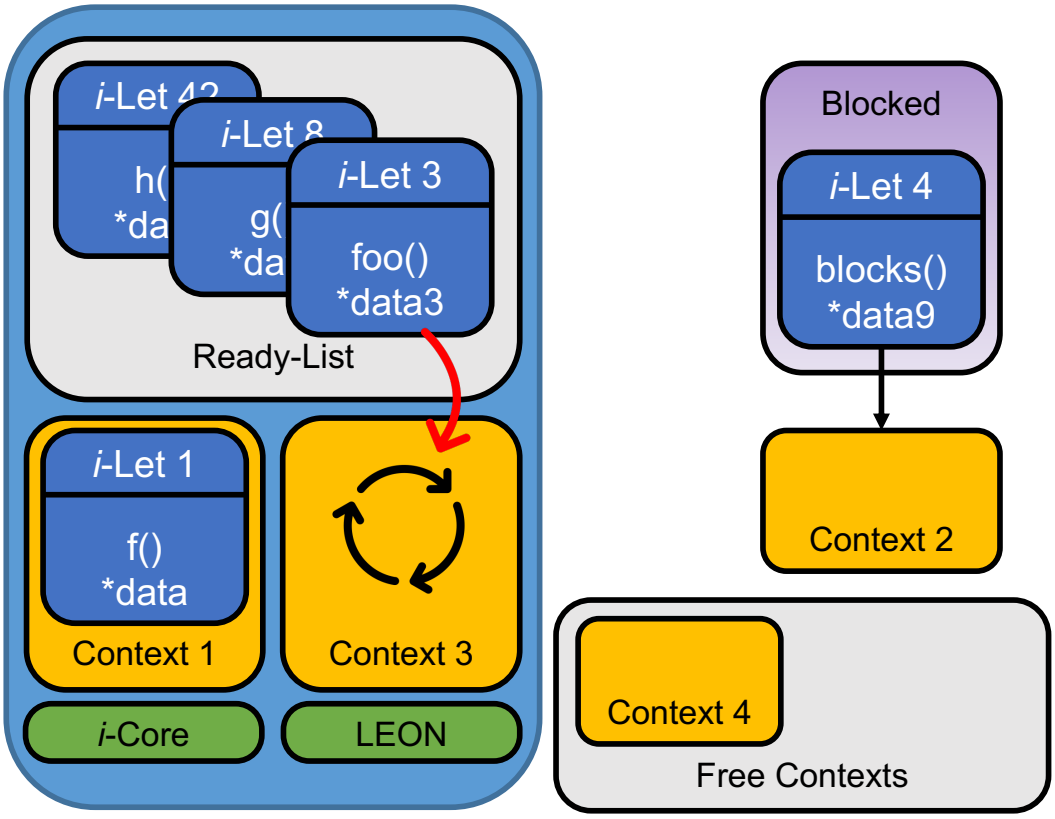


Figure 17.2.: OctoPOS *i*-let execution scheme. In OctoPOS, *i*-lets typically run to completion once they have been scheduled for execution on a core. This allows for a very light-weight multi-threading, as all the data structures (called the context) associated with the thread of execution may be reused once an *i*-let terminates. A Context is only bound to its *i*-let if the latter encounters a construct that requires it to interrupt its execution, for example a lock. Then, another context is scheduled for execution. At a later point, the *i*-let and its context may be brought to execution again, possibly on a different core. (Adapted from Pöppl (2017))

```
1  // Bring input data into a format understood by i-Core
2  callBufferRaw(0) = topHuRaw(col);
3  callBufferRaw(1) = topHRaw(col);
4  callBufferRaw(2) = botHuRaw(col);
5  callBufferRaw(3) = botHRaw(col);
6  callBufferRaw(4) = botBRaw(col);
7  callBufferRaw(5) = topBRaw(col);
8
9  // Execute i-Core special instruction
10 ICore.fwave(callBuffer, resultBuffer);
11
12 // Retrieve and apply result
13 topHUpRaw(col)  += dyInv * resultBufferRaw(0);
14 botHUpRaw(col)  += dyInv * resultBufferRaw(1);
15 topHuUpRaw(col) += dyInv * resultBufferRaw(2);
16 botHuUpRaw(col) += dyInv * resultBufferRaw(3);
17 maxWaveSpeed = Math.max(maxWaveSpeed, resultBufferRaw(4));
```

Figure 17.3.: Call site of the f-Wave CI. The special instruction requires its parameters to be in two contiguous buffers.

tile may be viewed as a shared memory multi-cores system with a heterogeneous ISA between the different cores. The set of available instructions for the LEON 3 core does not contain the CIs that may be configured on the *i*-Core. If an *i*-let expecting to be able to use a previously configured CI is brought to execution on the LEON 3 core instead of the *i*-Core after encountering a blocking operation, and it encounters a CI, it would generate an *illegal instruction trap* that essentially leads to a fatal system failure. On the other hand, *i*-lets that only contain the "conventional" SPARC instructions may be scheduled on any core (including the *i*-Core).

This obstacle was overcome through the use of *scheduling domains*. Scheduling domains mark parts of the hardware (cores) as eligible as scheduling targets for *i*-lets. To effectively use the *i*-Core tile in our application we chose to create a scheduling domain that contains all the invaded *i*-Cores on a tile. *i*-lets requiring the *i*-Core may now be added to a team that requires the *i*-Core's scheduling domain. The team concept is based on work presented by Cheriton et al. (1979), with the additional ability to reassign *i*-lets dynamically to another team if needed. Thus, when it is clear that an *i*-let will require the instructions provided by the *i*-Core, it may be assigned to the *i*-Core-only team. This ensures that the *i*-let will not encounter any non-standard instructions, and therefore avoid any illegal instruction traps.

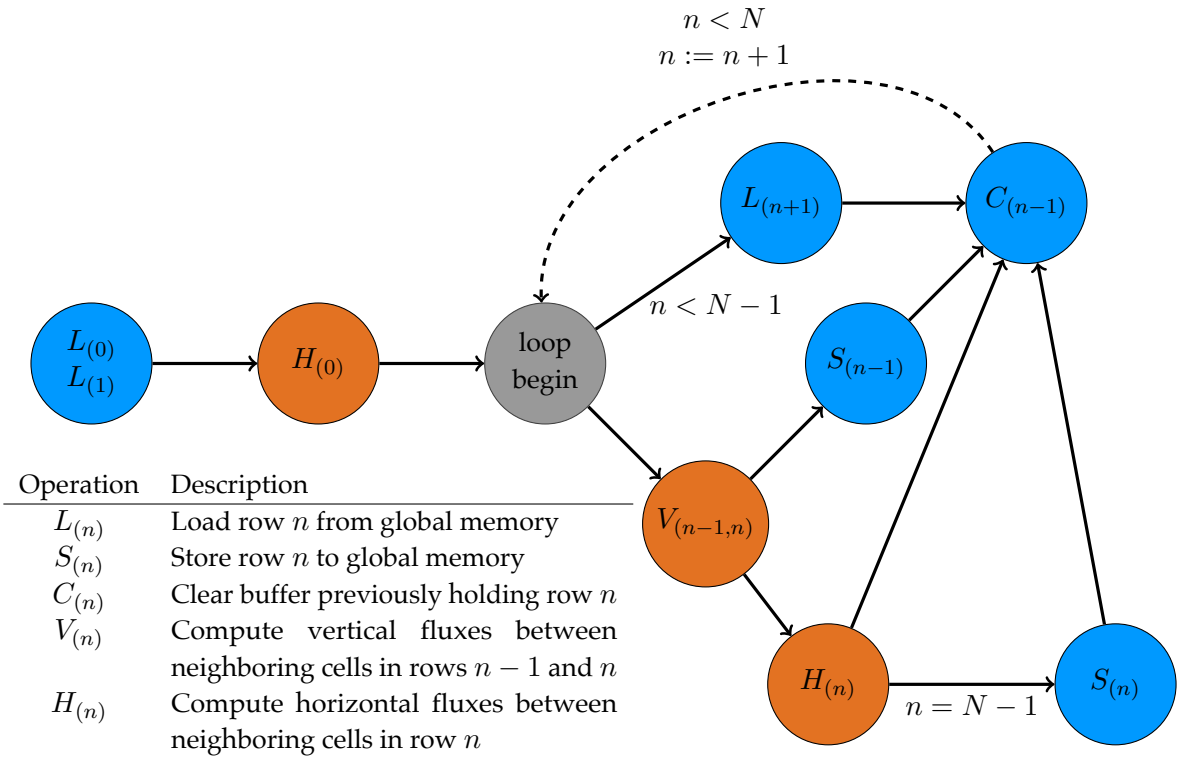
The X10 Runtime did not need major modifications. The main challenge was bridging the gap between high-level X10 constructs, and the low-level constructs expected by the *i*-Core. The CIs used on the *i*-Core are treated like any other assembly instruction. X10 does not provide any possibility to provide inline assembly instructions, which prevents the application developer from accessing the instruction directly. Furthermore, the CI expects its data arguments as plain C-style arrays.

However, the lowest-level layer of abstraction available in X10, the `IndexedMemoryChunk[T]`, still contains additional book-keeping information such as the array length, and another (opaque) pointer to the actual memory segment used to store its values. To bridge the divide, the runtime introduces wrapper functions for the CI that are implemented as C functions. Inside the function, the X10 arrays are unwrapped, and the f-Wave CI is called using inline assembly. This enables the application to call the CI as it would any other function, in a fashion similar to the one used by vector intrinsics (as depicted in Figure 17.3). However, each function call introduces additional overhead, and for a computational hotspot such as the f-Wave solver call, this may slow down application execution considerably. Furthermore, all the intermediate calls to copy the values for the calculation into the buffers mandated to be used by the *i*-Core, essentially, amounts to additional function calls. The X10 compiler mitigates this through aggressive inlining of small functions. This is made possible by viewing the entire program, including the runtime, as a single compilation unit (*whole-world-compilation*). Inlining is straightforward as the function declaration is always accessible in the call site. In turn, function inlining enables other optimizations that work within the scope of a single function. For our case, this eliminated the overhead of the wrappers introduced both by X10 and the CI wrapper to a point where the CI may directly access the respective raw arrays of the buffer objects.

17.3. Changes in SWE-X10

SWE-X10 only required very minor code changes to make it compatible with the APIs exposed by the invasive X10 compiler. Therefore, most of our effort was spent optimizing the performance on the *i*-Core. The application's computational hotspot is the calculation of fluxes between cell boundaries: $\mathcal{A}^{\pm}\Delta Q^{(\mathrm{n})}_{i\pm\frac{1}{2},j}$ and $\mathcal{B}^{\pm}\Delta Q^{(\mathrm{n})}_{i,j\pm\frac{1}{2}}$, in section 14.2. The f-Wave solver, as described in subsection 14.3.1, is one of the approximated Riemann solvers available to compute these net updates. The CI implemented for the *i*-Core may be directly used as a drop-in replacement for the X10 implementation of the f-Wave solver. One simply needs to copy all the necessary values for the solver into a contiguous buffer, call the `fwave` instruction, and finally retrieve the result. However, this way, the high-bandwidth connection to the *i*-Core TLM is not utilized, as data is retrieved from the global memory.

In order to fully utilize the *i*-Core, we created a specialized `ICorePatchIterator` class that uses the *i*-Core TLM to buffer the unknown values. However, the size of the TLM is insufficient to hold the entire patch at the same time. Instead, we preload the data on a per-row basis into the *i*-Core TLM using a triple buffering scheme with a *previous*, a *current* and a *next* row. The resulting *i*-let graph of the X10 implementation is given in Figure 17.4. *i*-lets, depicted in the graph as nodes, depend on the completion of the all *i*-lets that they are connected with through an incoming edge in the graph. In the beginning of the operation, the first two rows of the patch ($L_{(0)}$ and $L_{(1)}$) are synchronously loaded into the *i*-Core TLM, and assigned the *previous* and the *current* pointer. Next, we perform the calculation of the horizontal fluxes for the row stored in *previous* ($H_{(0)}$). Now, we may perform the inner part of the iteration in a loop for rows 1 to N, with N being the number of rows in the patch. In each iteration n, we begin by initiating an asynchronous load of the *next* row ($L_{(n+1)}$) into the TLM and perform the computation of the vertical fluxes ($V_{(n-1,n)}$ between the *previous* and the *current* row. Once the flux calculation is completed, the



Operation	Description
$L_{(n)}$	Load row n from global memory
$S_{(n)}$	Store row n to global memory
$C_{(n)}$	Clear buffer previously holding row n
$V_{(n)}$	Compute vertical fluxes between neighboring cells in rows $n-1$ and n
$H_{(n)}$	Compute horizontal fluxes between neighboring cells in row n

Figure 17.4.: *i*-let graph for the *i*-Core patch calculator. The graph depicts *i*-lets as nodes, and their dependencies as edges. Nodes that are not (transitively) connected may be executed in parallel. Nodes performing I/O operations are depicted in blue, while nodes performing a computation are depicted in orange. Edges annotated with a condition are only taken if the condition is met. (Pöppl et al., 2018)

Table 17.2.: Execution time and resource utilization results for the f-Wave solver kernel executed in software (without floating-point unit (FPU), with "lite" FPU and "high-performance" FPU from Gaisler) compared to `fwave` CI on the *i*-Core. Results were obtained using GRLIB on a Xilinx VC707 board (Virtex-7 FPGA) at 75 MHz. (Pöppl et al., 2018)

	LEON3 – no FPU		**LEON3 + FPU-lite**		**LEON3 + FPU-HP**		***i*-Core**	
Execution	[μs]	Speedup	[μs]	Speedup	[μs]	Speedup	[μs]	Speedup
Time[1]	183.6	1	14.9	12.3	9.8	18.7	1.3	141
Resource	LUTs	DSPs	LUTs	DSPs	LUTs	DSPs	LUTs	DSPs
Utilization	9,103	4	12,756	4	23,949	20	37,658	24
Resource	$\frac{\text{FLOPS}}{\text{LUTs}}$	$\frac{\text{FLOPS}}{\text{DSPs}}$	$\frac{\text{FLOPS}}{\text{LUTs}}$	$\frac{\text{FLOPS}}{\text{DSPs}}$	$\frac{\text{FLOPS}}{\text{LUTs}}$	$\frac{\text{FLOPS}}{\text{DSPs}}$	$\frac{\text{FLOPS}}{\text{LUTs}}$	$\frac{\text{FLOPS}}{\text{DSPs}}$
Efficiency	27.8	73,529	248	906,040	203.8	275,510	780.3	1,730,769

[1] Average over 1024 measurements

previous row is no longer needed, and its result may be asynchronously stored ($S_{(n-1)}$) back in the global memory. At the same time, we may compute the horizontal fluxes on the *current* row ($H_{(n)}$). If the current iteration is the last one, the contents of the last row need to be written back to the global memory as well ($S_{(n)}$). Finally, once both horizontal and vertical flux computations are performed, the contents of the row ($C_{(n-1)}$) are cleared from the buffer and the pointers are changed, so that the *current* row becomes the *previous* one, and the *next* becomes the *current* one.

17.4. Results

We evaluated the performance benefits in two parts. First, we classified the performance of the *i*-Core CI alone, and then compared it to computing the solution using the different floating-point implementations available for the LEON 3 core. Therein, we compared performance benefits and FPGA resource utilization. Afterwards we evaluated the performance of computing an entire simulation step of a whole patch on the *i*-Core against the solution that performs the computation on the LEON 3 core using its high-performance floating-point unit (HP-FPU).

Table 17.2 shows execution time and resource utilization results for the f-Wave solver kernel executed on a standard LEON3 (with different variants of floating-point support) in comparison with an execution on the *i*-Core. Compared to a standard LEON 3 core with HP-FPU (fastest floating-point support variant that also utilizes most resources), the *i*-Core is 7.5 times faster and 3.8 times more efficient in the use of lookup tables (LUTs) and digital signal processors (DSPs) on the Xilinx Virtex-7 (floating-point operations per second / LUTs). For the evaluation of the performance of the entire iteration, we chose a patch size of 60×60 grid cells. This results in ≈ 7200 flux update computations (and calls to the `fwave` CI) using the f-Wave solver (or f-Wave CI) Table 17.3 shows the execution time of one iteration of the patch calculator. The performance baseline is program execution on the LEON 3 with HP-FPU utilizing global memory. We show the average time taken to process one patch, and the speedup compared to processing the patch on a standard LEON 3 core without memory optimizations. Buffering data in the *i*-Core TLM while using the `ICorePatchIterator`'s triple buffering scheme and the HP-FPU results in a speedup of $1.75\times$. Execution on the *i*-Core CI with use of only global memory (without triple buffering)

Table 17.3.: Patch calculator execution time on the LEON3 (with FPU-HP) compared to execution time on the *i*-Core, with data in global DDR RAM or buffering in the tile-local memory (TLM). Results were obtained using the InvasIC Hardware Prototype on a Synopsis CHIPit system consisting of four Xilinx XC5VLX330 (Virtex-5 FPGA) at 25 MHz. (Pöppl et al., 2018)

	LEON3 – global		**LEON3 – TLM**		***i*-Core – global**		***i*-Core – TLM**	
Execution	[ms]	Speedup	[ms]	Speedup	[ms]	Speedup	[ms]	Speedup
Time[1]	2049	1	1169	1.75	1017	2.01	425	4.82

[1] Average over 200 measurements

speeds up the computation by a factor of 2. Both optimizations combined alleviate the memory bottleneck for the *i*-Core. Thus, we achieved a speedup of $4.82\times$ in total.

17.5. Discussion

The use of reconfigurable fabric for the acceleration of the approximate solution of Riemann problems serves to demonstrate the value of the interdisciplinary work performed in the context of InvasIC. Techniques known in the embedded computing domain, such as the use of reconfigurable processors, or being able to control the entire compute stack from the hardware level up to the software, may also serve to accelerate HPC applications. Using the *i*-Core, we accelerated the computational hotspot of SWE-X10 by a factor of 4.82 over the baseline solution using only the LEON 3's high-performance floating-point unit. At the same time, the resources of the reconfigurable fabric were used more efficiently (in terms of LUTs and DSPs). The software stack described in this chapter was used to demonstrate the overall system integration of the invasive technology stack at the review for the third funding phase of invasive computing.

18. Pond, An Actor-UPC++ Proxy Application

In chapter 11, I introduced an actor library for more classical HPC environments. The library uses UPC++ as a communication backend. Analogously to ActorX10 and SWE-X10, I developed a sample application to demonstrate the practical use of the model for "classical" HPC environments.

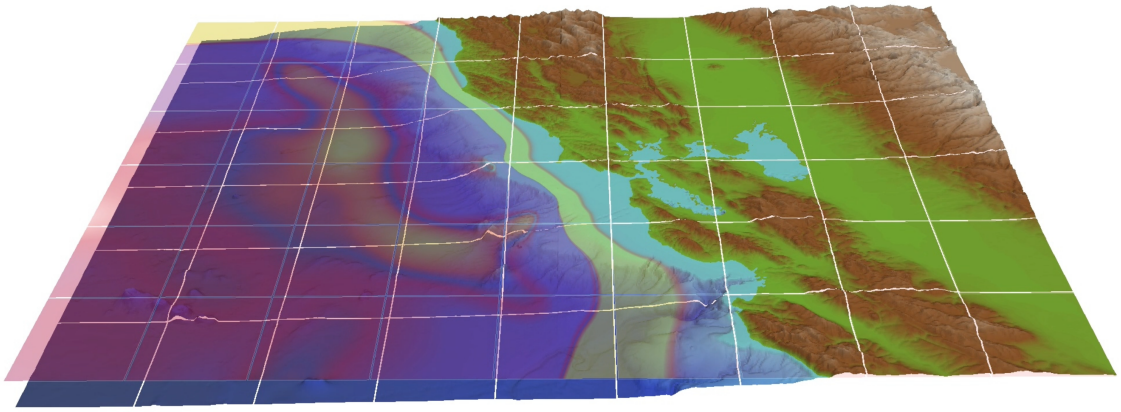


Figure 18.1.: Hypothetical tsunami scenario on the Pacific Coast of the United States. The tsunami was simulated using Pond with 36 actors on a single node.

Pond (see Figure 18.1 for an example scenario simulated using the application) uses an approach similar to the basic actor-based version of SWE-X10 discussed in section 16.2. In this chapter, I will briefly discuss the implementation of Pond, and then compare it to its two predecessors, ActorX10-based SWE-X10 and SWE, the original tsunami teaching code that uses MPI and OpenMP to implement a BSP approach for parallelization. I will show that use of this library yields a significantly higher performance in both a weak and a strong scaling test, as well as a significantly better performance with a lower per-core computational load compared to SWE-X10. Pond also exhibits a clear performance benefit over SWE. This chapter is based on my experiences implementing Pond and Actor-UPC++ during and after my research stay at the Lawrence Berkeley National Laboratory, and contains results I have previously presented there and in Pöppl, Bader, and Baden (2019).

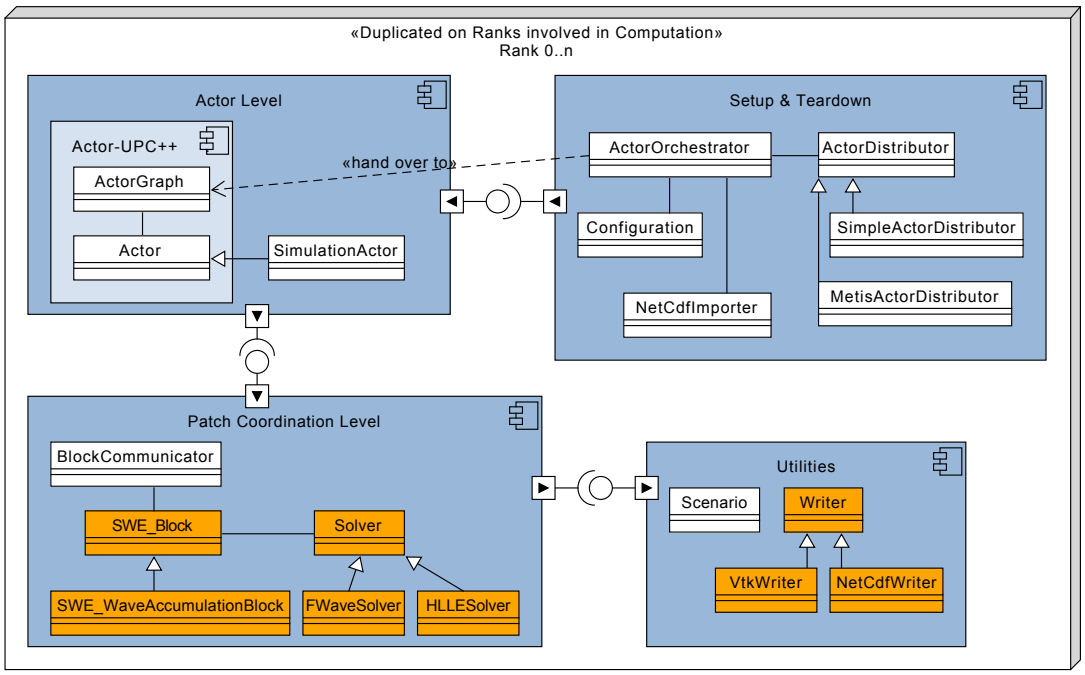


Figure 18.2.: Component Diagram for Pond. Classes drawn in orange are taken from SWE. Classes drawn in white are modified, or code specifically written for Pond.

18.1. Implementation

The basic system architecture of Pond is the same as SWE-X10's. However, as it was implemented in C++, it was possible to reuse components from SWE directly. Specifically, I used the `SWE_Block` class and the `SWE_WaveAccumulationBlock` class without modification. The code exhibits the same layered architecture as SWE-X10, however, there are some structural differences stemming from the different control model imposed by Actor-UPC++. Here, there is an actor graph instance on each rank. Following this, the creation of the actor graph is performed in a decentralized fashion. Unlike in SWE-X10, where the actors are created and wired on the root place, in Pond each rank determines the set of actors it will contain, and then creates them directly. For the simulation to be executed efficiently, a good load balancing of the actor graph with minimal communication is key. Unstructured grid codes often use a dual-view approach to view the grid as a graph and then use graph partitioning techniques such as the *Metis* library, proposed and developed by Karypis and Kumar (1998). I use Metis directly on the actor graph, and thus obtain a partitioning of the graph with balanced load and minimal edge-cut (to reduce communication between different ranks). The library was configured to produce contiguous partitions with a maximum load imbalance of $\pm 10\%$. The channels are created using the facilities provided by Actor-UPC++. They are always created from the rank of the actor with the outgoing port. For the communication with the `SWE_Block`, I implemented the `BlockCommunicator` class. It extracts the data from the token (`std::vector<float>` of size $3n_{\text{ghost}}$ containing the ghost layer data for the water height, and the momenta in the two spacial directions), and copies it into the data structure provided by the `SWE_Block`.

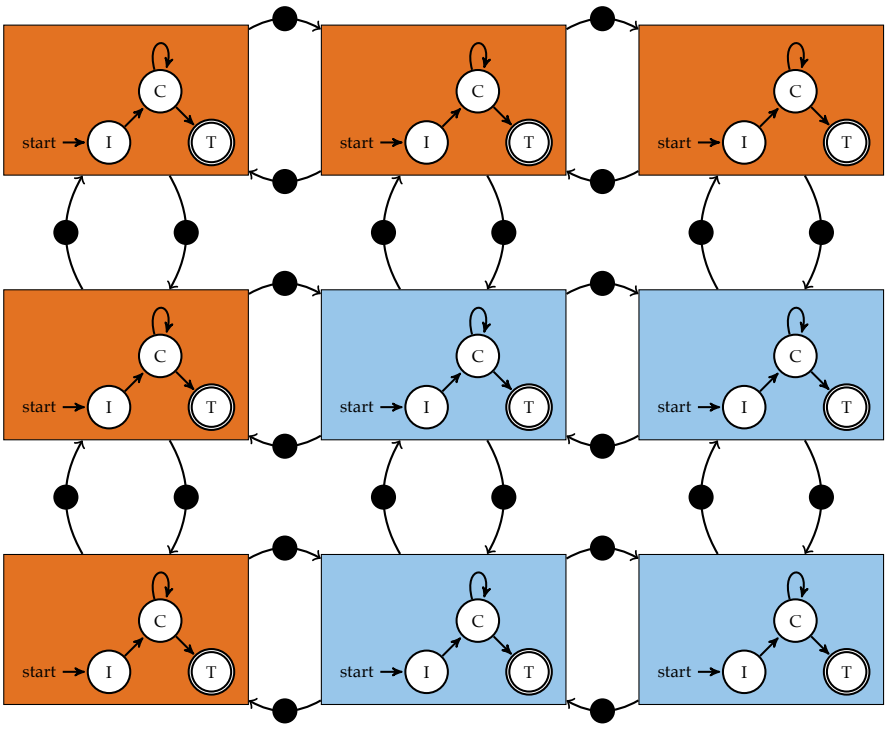


Figure 18.3.: Sample Actor Graph for Pond with a simulation being performed by nine actors distributed onto two different ranks. (Pöppl, Bader, and Baden, 2019)

The actor graph of Pond is similar to the one of SWE-X10, but, as there is currently no support for the advanced features of SWE-X10 and there is only need for a single channel in each direction for data exchange, the control channels are omitted. Figure 18.3 depicts the actor graph for a simple simulation run with nine actors distributed onto two different ranks. The structure of the actors is defined, similarly to the actors described in section 16.2, as:

$$G_a^{\text{Pond}} = (A_{\text{Pond}}, C_{\text{Pond}}) \tag{18.1}$$

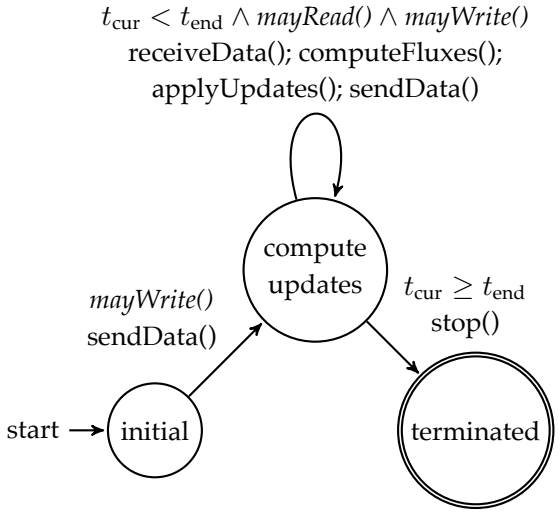
$$A_{\text{Pond}} = \left\{ a_{i,j} \middle| \, 0 \leqslant i < n_x \wedge 0 \leqslant j < n_y \right\}$$

$$C_{\text{Pond}} = \left\{ c_{a_{i,j},a_{i',j'}} \middle| \, \left(i = i' \wedge |j - j'| = 1 \right) \vee \left(|i - i'| = 1 \wedge j = j' \right) \right\},$$

$$a_{i,j} = \left(\text{ID}, r, I_{a_{i,j}}, O_{a_{i,j}}, F_a, R_a \right) \tag{18.2}$$

$$I_{a_{i,j}} = \left\{ ip_{i',j'} | c_{a_{i',j'},a_{i,j}} \in C_{\text{Pond}} \right\}$$

$$O_{a_{i,j}} = \left\{ op_{i',j'} | c_{a_{i,j},a_{i',j'}} \in C_{\text{Pond}} \right\}.$$



As in SWE-X10, the basic state is the compute state which is reached after the initial data is sent. Actors perform self-transitions to compute new steps whenever sufficient data from the neighbors

is available. This is done until their simulation time has reached the specified end time of the simulation. Then the actors terminate.

Pond and Actor-MPI In addition to the implementation using Actor-UPC++, there is also a version of Pond that uses Actor-MPI. It has been developed by Macedo Miguel during his master's thesis (Macedo Miguel, 2019). The solution is very similar to the one described above, except for some very minor differences mandated by differences in the interface of Actor-MPI. The interface of the simulation actor is identical to the one using Actor-UPC++, and the execution works in the same way. Only during the setup and tear-down, the application has to use some MPI functionality directly, in order to distribute actors and to collect the performance results.

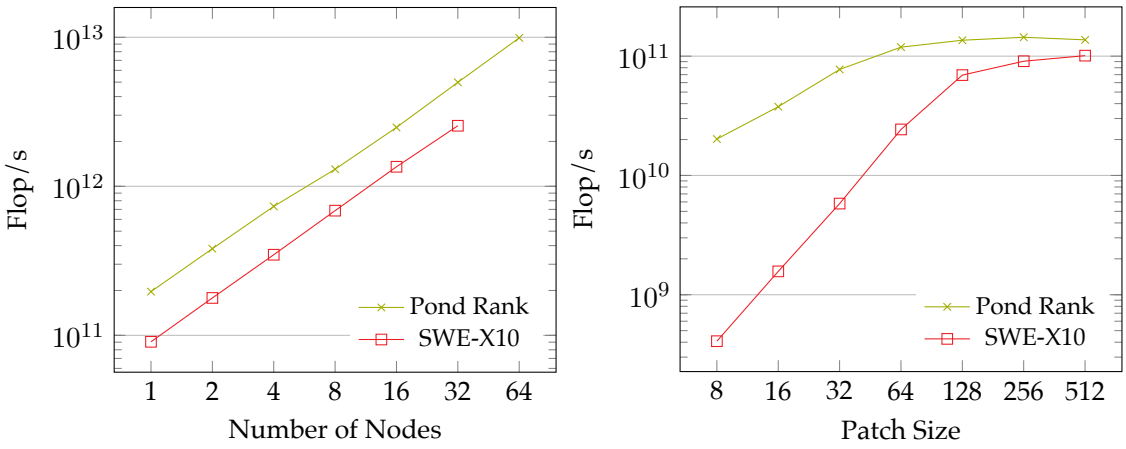
18.2. Evaluation of Pond and Actor-UPC++

I performed comparisons between the different implementations discussed in the previous sections on the Cori Cluster of NERSC (NERSC, 2020). Cori consists of two partitions, one with Intel Knights Landing many-core processors, and one with Intel Haswell multi-core processors. The Knights Landing partition employs 9688 nodes with a single-socket Intel Xeon Phi 7250 and a combined theoretical peak performance of 29.5 PFlop/s. Each Xeon Phi is equipped with 68 cores clocked at 1.4 GHz, yielding a theoretical peak performance of 6 TFlop/s (SP) per node. The peak memory bandwidth is 102 GB/s for the off-chip DDR memory, and around 460 GB/s for the on-chip MCDRAM. For my tests on this partition, I used the default configuration of the KNL node, and the Intel Programming Environment. The Haswell partition contains 2388 nodes with a dual-socket Intel Xeon E5-2698v3, and a combined peak performance of 2.81 PFlop/s. Each Xeon is equipped with 16 cores clocked at 2.3 GHz with a combined theoretical peak performance of 2.4 TFlop/s (SP) per node. The Haswell nodes have a peak memory bandwith of about 100 GB/s. In my tests, I used the default configuration of the KNL nodes, and the Intel Programming Environment. Optimizations were enabled (`O3`) for both partitions, and the iteration over the patch was therefore automatically vectorized. On the Haswell partition, vectorization was done using AVX-2, while on the Knights Landing partition AVX-512 was used.

In the tests on the Knights Landing partition, I compared Pond using Actor-UPC++ with its three execution strategies (as described in section 11.2) to SWE-X10 and ActorX10, and finally to BSP-based SWE. For the tests on the Haswell partition, I compared Pond using the rank-based execution strategy of Actor-UPC++ to SWE-X10. As described before, all these applications use the same numerical scheme, and for this evaluation, I configured all of them to use the HLLE solver. As with the evaluation in the previous chapter, I used a radial dam break scenario as the test scenario, as it is easily scaled to any size. Time was measured from the start of the actor graph until all actors are terminated for Pond and SWE-X10. For SWE, the execution time was measured from the beginning of the first to the end of the last iteration. File I/O was disabled for all configurations. Performance was measured in Flop/s, and determined based on the observed execution time, and the number of patch updates in Pond and SWE-X10, or the number of iterations in SWE. An overview of the configurations used for the evaluation is shown in Table 18.1 and Table 18.2.

Table 18.1.: Configurations used for the scaling tests on Cori (Haswell Partition)

Execution Type	Symbol	Description
Pond Rank	×	Pond using the rank-based execution strategy. One Rank per logical core, and two actors per rank
SWE-X10	□	SWE-X10 using ActorX10. Two Places per node (one per socket), and two actors per logical core



(a) Weak Scaling Test

(b) Overhead Analysis

Figure 18.4.: Performance comparison between SWE-X10 and Pond on Cori (Haswell Partition). The weak scaling test was performed using a working set of 4096×4096 cells per node. The overhead analysis was performed on a single node, using 256 actors in each configuration. Result from Pöppl (2018)

The first test I performed is a weak scaling test of SWE-X10 and Pond using the rank-based and the thread-based execution strategies. The test was performed to evaluate the scaling behavior of the UPC++-based implementation of both the library and the proxy application. Each node computes a working set of 4096×4096 grid cells, distributed onto actors with 256×256 cells each. Its results, depicted in Figure 18.4a, show a picture similar to the one reported previously in section 16.5. SWE-X10 scales linearly with the number of nodes in the computation. *Pond Rank* outperforms SWE-X10 by a factor of two for all node sizes. As the computations performed by both applications are identical, and they were both compiled using the same C++ compiler to ensure that the same optimizations are applied to the local computations, the data suggests that SWE-X10 exhibits a higher library overhead compared to Pond's ActorX10.

To evaluate this, I performed a second test on a single Haswell node. In this test, the simulation was performed using a fixed actor graph of 16×16 actors. In SWE-X10, they were distributed onto two places (one per socket), and in *Pond Rank*, there was one rank per logical CPU core. Simulations were performed for total grid sizes ranging from 128×128 (8×8 cells per actor) up to 8192×8192 (256×256 cells per actor). The results are depicted in Figure 18.4b. SWE-X10 needs relatively large patches to reach an optimal performance. The performance plateau of circa 100 GFlop/s is only reached using a patch size of 256×256 grid cells or larger. For smaller patch sizes such as 64×64, the performance is significantly diminished, and for the smallest

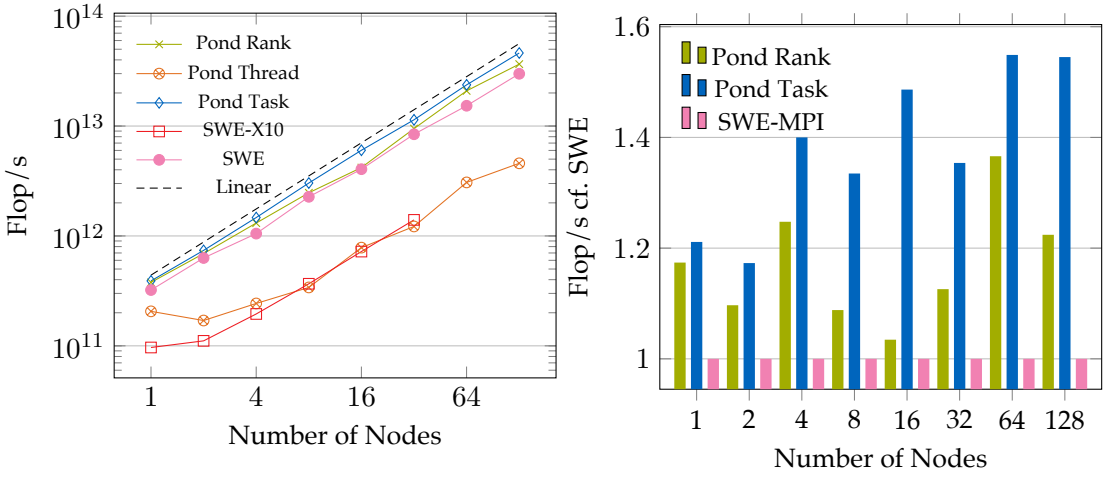
Table 18.2.: Configurations used for the scaling tests on Cori (Knights Landing Partition)

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

patches, the coordination overheads dominates the computation. The smallest patch size achieved a performance that is only at $\approx 1\%$ of the original performance. *Pond Rank* on the other hand reaches a higher plateau of about $140\,\mathrm{GFlop/s}$ already at a smaller patch size of 64×64. For smaller patch sizes, the library overhead becomes more significant. At a patch size of 32×32, the performance is still acceptable, at $78\,\mathrm{GFlop/s}$. Below, the library overhead dominates the calculation, and the performance falls below 25% of the best observed performance.

Aside from the tests performed on the Haswell nodes, I also evaluated the software packages using Cori's Knights Landing partition. The Xeon Phis used there rely on a high number of relatively simple CPU cores[1] with added support for 512-Bit-wide vector instructions (AVX-512). Compared to the Haswell cores, the cores of the Knights Landing architecture have a simpler control logic, but stronger floating-point units. This suggests that the advantage of Pond will be more pronounced in tests performed on that architecture, as the time spent on computing (where the Knights Landing is comparatively faster) decreases while coordination time increases (due to the simpler architecture of the Knights Landing core). Furthermore, the nodes of the Knights Landing partition have twice the number of cores (on a single socket), and $2.5\times$ the peak performance per node. The weak scaling test was performed with a per-node workload of 4096×4096 grid cells per node, and 256×256 cells per logical core. In addition to *Pond Rank* and SWE-X10, I also added *Pond Rank*, *Pond Task* and SWE to the evaluation. Their respective application configuration is described in Table 18.2. Results are depicted in Figure 18.5. SWE scales linearly with the number of nodes used in the computation. SWE-X10 exhibits the lowest performance, and is about an order of magnitude slower than SWE. This is very likely due to the comparatively higher library overhead which is exacerbated further by the Knights Landing μArch. Furthermore, the largest two configurations with 64 and 128 nodes failed to complete within the allocated time for the computation. Cancellation occurred during actor distribution, which is performed sequentially in SWE-X10. Unfortunately, in ActorX10 actor migration has not been not parallelized, and with the number of actors (> 16000) and channels (> 64000) needed for these configurations, this proved infeasible. Pond's performance depends on the execution

[1] In the generation used on Cori, the individual cores are based on the Intel Atom "Silvermont" μArch (Anthony, 2013).



(a) Weak Scaling

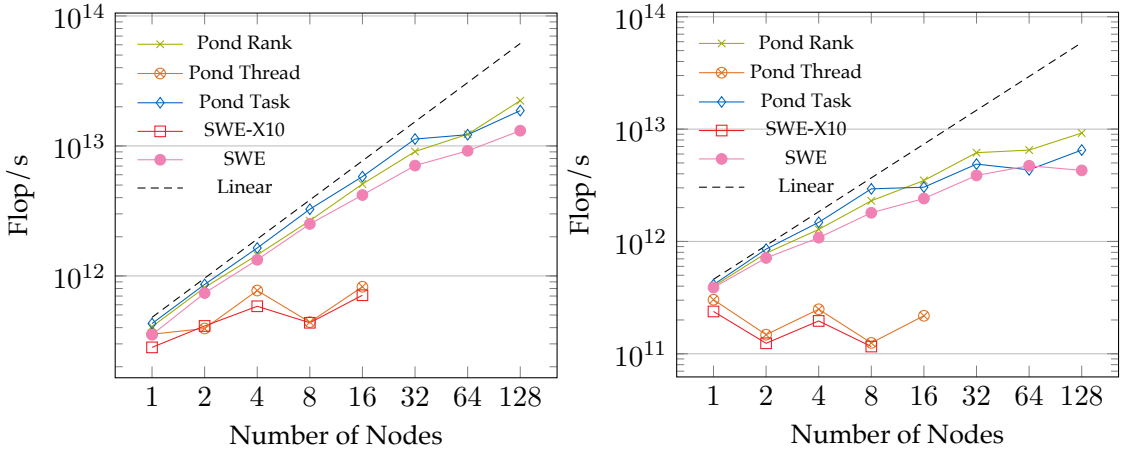
(b) Detailed comparison

Figure 18.5.: Weak Scaling test on Cori (Knights Landing Partition) with 4096×4096 cells per node. The largest two configurations of SWE-X10 did not run to completion. (Results from Pöppl, Bader, and Baden (2019))

strategy used in Actor-UPC++: The performance of *Pond Thread* proved to be a disappointment, with a performance similar to the one of SWE-X10. *Pond Rank* manages to consistently outperform SWE, with roughly 20% higher performance compared to SWE for the largest run with 128 nodes. *Pond Task* performs best in this test. It is on average 38% faster than SWE, and exhibits a 50% higher performance than SWE for the largest run.

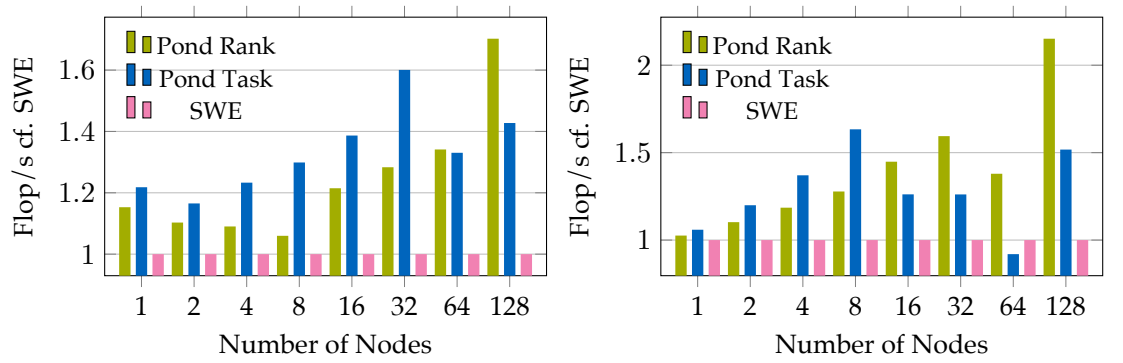
In addition to the weak scaling tests, I also performed strong scaling tests to explore the scalability limit of SWE-X10 and especially Pond on Knights Landing. In the first test, I set the size of the simulation to 16384×16384 grid cells. This led to a patch sizes ranging from 512×512 grid cells for the single node run down to 4096×4096 for the run with 128 nodes. In the second test, the simulation size was set to 8192×8192, yielding patch sizes from 256×256 to 32×32. For Pond and SWE-X10, the patch size for the individual actor was set such that it is ensured that each logical core has at least one actor. If it was not possible to divide the patch size evenly, the patch size was halved, and the smaller patches were distributed. In SWE the OpenMP parallel for-loop handles the distribution of the node-local patch data to the cores. In both cases, the per-node work load remained the same.

The results of the test, depicted in Figure 18.6, suggest similar conclusions, in line with previous findings. Performance of SWE degenerates gradually with the shrinking working set of each core. The actor-based solutions are currently limited to quadratic patches, leading to more abrupt drops in performance, as smaller patch sizes, needed to evenly distribute the grid, lead to more actors and therefore more coordination overhead. As before, the performance of SWE-X10 is acceptable for the largest patch sizes on the single node, but then degenerates, and is dominated by the library overhead. For patch sizes smaller than 512×512 grid cells, there were no benefits to increasing the number of ranks, as the additional compute resources are used up completely by the resulting additional overhead. The same behavior was observed with *Pond Thread*. This may be explained



(a) 16384 × 16384 grid cells

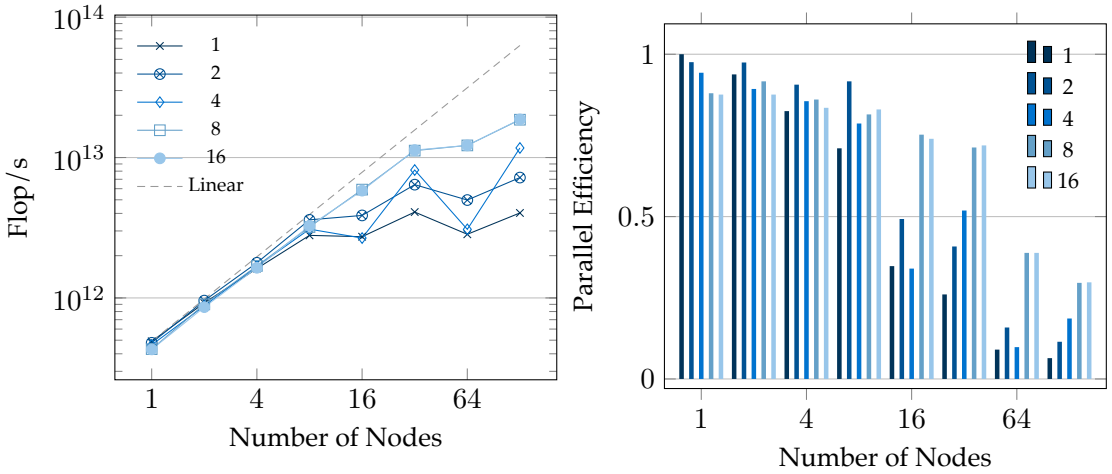
(b) 8192 × 8192 grid cells



(c) Detailed comparison: 16384 × 16384

(d) Detailed comparison: 8192 × 8192

Figure 18.6.: Strong Scaling test on Cori (Knights Landing Partition). See Table 18.2 for the configurations. In Figure 18.6c and Figure 18.6d, the relative performance compared to the run of SWE with the corresponding number of nodes is displayed. (Results from Pöppl, Bader, and Baden (2019))



(a) Absolute Performance

(b) Parallel Efficiency

Figure 18.7.: Strong scaling test evaluating different configurations of the task-based execution strategy of Pond on a 16384×16384 grid. Each configuration represents a different amount of UPC++ Ranks per physical node. The test was performed on Cori (Knights Landing Partition). (Result from Pöppl, Bader, and Baden (2019))

by looking at the implementation of their run loop. In both ActorX10 and Actor-UPC++ using the thread-based execution strategy, there is a one-to-one mapping of resources (operating system threads in ActorX10, and activities in X10) to actors. In ActorX10, actors are sent to sleep if there's nothing for them to do, and in Actor-UPC++, they will keep polling for progress. Furthermore, there is additional overhead for the communication between the threads. The major problem is the potentially large number of threads compared to available system resources. The operating system is not aware of which processes should be computing and which are only waiting for updates, and is making its scheduling decisions oblivious of that. The end result is the same in both cases: they perform worse for larger amounts of actors per core, requiring larger workloads per actor, and making smaller-grained parallelism infeasible.

In the other execution strategies of Actor-UPC++, this is not a problem. When Actor-UPC++ is configured to use the rank-based execution strategy, each logical core has its own rank, with multiple actors. Each rank is responsible for its own progress, and only actors that actually get triggered consume resources. Furthermore, the entire process is sequential, without the need for mutual exclusion and thread synchronization, and therefore, the sequential UPC++ backend, that forgoes these measures, is used. When Actor-UPC++ is configured to use the task-based execution strategy, the act invocations for actors that have been triggered are distributed in the form of OpenMP tasks onto worker threads, and then asynchronously executed there. Communication is performed in a centralized fashion by the master thread (OpenMP master thread as well as UPC++ master persona) of the computation. The master thread continuously polls the UPC++ runtime for incoming RPCs and executes them. If there is too much communication traffic, this may be a bottleneck, and lead to worker threads being idle while the master thread is busy handling the incoming communication. One can work around this problem by adding multiple ranks per node.

I explored this effect in a separate strong scaling test, performed again with 16384×16384 grid cells through comparison of the different number of UPC++ ranks per node. The results are shown in Figure 18.7. For the Xeon Phi, I found that configurations using eight or sixteen ranks per node performed better than the configurations with fewer ranks per node (one, two or four ranks per node) at the scaling limit of the application, with a small workload per actor. For runs with a smaller number of nodes, the additional overhead caused by the multiple communication threads causes a lower performance compared to the solutions with a lower number of ranks per node (until eight nodes). The UPC++ system design fixes these parameters at application startup, and hence it cannot be adjusted by the library at application startup time. Finding the best configuration for a given application and target architecture hence remains a burden of the application developer.

18.3. Evaluation of Pond and Actor-MPI

In addition to the tests performed on Actor-UPC++ in the previous section, I also tested the performance of Actor-MPI using the corresponding Pond implementation. Similarly to the performance tests in section 15.3, the tests were performed on the CoolMUC 2 cluster of the LRZ (LRZ, 2020a). The LRZ also hosts the CoolMUC 3[2] based on Xeon Phi (Knights Landing) nodes. That cluster would have had a very similar hardware setup to the Knights Landing partition of Cori. However, it unfortunately uses the Intel OmniPath interconnect which is not supported by the GASNet-EX framework underlying UPC++. The experiments were performed by Macedo Miguel in the context of his master's thesis using the Intel Compiler 2019, Intel MPI 2019 and UPC++ 2019.03 (Macedo Miguel, 2019).

Two tests were performed, a strong scaling test and a weak scaling test. As with the previous tests, a radial dam break scenario was used. Each node received a working set of 4096×2048 grid cells, distributed onto 32 actors with a patch size of 512×512 grid cells each. The resulting grid sizes range from the aforementioned 4096×2048 grid cells for the single node run up to 16384×16384 grid cells for the run with 32 nodes. In the strong scaling test, the domain size is fixed at 16384×16384 grid cells. With the same number of actors as in the weak scaling test, this leads to patch sizes ranging from 2048×2048 grid cells for the single node run down to 512×512 grid cells for the run with 32 nodes.

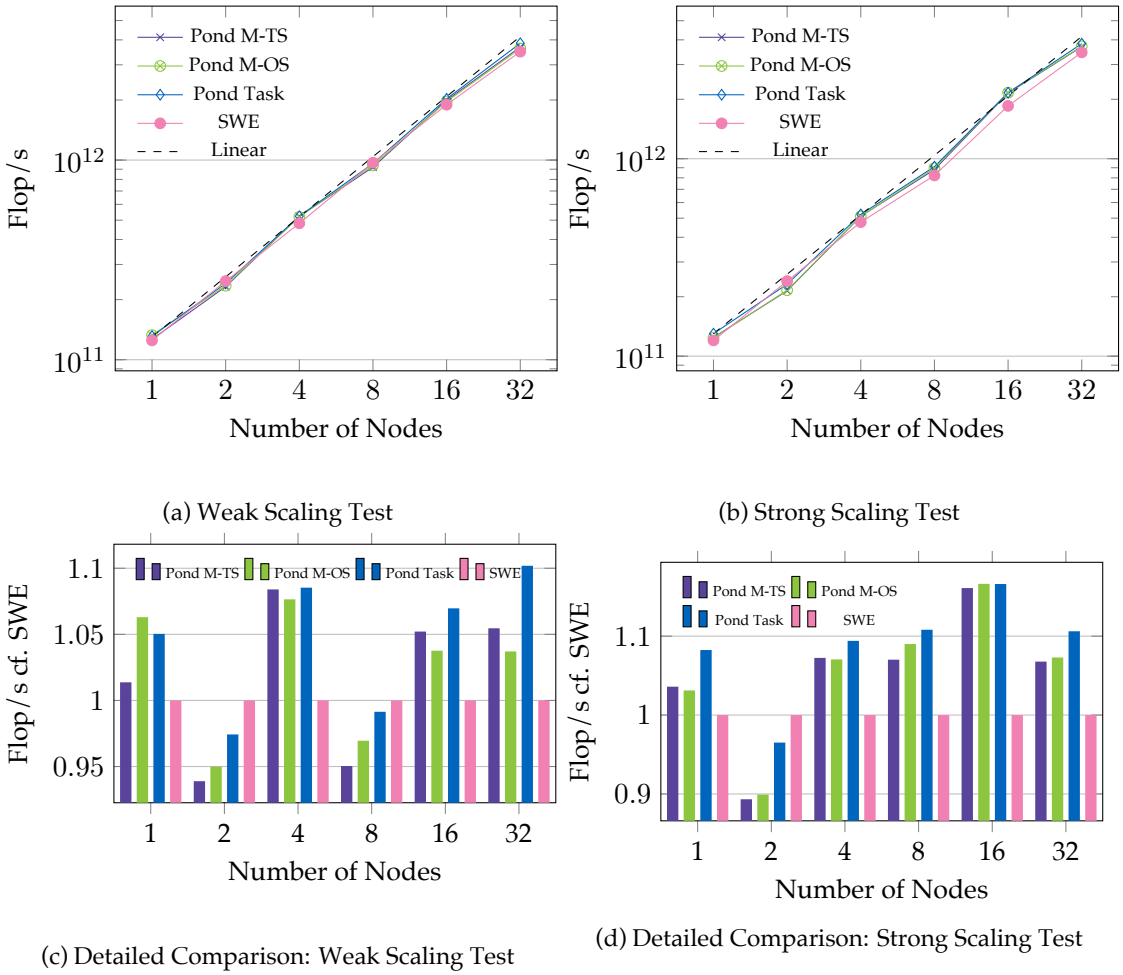
The results of the tests are depicted in Figure 18.8. In the weak scaling test, all application configurations scale linearly with the number of nodes involved in the computation. The same holds for the strong scaling test. Both tests suggest similar results: Performance across both solutions is relatively similar. Depending on the number of nodes, SWE or one of the configurations of Pond is slightly faster, but there is no clear advantage to either solution in terms of performance. However, for most configurations, *Pond Task* tends to slightly outperform *Pond M-TS* and *Pond T-OS*. Furthermore, the two-sided *Pond M-TS* seems to perform slightly better than *Pond T-OS*. However, as the differences in performance between the solutions are rather small, these conclusions should be taken with a grain of salt. The most important takeaway from the tests is that

[2] `https://doku.lrz.de/display/PUBLIC/CoolMUC-3`

Table 18.3.: Configurations used for the scaling tests with Actor-MPI on CoolMUC 2

Execution Type	Symbol	Description
Pond M-TS	×	Pond using Actor-MPI and two-sided communication operations. Four ranks per node, and about sixteen actors per rank.
Pond M-OS	⊗	Pond using Actor-MPI and one-sided communication operations. Four ranks per node, and about sixteen actors per rank.
Pond Task	◇	Pond using Actor-UPC++ and the task-based execution strategy. Four ranks per node, and about sixteen actors per rank.
SWE	●	SWE using MPI and OpenMP. One rank per node, and 28 OpenMP threads per rank.
Linear	- - -	Ideal scaling based on fastest single node configuration of Pond Task

Actor-MPI provides a viable alternative solution to Actor-UPC++, especially on clusters where GASNet-EX does not provide a native support for the interconnection network.



(a) Weak Scaling Test

(b) Strong Scaling Test

(c) Detailed Comparison: Weak Scaling Test

(d) Detailed Comparison: Strong Scaling Test

Figure 18.8.: Scaling tests performed on CoolMUC 2 comparing Pond using Actor-MPI with SWE and Pond using Actor-UPC++. Configurations are described in Table 18.3. In Figure 18.8c and Figure 18.8d, the relative performance compared to SWE with the same number of nodes is depicted. Experiments conducted by Macedo Miguel (2019).

19. Discussion and Outlook

In this part of the thesis, I presented the application domain of tsunami simulation as a target for actor-based application design. In the first chapter, I introduced the theoretical model. Afterwards, I described the SWE application. It serves as the basis for my two actor-based tsunami proxy applications, SWE-X10 and Pond. SWE is used to gain an overview of the currently available alternatives to my actor-based approach. The application was extended to work with Charm++, HPX and plain UPC++ (without the actor library). In a performance test, none of the available runtime systems were able to gain a clear advantage over the MPI baseline. The first actor-based proxy application introduced here is SWE-X10, based on ActorX10. It is a rewrite of the SWE application in X10 that targets HPC architectures as well as the InvasIC technology stack. I use the application to demonstrate advantages of actor-based application modelling by implementing lazy activation. Lazy activation uses inherent properties of hyperbolic PDEs to keep actors dormant until they are actually needed in the computation. Depending on the scenario, this optimization may lead to a significant reduction in CPU resource utilization. SWE-X10 is able to run on general-purpose CPUs as well as GPUs and the *i*-Core. The second tsunami proxy application, Pond, was implemented to demonstrate the benefits of using the actor model on a larger scale. It uses more standard components: modern C++ and the UPC++ APGAS library that offers communication backends for commonly used network technologies such as InfiniBand. In a comparison with SWE, Pond using the task-based execution strategy of Actor-UPC++ was, consistently, at least as fast as SWE, and for some runs up to $1.5\times$ faster.

As discussed before in chapter 13, the higher overhead introduced with ActorX10 becomes problematic for larger-scale parallelism. For applications with a very high number of actors per node to work, it is important that only actors that actually need to compute utilize compute resources. For SWE-X10 and Pond with the thread-based execution strategy, this is not the case, as they are bound to threads. As the number of actors per UPC++ rank or X10 Place grows, they perform successively worse. On the other hand, tasks seem to be a good analogy for the actions performed by the actors' FSMs. The best performance in the scaling test was reached for the task-based execution strategy.

In future work, it may be worth exploring how to formalize the actor's FSM directly in the library. Having knowledge about the firing behavior of an actor may be beneficial in avoiding redundant invocations of the `act()` method, and thereby reduce the performance bottleneck that is the master thread of the UPC++ rank. For Pond, it would be interesting to add support to GPUs, and use them alongside the CPU version on suitable hardware. Here, it would be worth exploring how to map the actions of the actors efficiently to CUDA kernels, and to use the same GPUs efficiently by multiple actors. Initial work on this topic has been done already by Molzer (2017).

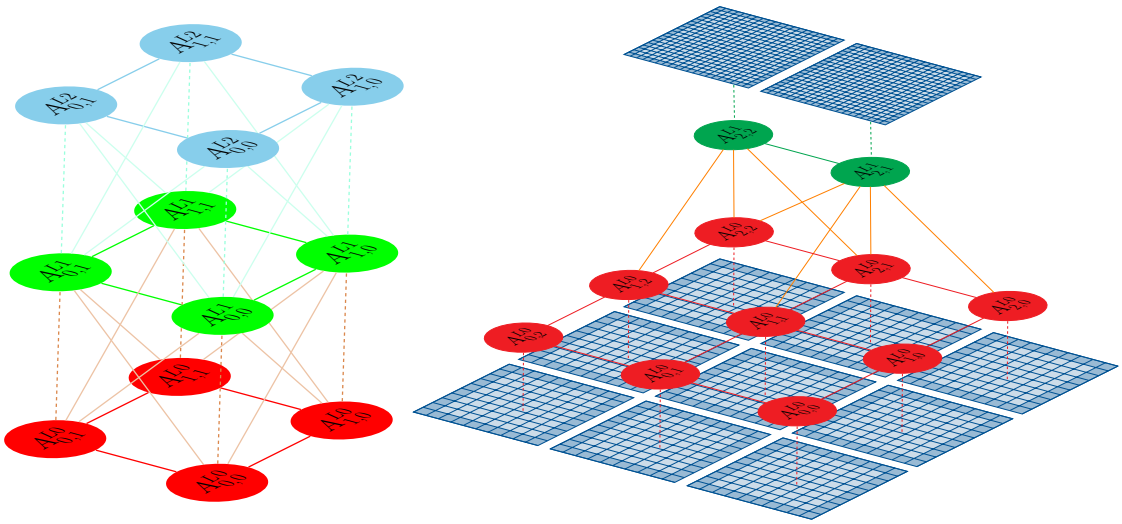


Figure 19.1.: Actor Graphs with multiple refinement levels. The actor graph on the left has a full grid of actors on three refinement levels (patches are not shown). The actor graph on the right only has refined actors for a limited number of patches that may be of particular interest. Here, the patch is depicted for each actor.

In SWE-X10, the next step would be integration of multi-level actor graphs. Currently, there is support for multi-rate local time stepping and actors of different resolutions on multiple levels. However, the support for a dynamic switching between different resolutions is not finished yet, and the solution has not been properly tested. Once it works, it allows for a static multi-level actor graph with uninitialized patches of multiple resolution levels. Each active actor knows, based on an initial coordination between actors at the beginning of the simulation, which of his neighbors in a given direction to talk to: its neighbor on the lower level, the one on the same level, or the one on the upper level. When an actor wants to hand over to another resolution level, it creates a refined or coarsened version of its patch, and sends it to his partner occupying the same spacial index, but on a neighboring level. In Figure 19.1 on the left side, for example, $A^{L1}_{0,0}$ may send a refined version of the patch to $A^{L2}_{0,0}$. Actors may switch only at full time step multiples. If they want to, they send a signal that they will perform a switch alongside the ghost layer data. The neighbors then know that any update past the time stamp of the ghost layer information that indicated the switch have to be sent to the neighbor on the new level. Actors support communication with three levels: the same level, the once-coarsened level, and the once-refined level.

Part IV.

Conclusion

20. Conclusion

In the beginning of the thesis, I posed the following research questions:

1. *Does the actor model ease the development of HPC applications compared to traditional models?*
2. *Does using the actor model in HPC applications yield performance competitive with the traditional approach?*

Before answering them, I will summarize the findings of the thesis.

ActorX10 and Actor-UPC++ I implemented two actor libraries: one targeting the X10 programming language and the invasive compute stack, the other targeting the classic HPC environment in the form of modern C++. Both libraries utilize PGAS principles to realize the FunState actor model on distributed systems. Application developers using the libraries specify their application through actors and their relationships to each other. Actors contain a finite state machine that is triggered whenever there are changes in one of the actor's ports. This allows actors to perform actions based only on their local data, received tokens, and capacity of their ports. There is no need for synchronization primitives such as mutual exclusion, barriers or semaphores. Actors are connected into a graph using channels holding tokens of fixed type. The graph may then be distributed across the nodes participating in the computation.

Tsunami Simulation Using ActorX10 I evaluated the use of the actor libraries using the application domain of tsunami simulation. The first application is the ActorX10-based SWE-X10. Thanks to X10, it is able to run efficiently on the invasive technology stack as well as on HPC systems. SWE-X10 demonstrates that programming applications using the actor model is not only feasible, but also eases development. The actor model adds another level of abstraction between the hardware and the application domain. This allows for more flexible coordination schemes taking advantage of optimization opportunities inherent to the problem domain. In SWE-X10, this means using the finite propagation speed of the tsunami wave to allow actors to only start working when the wave actually gets close to them. Depending on the scenario, this technique can lead to a significant reduction in used compute time. In a test on a distributed system with eight nodes and 256 actors and an initial perturbation in the corner of the simulation domain, I found a reduction in CPU time by over 40% compared to a baseline solution without lazy activation. SWE-X10 also supports the use of GPUs by its actors. Using CUDA streams, multiple actors may use the GPUs compute resources. In a scaling test, the GPU version of SWE-X10 performed comparably to a prior GPU version of the SWE code. Finally, SWE-X10 was extended to support the *i*-Core, a general purpose CPU with a closely coupled reconfigurable fabric. To fully utilize the capabilities of the core, a triple buffering scheme was implemented that preloads the next row in the patch

iteration, while computing the current one, using an asynchronous DMA engine. Compared to the default solution without the reconfigurable fabric, the patch iteration may be sped up by a factor of 4.82.

Tsunami Simulation using Actor-UPC++ The second actor library is built using the experiences gained from the X10 actor library. The most important take-away is that compute resources need to be uncoupled from actors. In ActorX10, actors are too closely linked to activities. Each actor is running on its own activity, and if there are multiple actors waiting for each other, the operating system scheduler may need to schedule between them. When the rank-based or the task-based execution strategies of Actor-UPC++ are used, actors only use compute resources if they actually need them, as the actor graph tracks channels connected to actors to notify them only if something changed, and the actors' FSMs need to be invoked. The library was evaluated using the Pond shallow water proxy application. Compared to SWE-X10, Pond is able to accommodate smaller patches while retaining good performance, and able to utilize parallelism more effectively without requiring huge computational domains. This is especially useful on MIC architectures such as the Intel Xeon Phi. Compared to conventional CPUs, these novel architectures feature a larger number of cores with improved floating-point performance and weaker general purpose performance. This makes a low-overhead actor execution even more important than on general purpose CPUs. In tests using the Xeon Phis on the Cori cluster, Pond managed to outperform a classical BSP-based shallow water application consistently, by up to a factor of $1.6\times$, and SWE-X10 by an order of magnitude.

Conclusions The first of my research questions may be answered with a yes. Compared to the traditional approach, use of the actor model reduces the number of frameworks, and the dependencies between different parallel components. Instead of looking at shared-memory and distributed-memory parallelism separately, the actor model allows for a cohesive view of the application, in terms of actors as units of computation. Instead of requiring a direct communication, actors use message queues to communicate. This decouples actors from each other. There is no need to synchronize before sending a message to make sure that the receiver is ready and data that is still needed is never overwritten. Instead, actors simply deposit the message in a queue and the receiving actor is notified that a new message is available. Actors also do not need to know details about their neighbors, only their communication interface. It is not relevant how (or where) the message is created. This makes a heterogeneous actor graph that uses accelerators alongside conventional hardware very straightforward to implement. One simply has to implement both, and connect them within the same graph.

The second research question may also be answered in the affirmative. I evaluated the performance behavior of both the X10 and the UPC++ actor libraries both on conventional multi-core and on many-core processors. In both cases, the Performance of the actor libraries was competitive with the performance of the application based on MPI and OpenMP. Furthermore, the higher abstraction level introduced with the actor formalism enables an easy implementation of high-level optimizations such as lazy activation, which enabled further performance benefits. While it is also possible to implement these using the classical frameworks, the burden on the application

developer is significantly larger, as all the low-level details, such as buffer management and synchronization, need to be explicitly handled.

Appendix

A. Code Samples

A.1. Cannon's Algorithm in ActorX10

In the following, the full code of the actor-based implementation of Cannon's Algorithm using ActorX10 is shown. The code contains an outer class `Cannon` that acts as a driver of the computation. The actor graph is implemented in the class `CannonActorGraph`. In the method built, the actors are created, their ports connected, and finally distributed onto the available X10 Places. The class `CannonActor` implements the actor itself. Besides the initialization code, it also contains the `act()` method that contains the actor's FSM. Its structure follows the theoretical one outlined in chapter 9. The test driver executes the algorithm with a total matrix size of 16×16, and a total of 4×4 actors.

```
1  import x10.util.*;
2  import actorlib.*;
3
4  public class Cannon {
5      static class Matrix(rows:Int, columns:Int) {
6          val storage : Array[Float]{rect, rank==2};
7
8          public def this(rows:Int, columns:Int, initialization:(Point(2))=>Float)
           ↪ {
9              property(rows, columns);
10             val matRegion = Region.makeRectangular([0,0],[rows - 1, columns -
               ↪ 1]);
11             this.storage = new Array[Float](matRegion, initialization);
12         }
13
14         public operator this + (other:Matrix{(self.rows == this.rows) &&
           ↪ (self.columns == this.columns)})
15             : Matrix{self.rows == this.rows && self.columns == this.columns} {
16             return new Matrix (rows, columns, ((p:Point(2)) => this.storage(p) +
               ↪ other.storage(p)));
17         }
18
19         public operator this * (other:Matrix{this.columns == self.rows})
20             : Matrix {(self.rows == this.rows) && (self.columns ==
               ↪ other.columns)} {
21             return new Matrix (this.rows, other.columns, ((p:Point) => {
```

```
22                  val row = p(0);
23                  val column = p(1);
24                  val target = this.columns;
25                  var res : Float = 0.0f;
26                  for ([i] in 0 .. (target - 1)) {
27                      res += this.storage(row, i) * other.storage(i, column);
28                  }
29                  return res;
30              }));
31          }
32
33          public def setMatrix(other:Matrix{(self.rows == this.rows) &&
            ↪  (self.columns == this.columns)}) {
34              for ([y,x] in storage) {
35                  this.storage(y,x) = other.storage(y,x);
36              };
37          }
38
39          public def toString() {
40              var res : String = "";
41              for (var y : Int = 0; y < rows; y++) {
42                  res += "| ";
43                  for (var x : Int = 0; x < rows; x++) {
44                      res += storage(y,x) + " ";
45                  }
46                  res += "|";
47              }
48              return res;
49          }
50      }
51
52
53      static class CannonActorGraph extends ActorGraph {
54          val n : Int;
55          val p : Int;
56          val actorRegion : Region{rect, zeroBased, rank==2};
57
58          def this(n:Int, p:Int) {
59              super("CannonActorGraph");
60              this.n = n;
61              this.p = p;
62              this.actorRegion = Region.makeRectangular([0,0], [p-1, p-1]);
63              if (n%p != 0) {
64                  throw new IllegalArgumentException("Matrix size must be an even
                    ↪  multiple of the number of actors!");
65              }
66          }
```

```
67
68          def build() {
69              val actors = new Array[CannonActor](actorRegion, (pt:Point(2)) => {
70                  val a = new CannonActor(pt(0), pt(1), n/p, p);
71                  addActor(a);
72                  a.initPorts();
73                  return a;
74              });
75
76              for ([i,j] in actorRegion) {
77                  val a = actors(i,j);
78                  val left = actors((i + p - 1) % p, j);
79                  val upper = actors(i, (j + p - 1) % p);
80                  connectPorts(a.left, left.right, 2);
81                  connectPorts(a.above, upper.below, 2);
82              }
83
84              val distPlan = Dist.makeBlockBlock(actorRegion, 0,1);
85              val distribution = new HashMap[String, Place]();
86              for (p in actorRegion) {
87                  val plc = distPlan(p);
88                  distribution.put(actors(p).name, plc);
89              }
90              distributeActors(distribution);
91
92              finish for ([i,j] in actorRegion) async {
93                  val aRef = getActor(actors(i,j).name);
94                  aRef.evalAtHome((a:Actor) => {
95                      (a as CannonActor).placeInitialTokens();
96                      return 0;
97                  });
98              }
99          }
100     }
101
102     static val STATE_COMPUTE = 1;
103     static val STATE_FINISHED = 2;
104
105     static class CannonActor extends Actor {
106         val above : OutPort[Matrix];
107         val left : OutPort[Matrix];
108         val right : InPort[Matrix];
109         val below : InPort[Matrix];
110
111         private var state:Int = STATE_COMPUTE;
112
113         val numOperations : Int;
```

```
114         val size : Int;
115         val i : Int;
116         val j : Int;
117         var c : Matrix;
118         var operationsPerformed : Int;
119
120
121         def this(i:Int, j:Int, size:Int, numOperations:Int) {
122             super("Cannon_"+i+"_"+j);
123             this.i = i;
124             this.j = j;
125             this.size = size;
126             this.numOperations = numOperations;
127             this.above = new OutPort[Matrix]("A");
128             this.left = new OutPort[Matrix]("L");
129             this.right = new InPort[Matrix]("R");
130             this.below = new InPort[Matrix]("B");
131             c = new Matrix(size, size, (Point(2)) => 0.0f);
132             operationsPerformed = 0;
133         }
134
135         def initPorts() {
136             addOutPort(above);
137             addOutPort(left);
138             addInPort(right);
139             addInPort(below);
140         }
141
142         def placeInitialTokens() {
143             val matrixA = new Matrix(size, size, (Point) => 1.0f);
144             val matrixB = new Matrix(size, size, (Point) => 1.0f);
145             above.write(matrixA);
146             left.write(matrixB);
147         }
148
149         protected def act() {
150             if (state == STATE_COMPUTE && operationsPerformed < numOperations &&
                ↪  left() && above() && below() && right()) {
151                 val a = below.read();
152                 val b = right.read();
153                 this.c = this.c + a * b;
154                 operationsPerformed++;
155                 Console.OUT.println(name + "\tperformed operation " +
                    ↪  operationsPerformed + "/" + numOperations +")");
156                 left.write(b);
157                 above.write(a);
```

```
158             } else if (state == STATE_COMPUTE && operationsPerformed ==
                ↪  numOperations) {
159                 state = STATE_FINISHED;
160                 atomic {
161                     val res = c.toString();
162                     Console.OUT.println(res);
163                 }
164                 stop();
165             } else if (state == STATE_FINISHED) {
166             }
167         }
168     }
169
170     public static def main(args:Array[String](1)) {
171         val graph = new CannonActorGraph(16, 4);
172         graph.build();
173         Console.OUT.println(graph.prettyPrint());
174         finish graph.start();
175     }
176 }
```

A.2. Cannon's Algorithm in Actor-UPC++

In the following, the full code of the actor-based implementation of Cannon's Algorithm using Actor-UPC++ is shown. The code is structured as follows: There is a class `CannonActor` for the actor:

```
1  #ifndef CANNON_ACTOR_HPP
2  #define CANNON_ACTOR_HPP
3
4  #include "actorlib/InPort.hpp"
5  #include "actorlib/OutPort.hpp"
6  #include "actorlib/Actor.hpp"
7
8  #include "Matrix.hpp"
9
10 #pragma once
11
12 enum class CannonActorState {
13     COMPUTE, FINISHED
14 };
15
16 class CannonActor : public Actor {
17     private:
```

```
18          InPort<Matrix, 4> *right;
19          InPort<Matrix, 4> *down;
20          OutPort<Matrix, 4> *left;
21          OutPort<Matrix, 4> *up;
22
23          size_t i;
24          size_t j;
25          size_t size;
26          size_t numOperations;
27          size_t operationsPerformed;
28          Matrix result;
29          CannonActorState currentState;
30      public:
31          CannonActor(size_t i, size_t j, size_t size, size_t numOperations);
32          void act();
33          void placeInitialTokens();
34      private:
35          void performPartialComputation();
36          void performShutdown();
37          bool mayRead();
38          bool mayWrite();
39  };
40
41  #endif
```

One difference is that in Actor-UPC++, the capacity of the ports is visible directly in the actors' signatures, otherwise, the implementation is very similar to the one in ActorX10. In C++, one typically separates the implementation of a class from its interface. Therefore, the implementation of the methods of the class is given in a different compilation unit:

```
1   #include "CannonActor.hpp"
2
3   #include <cstddef>
4   #include <cstdlib>
5   #include <string>
6
7   using namespace std::literals;
8
9   CannonActor::CannonActor(size_t i, size_t j, size_t size, size_t numOperations)
10    : Actor("Cannon_"s + std::to_string(i) + "_"s + std::to_string(j)),
11      right(nullptr),
12      down(nullptr),
13      left(nullptr),
14      up(nullptr),
15      i(i),
16      j(j),
17      size(size),
```

```
18      numOperations(numOperations),
19      operationsPerformed(0),
20      result(size,size,[](size_t, size_t) {return 0.0f;}),
21      currentState(CannonActorState::COMPUTE) {
22      right = this->makeInPort<Matrix, 4>("R");
23      down = this->makeInPort<Matrix, 4>("D");
24      left = this->makeOutPort<Matrix, 4>("L");
25      up = this->makeOutPort<Matrix,4>("U");
26  }
27
28  void CannonActor::placeInitialTokens() {
29      auto init = [](size_t, size_t) {return 1.0f;};
30      Matrix a(size, size, init);
31      Matrix b(size, size, init);
32      up->write(a);
33      left->write(b);
34  }
35
36  void CannonActor::act() {
37      switch (currentState) {
38          case CannonActorState::COMPUTE:
39              if (operationsPerformed < numOperations && mayRead() && mayWrite()) {
40                  performPartialComputation();
41              } else if (operationsPerformed == numOperations) {
42                  performShutdown();
43              }
44              break;
45          case CannonActorState::FINISHED:
46              break;
47          default:
48              abort();
49              break;
50      }
51  }
52
53  void CannonActor::performPartialComputation() {
54      auto a = down->read();
55      auto b = right->read();
56      auto tmp = a * b;
57      this->result = this->result + tmp;
58      operationsPerformed++;
59      left->write(b);
60      up->write(a);
61      std::cout << name << "\tperformed operation " << operationsPerformed << "/"
        ↪  << numOperations << std::endl;
62  }
63
```

```
64 void CannonActor::performShutdown() {
65     auto res = result.to_string();
66     std::cout << res << std::endl;
67     stop();
68 }
69
70 bool CannonActor::mayRead() {
71     return this->down->available() > 0 && this->right->available() > 0;
72 }
73
74 bool CannonActor::mayWrite() {
75     return this->left->freeCapacity() > 0 && this->up->freeCapacity() > 0;
76 }
```

The main difference to the ActorX10 version is that the complete initialization of the actor is performed in the constructor. It is no longer necessary to call another method to initiate the ports.

The tokens passed around are objects of the `Matrix` class. The functionality implemented in the class is the same as in the X10 version, with the addition of manual serialization. This is visible in the signature of the class:

```
1  #ifndef MATRIX_HPP
2  #define MATRIX_HPP
3
4  #include <upcxx/upcxx.hpp>
5
6  #include <cstddef>
7  #include <functional>
8  #include <vector>
9
10 #pragma once
11
12 struct Matrix {
13     size_t rows;
14     size_t cols;
15     std::vector<float> data;
16
17     UPCXX_SERIALIZED_FIELDS(rows, cols, data)
18
19     Matrix();
20     Matrix(size_t rows, size_t cols, std::function<float(size_t, size_t)> init);
21     Matrix operator* (Matrix &other);
22     Matrix operator+ (Matrix &other);
23
24     std::string to_string();
```

```
};
#endif
```

Alternatively, one may also provide a custom serialization. This may be done using an explicit specialization of the class template `upcxx::serialization<T>`. Its implementation is shown below.

```
#ifndef MATRIX_SERIALIZATION_HPP
#define MATRIX_SERIALIZATION_HPP

#include <upcxx/serialization.hpp>
#include <upcxx/upcxx.hpp>

#include <cstddef>
#include <vector>

#include "Matrix.hpp"

#pragma once

namespace upcxx {
    template <>
    struct serialization<Matrix> {
        template<typename Reader>
        static Matrix* deserialize(Reader &r, void *storage) {
            size_t cols = r.template read<size_t>();
            size_t rows = r.template read<size_t>();
            std::vector<float> data(rows*cols);
            r.template read_sequence_into<float>((void *)data.data(), rows*cols);
            Matrix *m = new (storage) Matrix();
            m->data = data;
            m->rows = rows;
            m->cols = cols;
            return m;
        }

        template<typename Writer>
        static void serialize(Writer &w, Matrix const &m) {
            w.write(m.rows);
            w.write(m.cols);
            w.write_sequence(m.data.begin(), m.data.end());
        }
    };
}
#endif
```

The actor graph is created using composition rather than inheritance in Actor-UPC++. Therefore, the Actor-UPC++ graph instance is an attribute of the `CannonActorGraph` class. Furthermore, the interface is simplified: one only needs to start the computation once the graph has been created. Initialization of the graph is concluded once the constructor returns.

```
#ifndef CANNON_ACTOR_GRAPH_HPP
#define CANNON_ACTOR_GRAPH_HPP

#include <cstddef>
#include <vector>

#include <upcxx/upcxx.hpp>

#include "actorlib/ActorGraph.hpp"

#include "CannonActor.hpp"

#pragma once

class CannonActorGraph {
    size_t const n;
    size_t const p;

    ActorGraph graph;
    std::vector<CannonActor *> localActors;

    public:
    CannonActorGraph(size_t n, size_t p);
    ~CannonActorGraph();
    void initialize();
    void performComputation();

    private:
    void forallLocalActors(std::function<void(size_t, size_t)> action);
    upcxx::intrank_t getActorRank(size_t xPos, size_t yPos);
};

#endif
```

The implementation of the class's methods is similar to the one in the ActorX10 version. The computation is set up in three passes: First, the actors are created. Second, their ports are connected following the scheme laid out in Figure 9.1. Finally, the initial tokens are added. The main difference to the ActorX10 version is that the instances on all ranks perform the work concurrently in an SPMD fashion. Furthermore, the CannonActorGraph class is also responsible for setup and tear-down of the UPC++ runtime.

```cpp
#include <upcxx/backend_fwd.hpp>
#include <upcxx/barrier.hpp>
#include <upcxx/upcxx.hpp>

#include <cstddef>
#include <cmath>
#include <iostream>
#include <string>

#include "CannonActor.hpp"
#include "CannonActorGraph.hpp"
#include "actorlib/Actor.hpp"

CannonActorGraph::CannonActorGraph(size_t n, size_t p)
  : n(n),
    p(p) {
    using namespace std::literals;
    upcxx::init();
    forallLocalActors([&](size_t x, size_t y) {
        CannonActor *ca = new CannonActor(x,y,n/p, p);
        localActors.push_back(ca);
        graph.addActor(ca);
    });

    upcxx::barrier();

    forallLocalActors([&](size_t x, size_t y) {
        auto yTop = (y + p - 1) % p;
        auto xLeft = ( x + p - 1) % p;
        GlobalActorRef a = graph.getActor("Cannon_"s + std::to_string(xLeft) +
        ↪  "_"s + std::to_string(y));
        GlobalActorRef leftActor = graph.getActor("Cannon_"s
        ↪  +std::to_string(xLeft) + "_"s + std::to_string(y));
        GlobalActorRef topActor = graph.getActor("Cannon_"s + std::to_string(x) +
        ↪  "_"s + std::to_string(yTop));
        graph.connectPorts(a, "L", leftActor, "R");
        graph.connectPorts(a, "U", topActor, "D");
    });

    upcxx::barrier();

    for (auto ca : localActors) {
        ca->placeInitialTokens();
    }
}

```

```
44 void CannonActorGraph::forallLocalActors(std::function<void(size_t, size_t)>
   ↪  action) {
45     for (size_t y = 0; y < p; y++) {
46         for (size_t x = 0; x < p; x++) {
47             if (getActorRank(x,y) == upcxx::rank_me()) {
48                 action(x,y);
49             }
50         }
51     }
52 
53 }
54 
55 CannonActorGraph::~CannonActorGraph() {
56     upcxx::finalize();
57 }
58 
59 void CannonActorGraph::performComputation() {
60     auto duration = graph.run();
61     std::cout << "Computation took " << duration << "s to finish." << std::endl;
62 }
63 
64 upcxx::intrank_t CannonActorGraph::getActorRank(size_t xPos, size_t yPos) {
65     size_t xSize = p;
66     size_t ySize = p;
67     auto tmp1 = static_cast<double>(xSize) /
       ↪  std::sqrt(static_cast<double>(upcxx::rank_n()));
68     double xBlockSize = std::floor(tmp1);
69     size_t xSplits = std::max(xSize / static_cast<size_t>(xBlockSize), 1ul);
70     size_t ySplits = upcxx::rank_n() / xSplits;
71     auto tmpX = xPos / (xSize / xSplits);
72     auto tmpY = yPos / (ySize / ySplits);
73     auto res = static_cast<upcxx::intrank_t>(tmpX * ySplits + tmpY);
74     return std::min(res, upcxx::rank_n() - 1);
75 }
```

Finally, the execution of the code is started using a test driver.

```
1 #include "CannonActorGraph.hpp"
2 
3 #include <iostream>
4 
5 int main(int argc, const char **argv) {
6     CannonActorGraph graph(16,4);
7     graph.performComputation();
8 }
```

B. Scaling Tests of SWE on CoolMUC2

This appendix is an edited version of the one that appeared previously in Bogusz et al. (2020).

B.1. Summary of the Experimental Setup

We ran strong scaling tests with the global and local time stepping scheme. Tests were executed on the CoolMUC2 cluster with node configurations of 1, 2, 4, 8, 16 and 32 with the MPI, UPC++, Charm++, HPX and Chameleon SWE applications. Different execution strategies were examined: MPI and UPC++ were configured with 1 block per rank and 1 rank per core. Charm++, HPX and Chameleon used 64 and 128 blocks per computational node.

Application	Compile Command
MPI	`CC=mpicc CXX=mpicxx cmake -DCMAKE_BUILD_TYPE=Release .. & make`
UPC++	`CC=mpicc CXX=mpicxx cmake -DCMAKE_BUILD_TYPE=Release .. & make`
Charm++	`scons writeNetCDF=True compiler=intel openmp=false solver=hybrid parallelization=charm asagi=false copyenv=true vectorize=true`
HPX	`CC=mpicc CXX=mpicxx cmake -DCMAKE_BUILD_TYPE=Release .. & make`
Chameleon	`CC=mpicc CXX=mpicxx cmake -DCMAKE_BUILD_TYPE=Release .. & make`

Note that compilation requires setting the installation path of the respective libraries as environment variables, e.g., for Charm++, `CHARM_PATH` must be set to a valid installation of Charm++.

B.2. List of Artifacts

- **SWE-Benchmark:** `https://gitlab.lrz.de/poeppl/swe-benchmark`
- **Chameleon:** `https://github.com/chameleon-hpc/chameleon`

B.3. Environment of the Experiment

CoolMUC2 consists of 812 compute nodes with a combined peak performance of 1.2 PFlop/s. Each node is equipped with two Intel Xeon E5-2690v3 "Haswell" CPUs with 14 cores each as well as 64 GB of main memory. The nodes are connected using the Mellanox InfiniBand FDR14 interconnection fabric. On the compute nodes, *SUSE Linux Enterprise Server 15 SP1* running

Linux kernel 4.12.14-197.40-default is used. For the compilation of SWE and the frameworks we evaluated, we used as compiler:

- *Intel C++ Compiler:* icpc version 19.0.5.281 (gcc version 8.2.0 compatibility)

For the parallelization frameworks, we used:

- IntelMPI 2019.7.217
- Chameleon 0.1
- UPC++ 2020.3.0
- Charm++ 6.10.1
- HPX 1.4.1

Paper Modifications For all tests we use the SWE-Benchmark repository, which is an extended version of SWE for benchmarking and performance. The version used for this paper is tagged with *SC_PAW-ATM_Workshop_submission*. For the Chameleon library we created an external hard-coded CPU pinning patch for Cluster execution environment, as we had trouble with the correct pinning of the communication thread to the last CPU. The patch is located in the aforementioned branch in the folder *patches*.

Output from scripts that gathers execution environment information The configuration was used for all jobs on CoolMUC2. The configuration below may only be considered a snapshot. As the cluster is updated, the versions listed below may no longer be available.

1. admin/1.0
2. tempdir/1.0
3. lrz/1.0
4. spack/staging/20.1.1
5. intel/19.0.5
6. intel-mkl/2019.5.281
7. intel-mpi/2019.7.217
8. netcdf-hdf5-all/4.7_hdf5-1.10-intel19-impi
9. python/2.7_intel
10. gcc/8
11. hwloc/1.11
12. cmake/3.15.4
13. scons/3.1.1
14. slurm_setup/1.0

Table B.1.: CoolMUC 2 Compute Node CPU Information

CPU information	
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Address sizes:	46 bits physical, 48 bits virtual
CPU(s):	56
On-line CPU(s) list:	0-55
Thread(s) per core:	2
Core(s) per socket:	14
Socket(s):	2
NUMA node(s):	4
Vendor ID:	GenuineIntel
CPU family:	6
Model:	63
Model name:	Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
Stepping:	2
CPU MHz:	2599.973
CPU max MHz:	2600.0000
CPU min MHz:	1200.0000
BogoMIPS:	5199.94
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	17920K
NUMA node0 CPU(s):	0-6,28-34
NUMA node1 CPU(s):	7-13,35-41
NUMA node2 CPU(s):	14-20,42-48
NUMA node3 CPU(s):	21-27,49-55
Flags:	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm cpuid_fault epb invpcid_single pti intel_ppin ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm xsaveopt cqm_llc cqm_occup_llc dtherm ida arat pln pts md_clear flush_l1d

C. Scaling Tests of Pond and SWE on Cori

This appendix is an edited version of the one that appeared previously in Pöppl, Bader, and Baden (2019).

C.1. Summary of the Experimental Setup

I ran strong scaling and weak scaling tests with node configurations of 1, 2, 4, 8, 16, 32, 64 and 128 nodes with the applications SWE, SWE-X10 and Pond on Cori. For Pond, I tested the three different execution strategies (rank-based execution strategy, thread-based execution strategy and task-based execution strategy). SWE was built using the classical MPI+OpenMP stack. SWE-X10 used the portable MPI backend, as the only native interface supported is the PAMI interconnect of the BlueGene clusters. I built the applications using the following commands:

Application	Compile Command
SWE	`scons buildVariablesFile = build/options/SWE_cray_mpi_-vectorized.py`
SWE-X10	`make clean; make NATIVE=1 NERSC=1 LOG=info RT=mpi`
Pond Rank	`CXX="env UPCXX_THREADMODE=seq upcxx -O3" cmake -DBUILD_RELEASE=ON -DBUILD_USING_UPCXX_WRAPPER=ON -DENABLE_FILE_OUTPUT=OFF -DENABLE_-MEMORY_SANITATION=OFF -DENABLE_LOGGING=OFF -DENABLE_O3_UPCXX_-BACKEND=ON -DENABLE_PARALLEL_UPCXX_BACKEND=OFF -DACTORLIB_USE_-OPENMP_TASKS=OFF -DIS_CROSS_COMPILING=ON --build pond <PATH>; make`
Pond Thread	`CXX="env UPCXX_THREADMODE=par upcxx -O3" cmake -DBUILD_RELEASE=ON -DBUILD_USING_UPCXX_WRAPPER=ON -DENABLE_FILE_OUTPUT=OFF -DENABLE_-MEMORY_SANITATION=OFF -DENABLE_LOGGING=OFF -DENABLE_O3_UPCXX_-BACKEND=ON -DENABLE_PARALLEL_UPCXX_BACKEND=ON -DACTORLIB_USE_-OPENMP_TASKS=OFF -DIS_CROSS_COMPILING=ON --build pond <PATH>; make`
Pond Task	`CXX="env UPCXX_THREADMODE=par upcxx -O3" cmake -DBUILD_RELEASE=ON -DBUILD_USING_UPCXX_WRAPPER=ON -DENABLE_FILE_OUTPUT=OFF -DENABLE_-MEMORY_SANITATION=OFF -DENABLE_LOGGING=OFF -DENABLE_O3_UPCXX_-BACKEND=ON -DENABLE_PARALLEL_UPCXX_BACKEND=ON -DACTORLIB_USE_-OPENMP_TASKS=ON -DIS_CROSS_COMPILING=ON --build pond <PATH>; make`

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

C.2. List of Artifacts

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

C.3. Environment of the Experiment

The experiments were performed on NERSC's Cori *KNL Compute Nodes*. Each node contains a single-socket Intel® Xeon Phi™ Processor 7250 ("Knights Landing") processor with 68 cores per node @ 1.4 GHz. The nodes' operating system at the time of the experiment is SUSE Linux Enterprise Server 15. To compile the applications, I used

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

In Pond, the following libraries and frameworks were used.

- UPC++ 2019.3.0
- Metis 5.1.0

Input datasets and versions As the scenario for the tests, I used a synthetic radial dam break scenario that is generated programmatically at the start of the application.

Modifications for the Experiments For the default version of SWE, the OpenMP parallelization was not functional at the time the tests were performed, I re-enabled it in the build system and modified the code where necessary. Tests in SWE where performed using the most recent commit of the master branch at the time (*e80170a44*[1]). I also added the HLLE solver (not part of the main repository. It works as a drop-in replacement to the other Riemann solvers. Finally, I had to slightly modify the build system to make it work on Cori. A GIT patch containing the modifications, and the HLLE solver are available at: `https://bitbucket.org/apoeppl/actor-upcxx/downloads/` Changes necessary to run SWE-X10 on Cori are pushed to the SWE-X10 Git repository in the

[1] `https://github.com/TUM-I5/SWE/commit/e80170a445e8d1896a8b59bc6d5669ac7ce7d465`

branch *fix_cori-compilation*. The C++ code the X10 compiler generates seems to trigger a bug in the Intel C++ Compiler, version 19. Furthermore, the newest Java version capable of running the X10 Compiler is Java 7, therefore, the *JAVA_HOME* variable needs to be pointed to such an installation. The version of Pond and the actor library that has the functionality that was used for the test is marked in the actor-upcxx repository with the tag *pond-paper-submission-commit*. In some cases, the commits may not match the logs, this is due to changes in the job script generator that was used for the generation of the SLURM scripts for the tests.

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

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

Table C.1.: Cori Compute Node CPU Information

CPU information	
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	272
On-line CPU(s) list:	0-271
Thread(s) per core:	4
Core(s) per socket:	68
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	87
Model name:	Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
Stepping:	1
CPU MHz:	1401.000
CPU max MHz:	1401.0000
CPU min MHz:	1000.0000
BogoMIPS:	2799.98
L1d cache:	32K
L1i cache:	32K
L2 cache:	1024K
NUMA node0 CPU(s):	0-271
Flags:	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl est tm2 ssse3 fma cx16 xtpr pdcm sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch ring3mwait cpuid_fault epb pti intel_ppin ibrs ibpb fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms avx512f rdseed adx avx512pf avx512er avx512cd xsaveopt dtherm ida arat pln pts

Bibliography

Abdelfattah, Ahmed, Stanimire Tomov, and Jack Dongarra (Nov. 2019). "Towards Half-Precision Computation for Complex Matrices: A Case Study for Mixed Precision Solvers on GPUs". In: *2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pp. 17–24. doi: 10.1109/ScalA49573.2019.00008.

Agha, Gul (1985). *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Tech. rep. AITR-844. MIT Artificial Intelligence Laboratory.

Agha, Gul and Carl Hewitt (1988). "Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism". In: *Readings in Distributed Artificial Intelligence*. Ed. by Alan H. Bond and Les Gasser. Morgan Kaufmann, pp. 398–407. isbn: 978-0-934613-63-7. doi: https://doi.org/10.1016/B978-0-934613-63-7.50042-5. url: http://www.sciencedirect.com/science/article/pii/B9780934613637500425.

Altera (Oct. 2007). *Accelerating High-Performance Computing With FPGAs*. Tech. rep. Altera Corporation.

Anthony, Sebastian (Nov. 2013). *Intel unveils 72-core x86 Knights Landing CPU for exascale supercomputing*. url: https://www.extremetech.com/extreme/171678-intel-unveils-72-core-x86-knights-landing-cpu-for-exascale-supercomputing.

Armstrong, Joe (2007). "A History of Erlang". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, pp. 6–1–6–26. isbn: 9781595937667. doi: 10.1145/1238844.1238850. url: https://doi.org/10.1145/1238844.1238850.

Augonnet, Cédric, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier (2011). "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures". In: *Concurrency and Computation: Practice and Experience* 23.2, pp. 187–198. doi: 10.1002/cpe.1631. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1631. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1631.

Augustsson, Lennart (Sept. 1998). "Cayenne—a Language with Dependent Types". In: *SIGPLAN Not.* 34.1, pp. 239–250. issn: 0362-1340. doi: 10.1145/291251.289451. url: https://doi.org/10.1145/291251.289451.

Bachan, John (Mar. 2019). *UPC++ Specification, v1.0 Draft 10*. Tech. rep. LBNL-2001192. Lawrence Berkeley National Laboratory. doi: 10.25344/S4JS30. url: https://escholarship.org/uc/item/25m555p9.

Bachan, John, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed (2019). "UPC++: A High-Performance Communication Framework for Asynchronous Computation". In: *Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium*. IPDPS. Rio de Janeiro, Brazil: IEEE. doi: 10.25344/S4V88H. url: https://escholarship.org/uc/item/1gd059hj.

Bader, Michael (Oct. 2019). *High Performance Computing - Algorithms and Applications, Dense Linear Algebra*. Lecture Slides.

Bader, Michael, Alexander Breuer, Wolfgang Hölzl, and Sebastian Rettenberger (2014). "Vectorization of an augmented Riemann solver for the shallow water equations". In: *2014 International Conference on High Performance Computing Simulation (HPCS)*, pp. 193–201.

Bader, Michael, Alexander Pöppl, Oliver Meister, and Alexander Breuer (Jan. 2020). *Tutorial: HPC - Algorithms and Applications WS 19/20 - SWE Case Study*. Exercise Sheet. Available at: https://www.moodle.tum.de/course/view.php?id=49335, Retrieved at: April 29th 2020.

Baker, Henry and Carl Hewitt (Aug. 1977). "The Incremental Garbage Collection of Processes". In: *SIGPLAN Not.* 12.8, pp. 55–59. issn: 0362-1340. doi: 10.1145/872734.806932. url: https://doi-org.eaccess.ub.tum.de/10.1145/872734.806932.

Baker, Matthew, Swen Boehm, Aurelien Boutellier, and Barbara Chapman et al. (Dec. 2017). *OpenSHMEM Application Programming Interface*. Tech. rep. Open Source Software Solutions, Inc. (OSSS). url: http://www.openshmem.org.

Bale, Derek, Randall LeVeque, Sorin Mitran, and James A. Rossmanith (2003). "A Wave Propagation Method for Conservation Laws and Balance Laws with Spatially Varying Flux Functions". In: *SIAM Journal on Scientific Computing* 24.3, pp. 955–978. doi: 10.1137/S106482750139738X. eprint: http://dx.doi.org/10.1137/S106482750139738X. url: http://dx.doi.org/10.1137/S106482750139738X.

Barker, Brandon (2015). "Message Passing Interface (MPI)". In: *Workshop: High Performance Computing on Stampede*. Vol. 262. url: http://www.cac.cornell.edu/education/training/StampedeJan2015/IntroMPI.pdf.

Barney, Blaise (2010). "Introduction to Parallel Computing". In: *Lawrence Livermore National Laboratory* 6.13, p. 10.

Barthe, Gilles and Thierry Coquand (2002). "An Introduction to Dependent Type Theory". In: *Applied Semantics*. Ed. by Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–41. isbn: 978-3-540-45699-5.

Bauer, Lars (2009). "RISPP: A Run-time Adaptive Reconfigurable Embedded Processor". PhD thesis. doi: 10.5445/IR/1000021186.

Bauer, Lars, Artjom Grudnitsky, Marvin Damschen, Srinivas Rao Kerekare, and Jörg Henkel (Oct. 2015). "Floating Point Acceleration for Stream Processing Applications in Dynamically Reconfigurable Processors". In: *IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. Amsterdam, The Netherlands. doi: 10.1109/ESTIMedia.2015.7351762.

Bauer, Michael (2014). "Legion: Programming Distributed Heterogeneous Architectures with Logical Regions". PhD thesis. Ph. D. dissertation, Stanford University.

Bauer, Michael and Michael Garland (2019). "Legate NumPy: Accelerated and Distributed Array Computing". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery. isbn: 9781450362290. doi: 10.1145/3295500.3356175. url: https://doi.org/10.1145/3295500.3356175.

Bauer, Michael, Sean Treichler, Elliott Slaughter, and Alex Aiken (Nov. 2012). "Legion: Expressing locality and independence with logical regions". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11. doi: 10.1109/SC.2012.71.

Bogusz, Martin (Sept. 2019). "Exploring Modern Runtime Systems for the SWE Framework". Bachelorarbeit. Technical University of Munich.

Bogusz, Martin, Philipp Samfass, Alexander Pöppl, Jannis Klinkenberg, and Michael Bader (Nov. 2020). "Evaluation of Multiple HPC Parallelization Frameworks in a Shallow Water Proxy Application with Multi-Rate Local Time Stepping". In: *2020 IEEE/ACM 3rd Annual Parallel*

Applications Workshop: Alternatives To MPI+X (PAW-ATM). IEEE, pp. 27–39. doi: 10.1109/PAWATM51920.2020.00008.

Bonachea, Dan and Paul H. Hargrove (Oct. 2018). *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. doi: 10.25344/S4QP4W. url: https://escholarship.org/uc/item/0xg7b704.

Borrell, Ricard, Damien Dosimont, Marta Garcia-Gasulla, Guillaume Houzeaux, Oriol Lehmkuhl, Vishal Mehta, Herbert Owen, Marriano Vázquez, and Guillermo Oyarzun (2020). "Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture: Application to airplane aerodynamics". In: *Future Generation Computer Systems* 107, pp. 31–48. issn: 0167-739X. doi: https://doi.org/10.1016/j.future.2020.01.045. url: http://www.sciencedirect.com/science/article/pii/S0167739X1930994X.

Brand, Marcel, Frank Hannig, Alexandru Tanase, and Jürgen Teich (Sept. 2017). "Orthogonal Instruction Processing: An Alternative to Lightweight VLIW Processors". In: *2017 IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp. 5–12. doi: 10.1109/MCSoC.2017.17.

Breuer, Alexander and Michael Bader (2012). "Teaching Parallel Programming Models on a Shallow-Water Code". In: *2012 11th International Symposium on Parallel and Distributed Computing*, pp. 301–308.

Buchwald, Sebastian, Manuel Mohr, and Andreas Zwinkau (Mar. 2015). "Malleable Invasive Applications". In: *Proceedings of the 8th Working Conference on Programming Languages (ATPS'15)*.

Budanaz, Yakup (Aug. 2020). "Dynamic Actor Migration for a Distributed Actor Library". Bachelor's Thesis. Technical University of Munich.

Bungartz, Hans-Joachim, Christoph Riesinger, Martin Schreiber, Gregor Snelting, and Andreas Zwinkau (2013). "Invasive Computing in HPC with X10". In: *Proceedings of the Third ACM SIGPLAN X10 Workshop*. X10 '13. Seattle, Washington: Association for Computing Machinery, pp. 12–19. isbn: 9781450321570. doi: 10.1145/2481268.2481274. url: https://doi.org/10.1145/2481268.2481274.

Cannon, Lynn Elliot (1969). "A Cellular Computer to Implement the Kalman Filter Algorithm". AAI7010025. PhD thesis. USA: Montana State University.

Charousset, Dominik, Raphael Hiesgen, and Thomas C. Schmidt (Apr. 2016). "Revisiting Actor Programming in C++". In: *Computer Languages, Systems & Structures* 45, pp. 105–131. url: http://dx.doi.org/10.1016/j.cl.2016.01.002.

Charousset, Dominik, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch (Oct. 2013). "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, pp. 87–96.

Cheriton, David, Michael Malcolm, Lawrence Melen, and Gary Sager (Feb. 1979). "Thoth, a Portable Real-Time Operating System". In: *Communications of the ACM* 22.2, pp. 105–115. issn: 0001-0782. doi: 10.1145/359060.359074. url: https://doi.org/10.1145/359060.359074.

Chisnall, David, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann (Mar. 2015). "Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine". In: *SIGARCH Computer Architecture News* 43.1, pp. 117–130. issn: 0163-5964. doi: 10.1145/2786763.2694367. url: http://doi.acm.org/10.1145/2786763.2694367.

Christon, Mark, David Crawford, Eugene Hertel, James Peery, and Allen Robinson (June 1997). "ASCI Red – Experiences and lessons learned with a massively parallel teraFLOP supercomputer". In: -.

Cobham Gaisler AB (Jan. 2016). *GRLIB IP Library User's Manual*. Tech. rep. Version 1.5.0, retrieved on May 2nd, 2017. Göteborg, Sweden: Cobham Gaisler AB. url: http://www.gaisler.com/products/grlib/grlib.pdf.

Comprés, Isaias, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz (2016). "Infrastructure and API Extensions for Elastic Execution of MPI Applications". In: *Proceedings of the 23rd European MPI Users' Group Meeting*. EuroMPI 2016. Edinburgh, United Kingdom: Association for Computing Machinery, pp. 82–97. isbn: 9781450342346. doi: 10.1145/2966884.2966917. url: https://doi.org/10.1145/2966884.2966917.

Courant, Richard, Kurt Friedrichs, and Hans Lewy (Dec. 1928). "Über die partiellen Differenzengleichungen der mathematischen Physik". In: *Mathematische Annalen* 100.1, pp. 32–74. doi: 10.1007/BF01448839. url: https://doi.org/10.1007/BF01448839.

Cray Research Inc. (1976). *The CRAY-1 Computer System*. Tech. rep. Cray Research Inc.

Dagum, Leonardo and Ramesh Menon (1998). "OpenMP: an industry standard API for shared-memory programming". In: *Computational Science & Engineering, IEEE* 5.1, pp. 46–55. doi: 10.1109/99.660313.

Daiß, Gregor, Parsa Amini, John Biddiscombe, Patrick Diehl, Juhan Frank, Kevin Huck, Hartmut Kaiser, Dominic Marcello, David Pfander, and Dirk Pfüger (Nov. 2019). "From Piz Daint to the Stars: Simulation of Stellar Mergers Using High-Level Abstractions". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery. isbn: 9781450362290. doi: 10.1145/3295500.3356221. url: https://doi.org/10.1145/3295500.3356221.

Damschen, Marvin (2019). "Worst-Case Execution Time Guarantees for Runtime-Reconfigurable Architectures". PhD thesis. Karlsruher Institut für Technologie (KIT). 106 pp. doi: 10.5445/IR/1000089975. url: https://git.scc.kit.edu/CES/corq.

Damschen, Marvin, Martin Rapp, Lars Bauer, and Jörg Henkel (2020). "i-Core: A Runtime-Reconfigurable Processor Platform for Cyber-Physical Systems". In: *Embedded, Cyber-Physical, and IoT Systems: Essays Dedicated to Marilyn Wolf on the Occasion of Her 60th Birthday*. Ed. by Shuvra S. Bhattacharyya, Miodrag Potkonjak, and Senem Velipasalar. Cham: Springer International Publishing, pp. 1–36. isbn: 978-3-030-16949-7. doi: 10.1007/978-3-030-16949-7{_}1. url: https://doi.org/10.1007/978-3-030-16949-7_1.

Darema, Frederica (2001). "The SPMD Model: Past, Present and Future". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Yiannis Cotronis and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–1. isbn: 978-3-540-45417-5.

DeVito, Zachary, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek (2013). "Terra: A Multi-Stage Language for High-Performance Computing". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, pp. 105–116. isbn: 9781450320146. doi: 10.1145/2491956.2462166. url: https://doi.org/10.1145/2491956.2462166.

Dongarra, Jack, Robert Graybill, William Harrod, Robert Lucas, Ewing Lusk, Piotr Luszczek, Janice Mcmahon, Allan Snavely, Jeffrey Vetter, Katherine Yelick, Sadaf Alam, Roy Campbell, Laura Carrington, Tzu-Yi Chen, Omid Khalili, Jeremy Meredith, and Mustafa Tikir (2008). "DARPA's HPCS Program: History, Models, Tools, Languages". In: *Advances in COMPUTERS*. Vol. 72. Advances in Computers. Elsevier, pp. 1–100. doi: https://doi.org/10.1016/S0065-2458(08)00001-6. url: http://www.sciencedirect.com/science/article/pii/S0065245808000016.

Dumbser, Michael, Dinshaw S. Balsara, Eleuterio F. Toro, and Claus-Dieter Munz (2008). "A unified framework for the construction of one-step finite volume and discontinuous Galerkin schemes on unstructured meshes". In: *Journal of Computational Physics* 227.18, pp. 8209–8253. issn: 0021-9991. doi: https://doi.org/10.1016/j.jcp.2008.05.025. url: http://www.sciencedirect.com/science/article/pii/S0021999108002829.

Edwards, H. Carter, Christian R. Trott, and Daniel Sunderland (2014). "Kokkos: Enabling many-core performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. issn: 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2014.07.003. url: http://www.sciencedirect.com/science/article/pii/S0743731514001257.

Eijkhout, Victor, Robert van de Geijn, and Edmond Chow (2011). *Introduction to High Performance Scientific Computing*. http://www.tacc.utexas.edu/~eijkhout/istc/istc.html. lulu.com. isbn: 978-1-257-99254-6.

Einfeldt, Bernd (1988). "On Godunov-Type Methods for Gas Dynamics". In: *SIAM Journal on Numerical Analysis* 25.2, pp. 294–318. doi: 10.1137/0725021. eprint: https://doi.org/10.1137/0725021. url: https://doi.org/10.1137/0725021.

ElasticX10 (Nov. 2015). retrieved on January 31st 2020. url: http://x10-lang.org/documentation/practical-x10-programming/elastic-x10.html.

Espasa, Roger, Mateo Valero, and James E. Smith (1998). "Vector Architectures: Past, Present and Future". In: *Proceedings of the 12th International Conference on Supercomputing*. ICS '98. Melbourne, Australia: ACM, pp. 425–432. isbn: 0-89791-998-X. doi: 10.1145/277830.277935. url: http://doi.acm.org/10.1145/277830.277935.

Flich, José, Giovanni Agosta, Philipp Ampletzer, David Alonso, Carlo Brandolese, Etienne Cappe, Alessandro Cilardo, Leon Dragić, Alexandre Dray, Alen Duspara, William Fornaciari, Gerald Guillaume, Ynse Hoornenborg, Arman Iranfar, Mario Kovač, Simone Libutti, Bruno Maitre, José Maria Martínez, Giuseppe Massari, Hrvoje Mlinarić, Ermis Papastefanakis, Tomás Picornell, Igor Piljić, Anna Pupykina, Federico Reghenzani, Isabelle Staub, Rafael Tornero, Marina Zapater, and Davide Zoni (Aug. 2017). "MANGO: Exploring Manycore Architectures for Next-GeneratiOn HPC Systems". In: *2017 Euromicro Conference on Digital System Design (DSD)*, pp. 478–485. doi: 10.1109/DSD.2017.51.

Flynn, Michael. J. (Sept. 1972). "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9, pp. 948–960. doi: 10.1109/TC.1972.5009071.

Gärtner, Ludwig (Aug. 2016). "A GPU-based Solver for the Shallow Water Equations in SWE-X10". Bachelor's Thesis. Technical University of Munich.

George, David (2006). "Finite Volume Methods and Adaptive Refinement for Tsunami Propagation and Inundation". Dissertation. Washington: Graduate School, University of Washington. url: https://faculty.washington.edu/rjl/students/dgeorge/DLGeorgeDissertationSS.pdf.

George, David (2008). "Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation". In: *Journal of Computational Physics* 227.6, pp. 3089–3113. issn: 0021-9991. doi: https://doi.org/10.1016/j.jcp.2007.10.027. url: http://www.sciencedirect.com/science/article/pii/S0021999107004767.

Gudu, Diana-Mihaela (Nov. 2012). "Parallel Tsunami Simulations With Block-Structured Adaptive Mesh Refinement". Master's thesis.

Hager, Georg and Gerhard Wellein (2011). *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton: CRC Press. isbn: 978-0-429-19061-2. doi: 10.1201/EBK1439811924.

Hannig, Frank, Vahid Lari, Srinivas Boppu, Alexandru Tanase, and Oliver Reiche (Apr. 2014). "Invasive Tightly-Coupled Processor Arrays: A Domain-Specific Architecture/Compiler Co-Design Approach". In: *ACM Trans. Embed. Comput. Syst.* 13.4s. issn: 1539-9087. doi: 10.1145/2584660. url: https://doi-org.eaccess.ub.tum.de/10.1145/2584660.

Harris, Mark (2010). *Optimizing Parallel Reduction in CUDA*. Tech. rep. NVIDIA Corporation, Developer Technology.

Harten, Amiram, Peter D. Lax, and Bram van Leer (1983). "On Upstream Differencing and Godunov-Type Schemes for Hyperbolic Conservation Laws". In: *SIAM Review* 25.1, pp. 35–61. doi: 10.1137/1025002. eprint: https://doi.org/10.1137/1025002. url: https://doi.org/10.1137/1025002.

Hebert, Fred (2013). *Learn you some Erlang for great good!: a beginner's guide*. No Starch Press.

Heene, Mario, Alfredo Parra Hinojosa, Michael Obersteiner, Hans-Joachim Bungartz, and Dirk Pflüger (Mar. 2018). "EXAHD: An Exa-Scalable Two-Level Sparse Grid Approach for Higher-Dimensional Problems in Plasma Physics and Beyond". In: *High Performance Computing in Science and Engineering '17*. Ed. by Wolfgang Nagel, Dietmar Kröner, and Michael Resch. Springer-Verlag. isbn: 9783319683935.

Heißwolf, J., R. König, and J. Becker (July 2012). "A Scalable NoC Router Design Providing QoS Support Using Weighted Round Robin Scheduling". In: *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pp. 625–632. doi: 10.1109/ISPA.2012.93.

Heisswolf, J., A. Zaib, A. Weichslgartner, M. Karle, M. Singh, T. Wild, J. Teich, A. Herkersdorf, and J. Becker (Feb. 2014). "The Invasive Network on Chip - A Multi-Objective Many-Core Communication Infrastructure". In: *ARCS 2014; 2014 Workshop Proceedings on Architecture of Computing Systems*, pp. 1–8.

Mo-Hellenbrand, Ao (2019). "Resource-Aware and Elastic Parallel Software Development for Distributed-Memory HPC Systems". Dissertation. München: Technische Universität München.

Mo-Hellenbrand, Ao, Isaias Comprés, Oliver Meister, Hans-Joachim Bungartz, Michael Gerndt, and Michael Bader (2017). "A Large-Scale Malleable Tsunami Simulation Realized on an Elastic MPI Infrastructure". In: *Proceedings of the Computing Frontiers Conference*. CF'17. Siena, Italy: Association for Computing Machinery, pp. 271–274. isbn: 9781450344876. doi: 10.1145/3075564.3075585. url: https://doi.org/10.1145/3075564.3075585.

Heller, Thomas, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser (2017). "HPX – An open source C++ Standard Library for Parallelism and Concurrency". In: *Proceedings of OpenSuCo*. Denver, CO, USA: ACM, p. 5.

Henkel, J., A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe (2012). "Invasive manycore architectures". In: *17th Asia and South Pacific Design Automation Conference*, pp. 193–200.

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

HLRS (May 2020). *Hawk Cluster Configuration*. Webpage: https://www.hlrs.de/systems/hpe-apollo-hawk/, retrieved on 15.05.2020.

Hölzl, Wolfgang (July 2013). "Vectorization and GPGPU-Acceleration of an Augmented Riemann Solver for the Shallow Water Equations". Bachelor's thesis. Institut für Informatik, Technische Universität München. url: http://www5.in.tum.de/pub/hoelzl_bsc_2013.pdf.

Horie, Michihiro, Mikio Takeuchi, Kiyokuni Kawachiya, and David Grove (2015). "Optimization of X10 Programs with ROSE Compiler Infrastructure". In: *Proceedings of the ACM SIGPLAN Workshop on X10*. X10 2015. Portland, OR, USA: ACM, pp. 19–24. isbn: 978-1-4503-3586-7. doi:

10.1145/2771774.2771777. url: http://doi.acm.org.eaccess.ub.tum.de/10.1145/2771774.2771777.

Huffstetler, Jennifer (June 2018). *Intel Processors and FPGAs - Better Together*. website. url: https://itpeernetwork.intel.com/intel-processors-fpga-better-together/#gs.7e0k3g.

Intel Corporation (June 2011). *Intel Advanced Vector Extensions Programming Reference*. Tech. rep. Available at: https://software.intel.com/sites/default/files/4f/5b/36945, retrieved on 27.08.2020. Intel Corporation.

Intel Corporation (Sept. 2016). *Intel® 64 and IA-32 Architectures Developer's Manual*. Developer's Manual 3A. Santa Clara, USA: Intel Corporation.

InvasIC (June 2010). *Invasive Computing Miscellaneous Material*. Material from Presentations and Tech Reports of the Invasive Computing SFB. url: http://invasic.informatik.uni-erlangen.de/.

Jouppi, Norman P., Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon (June 2017). "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *SIGARCH Comput. Archit. News* 45.2, pp. 1–12. issn: 0163-5964. doi: 10.1145/3140659.3080246. url: https://doi.org/10.1145/3140659.3080246.

Kaiser, Hartmut, Maciek Brodowicz, and Thomas Sterling (Sept. 2009). "ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications". In: *2009 International Conference on Parallel Processing Workshops*, pp. 394–401. doi: 10.1109/ICPPW.2009.14.

Kalé, Laxmikant V and Sanjeev Krishnan (1993). "CHARM++: A Portable Concurrent Object Oriented System Based on C++". In: *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '93. Washington, D.C., USA: Association for Computing Machinery, pp. 91–108. isbn: 0897915879. doi: 10.1145/165854.165874. url: https://doi.org/10.1145/165854.165874.

Kalé, Laxmikant V and Gengbin Zheng (2016). "The Charm++ Programming Model". In: *Parallel Science and Engineering Applications: The Charm++ Approach*. CRC Press, pp. 1–16.

Karypis, George and Vipin Kumar (1998). "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM Journal on Scientific Computing* 20.1, pp. 359–392. doi: 10.1137/S1064827595287997. url: https://doi.org/10.1137/S1064827595287997.

Khronos Group (Apr. 2020). *SYCL Specification*. Tech. rep. KHRONOS SyCl Working Group.

Kopetz, Hermann (2011). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd ed. Real-Time Systems Series 978-1-4419-8236-0. Springer US.

Lari, Vahid, Andriy Narovlyanskyy, Frank Hannig, and Jürgen Teich (Sept. 2011). "Decentralized dynamic resource management support for massively parallel processor arrays". In: *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 87–94. doi: 10.1109/ASAP.2011.6043240.

Lee, Wonchan, Manolis Papadakis, Elliott Slaughter, and Alex Aiken (2019). "A Constraint-Based Approach to Automatic Data Partitioning for Distributed Memory Execution". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery. isbn: 9781450362290. doi: 10.1145/3295500.3356199. url: https://doi.org/10.1145/3295500.3356199.

LeVeque, Randall J. (2002). *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press. doi: 10.1017/CBO9780511791253.

LeVeque, Randall J., David L. George, and Marsha J. Berger (2011). "Tsunami modelling with adaptively refined finite volume methods". In: *Acta Numerica* 20, pp. 211–289. doi: 10.1017/S0962492911000043.

LRZ (2020a). *CoolMUC 2 Cluster Configuration*. Tech. rep. Website: https://doku.lrz.de/display/PUBLIC/CoolMUC-2. Leibniz Supercomputing Centre.

LRZ (May 2020b). *SuperMUC-NG Cluster Configuration*. Tech. rep. Webpage: https://doku.lrz.de/display/PUBLIC/Hardware+of+SuperMUC-NG, retrieved on 15.05.2020. Leibniz Supercomputing Centre.

Lukasiewycz, Martin, Michael Glaß, Felix Reimann, and Jürgen Teich (July 2011). "Opt4J - A Modular Framework for Meta-heuristic Optimization". In: *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*. Dublin, Ireland, pp. 1723–1730.

Macedo Miguel, Bruno (Nov. 2019). "A Distributed Actor Library for HPC Applications". Masterarbeit. Technical University of Munich.

Meister, Oliver (Dec. 2016). "Sierpinski Curves for Parallel Adaptive Mesh Refinement in Finite Element and Finite Volume Methods". Dissertation. München: Institut für Informatik, Technische Universität München. url: https://mediatum.ub.tum.de/doc/1320149/1320149.pdf.

Meng, Qingyu, Alan Humphrey, and Martin Berzins (2012). "The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System". In: *Digital Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis*. SC'12 –2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC 2012, pp. 2441–2448. url: http://www.sci.utah.edu/publications/Men2012b/uintah-wolfhpc12.pdf.

Mohr, Manuel, Sebastian Buchwald, Andreas Zwinkau, Christoph Erhardt, Benjamin Oechslein, Jens Schedel, and Daniel Lohmann (2015). "Cutting out the Middleman: OS-Level Support for X10 Activities". In: *Proceedings of the ACM SIGPLAN Workshop on X10*. X10 2015. Portland, OR, USA: Association for Computing Machinery, pp. 13–18. isbn: 9781450335867. doi: 10.1145/2771774.2771775. url: https://doi.org/10.1145/2771774.2771775.

Mohr, Manuel and Carsten Tradowsky (2017). "Pegasus: Efficient data transfers for PGAS languages on non-cache-coherent many-cores". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. Lausanne, CH, pp. 1781–1786. doi: 10.23919/DATE.2017.7927281.

Molzer, Andreas (Feb. 2017). "Optimierung eines Lösers der Flachwassergleichungen für heterogene GPU-Architekturen". Bachelor's Thesis. Technical University of Munich.

Monnerat, Luiz and Claudio L. Amorim (2015). "An effective single-hop distributed hash table with high lookup performance and low traffic overhead". In: *Concurrency and Computation: Practice and Experience* 27.7, pp. 1767–1788. doi: 10.1002/cpe.3342. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3342. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3342.

MPI Forum (June 2015). *MPI: A Message-Passing Interface Standard*. Vol. Version 3.1. High Performance Computing Center Stuttgart (HLRS).

Naksinehaboon, Nichamon, Yudan Liu, Chokchai Leangsuksun, Raja Nassar, Mihaela Paun, and Stephen L. Scott (2008). "Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments". In: *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 783–788. doi: 10.1109/CCGRID.2008.109.

NERSC (May 2020). *Cori Cluster Configuration*. Webpage: https://docs.nersc.gov/systems/cori/, retrieved on 15.05.2020.

Nordwall, Patrik, Johan Andrén, Johannes Rudolph, Arnout Engelen, Christopher Batay, and Helena Edelson (2011). *The Akka Actor Library Documentation*. url: https://doc.akka.io/docs/akka/current/.

NVIDIA Corporation (June 2012). *Tesla M2090 Dual-Slot Computing Processor Module*. Board Specification BD-05766-001_v03. NVIDIA Corporation.

NVIDIA Corporation (Aug. 2017). *NVIDIA Tesla V100 GPU Architecture*. Tech. rep. NVIDIA Corporation.

NVIDIA Corporation (Aug. 2020). *NVIDIA A100 Tensor Core GPU Architecture*. Tech. rep. NVIDIA Corporation.

OctoPOS API Description (May 2020). API Description generated fromt the OctoPOS Source Code through doxygen. url: gitolite@cs.fau.de:octopos-app-dev.

OctoPOS Application Development GIT Repository (May 2020). Project-internal GIT repository. url: gitolite@cs.fau.de:octopos-app-dev.

Olden, Jurek (Oct. 2018). "Performance Analysis of SWE Implementations Based on Modern Parallel Runtime Systems". Bachelor's Thesis. Technical University of Munich.

OpenMP (Nov. 2018). *OpenMP Application Programming Interface*. OpenMP Architecture Review Board. isbn: 1795759887.

Patil, Onkar, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang (2019). "Performance Characterization of a DRAM-NVM Hybrid Memory Architecture for HPC Applications Using Intel Optane DC Persistent Memory Modules". In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS '19. Washington, District of Columbia, USA: Association for Computing Machinery, pp. 288–303. isbn: 9781450372060. doi: 10.1145/3357526.3357541. url: https://doi.org/10.1145/3357526.3357541.

Petrini, Fabrizio, Darren J. Kerbyson, and Scott Pakin (2003). "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q". In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. SC '03. Phoenix, AZ, USA: Association for Computing Machinery, p. 55. isbn: 1581136951. doi: 10.1145/1048935.1050204. url: https://doi.org/10.1145/1048935.1050204.

Pierce, Paul (Jan. 1988). "The NX/2 operating system". In: *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications* 1988, pp. 384–390. doi: 10.1145/62297.62341.

Pöppl, Alexander (June 2011). "Dependent Types". Seminar Paper. Seminar Paper (Having Fun with Types '11).

Pöppl, Alexander (Aug. 2017). "Shallow Water Waves on a Deep Technology Stack: Accelerating a Finite Volume Tsunami Model using Reconfigurable Hardware in Invasive Computing". en. In: *The 10th Workshop on UnConventional High Performance Computing 2017 (UCHPC 2017)*. Universidade de Santiago de Compostela. Santiago de Compostela, Spain. url: https://mediatum.ub.tum.de/1487718.

Pöppl, Alexander (Oct. 2018). *Shallow Water on a Berkeley Stack – Actor-Based Tsunami Simulation through UPC++*. Final Presentation for author's stay at LBNL.

Pöppl, Alexander (Nov. 2019). "A UPC++ Actor Library and its Evaluation on a Shallow Water Application". en. In: *PAW-ATM: Parallel Applications Workshop, Alternatives To MPI+X*. contributed.

IEEE, ACM sigARCH. Denver, CO, USA. url: https://mediatum.ub.tum.de/doc/1531045/1531045.pdf.

Pöppl, Alexander and Michael Bader (June 2016). "SWE-X10: An Actor-based and Locally Coordinated Solver for the Shallow Water Equations". In: *Proceedings of the Sixth ACM SIGPLAN X10 Workshop (X10)*. Extended Abstract. Santa Barbara, CA, USA: ACM. doi: 10.1145/2931028.2931034.

Pöppl, Alexander, Michael Bader, and Scott Baden (Nov. 2019). "A UPC++ Actor Library and Its Evaluation on a Shallow Water Proxy Application". en. In: *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*. IEEE. Denver, Colorado, United States of America: IEEE/ACM/SigArch, pp. 11–24. doi: 10.1109/PAW-ATM49560.2019.00007.

Pöppl, Alexander, Michael Bader, Tobias Schwarzer, and Michael Glaß (Nov. 2016). "SWE-X10: Simulating Shallow Water Waves with Lazy Activation of Patches Using Actorx10". In: *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*, pp. 32–39. doi: 10.1109/ESPM2.2016.010.

Pöppl, Alexander, Marvin Damschen, Florian Schmaus, Andreas Fried, Manuel Mohr, Matthias Blankertz, Lars Bauer, Jörg Henkel, Wolfgang Schröder-Preikschat, and Michael Bader (2018). "Shallow Water Waves on a Deep Technology Stack: Accelerating a Finite Volume Tsunami Model Using Reconfigurable Hardware in Invasive Computing". In: *Euro-Par 2017: Parallel Processing Workshops*. Ed. by Dora B. Heras, Luc Bougé, Gabriele Mencagli, Emmanuel Jeannot, Rizos Sakellariou, Rosa M. Badia, Jorge G. Barbosa, Laura Ricci, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer. Cham: Springer International Publishing, pp. 676–687. isbn: 978-3-319-75178-8. doi: 10.1007/978-3-319-75178-8_54.

Pourmohseni, Behnaz, Michael Glaß, and Jürgen Teich (Mar. 2017). "Automatic operating point distillation for hybrid mapping methodologies". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1135–1140. doi: 10.23919/DATE.2017.7927160.

Rheindt, Sven, Andreas Fried, Oliver Lenke, Lars Nolte, Thomas Wild, and Andreas Herkersdorf (2019). "NEMESYS: Near-Memory Graph Copy Enhanced System-Software". In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS '19. Washington, District of Columbia: Association for Computing Machinery, pp. 3–18. isbn: 9781450372060. doi: 10.1145/3357526.3357545. url: https://doi.org/10.1145/3357526.3357545.

Rheindt, Sven, Sebastian Maier, Florian Schmaus, Thomas Wild, Wolfgang Schröder-Preikschat, and Andreas Herkersdorf (2019). "SHARQ: Software-Defined Hardware-Managed Queues for Tile-Based Manycore Architectures". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Ed. by Dionisios N. Pnevmatikatos, Maxime Pelcat, and Matthias Jung. Cham: Springer International Publishing, pp. 212–225. isbn: 978-3-030-27562-4.

Roe, Philip L. (1981). "Approximate Riemann solvers, parameter vectors, and difference schemes". In: *Journal of Computational Physics* 43.2, pp. 357–372. issn: 0021-9991. doi: https://doi.org/10.1016/0021-9991(81)90128-5. url: http://www.sciencedirect.com/science/article/pii/0021999181901285.

Roloff, Sascha, Frank Hannig, and Jürgen Teich (Feb. 2014). "Towards Actor-oriented Programming on PGAS-based Multicore Architectures". In: *Workshop Proceedings of the 27th International Conference on Architecture of Computing Systems (ARCS)*. Lübeck, Germany: VDE Verlag, pp. 1–2. isbn: 978-3-8007-3579-2.

Roloff, Sascha, Alexander Pöppl, Tobias Schwarzer, Stefan Wildermann, Michael Bader, Michael Glaß, Frank Hannig, and Jürgen Teich (2016). "ActorX10: An Actor Library for X10". In: *Proceedings of the Sixth ACM SIGPLAN X10 Workshop (X10)*. Santa Barbara, CA, USA: ACM. doi: 10.1145/2931028.2931033.

Saraswat, Vijay, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove (Feb. 2013). *X10 Language Specification, Version 2.3*. Tech. rep. IBM Research. url: http://x10-lang.org/releases/x10-release-231.html.

Saraswat, Vijay, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove (Jan. 2019). *X10 Language Specification, Version 2.6.2*. Tech. rep. IBM Research. url: http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf.

Schaffroth, Nicolai (Sept. 2015). "Simulation of Rain-Induced Floods on High Performance Computers". Master's thesis. Institut für Informatik, Technische Universität München. url: http://www5.in.tum.de/pub/Schaffroth2015_MasterThesis.pdf.

Schwarzer, Tobias, Andreas Weichslgartner, Michael Glaß, Stefan Wildermann, Peter Brand, and Jürgen Teich (Feb. 2018). "Symmetry-Eliminating Design Space Exploration for Hybrid Application Mapping on Many-Core Architectures". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.2, pp. 297–310. issn: 1937-4151. doi: 10.1109/TCAD.2017.2695894.

Shaw, David, J. P. Grossman, Joseph Bank, Brandon Batson, Adam Butts, Jack Chao, Martin Deneroff, Ron Dror, Amos Even, Christopher Fenton, Anthony Forte, Joseph Gagliardo, Genette Gill, Brian Greskamp, C. Richard. Ho, Douglas Ierardi, Lev Iserovich, Jeffrey Kuskin, Richard Larson, Timothy Layman, Li-Siang Lee, Adam Lerer, Chester Li, Daniel Killebrew, Kenneth Mackenzie, Shark Yeuk-Hai Mok, Mark Moraes, Rolf Mueller, Lawrence Nociolo, Jon Peticolas, Terry Quan, Daniel Ramot, John Salmon, Daniele Scarpazza, Ben Schafer, Naseer Siddique, Christopher Snyder, Jochen Spengler, Ping Tak Peter Tang, Michael Theobald, Horia Toma, Brian Towles, Benjamin Vitale, Stanley Wang, and Cliff Young (Nov. 2014). "Anton 2: Raising the Bar for Performance and Programmability in a Special-Purpose Molecular Dynamics Supercomputer". In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 41–53. doi: 10.1109/SC.2014.9.

Slaughter, Elliott (2017). "Regent: A high-productivity programming language for implicit parallelism with logical regions". PhD thesis. Ph. D. dissertation, Stanford University.

Slaughter, Elliott and Alex Aiken (Nov. 2019). "Pygion: Flexible, Scalable Task-Based Parallelism with Python". In: *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, pp. 58–72. doi: 10.1109/PAW-ATM49560.2019.00011.

Slaughter, Elliott, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken (2015). "Regent: A High-Productivity Programming Language for HPC with Logical Regions". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: Association for Computing Machinery. isbn: 9781450337236. doi: 10.1145/2807591.2807629. url: https://doi.org/10.1145/2807591.2807629.

Sodani, Avinash (Aug. 2015). "Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor". In: *2015 IEEE Hot Chips 27 Symposium (HCS)*, pp. 1–24. doi: 10.1109/HOTCHIPS.2015.7477467.

Spector, Alfred and David Gifford (Sept. 1984). "The Space Shuttle Primary Computer System". In: *Commun. ACM* 27.9, pp. 872–900. issn: 0001-0782. doi: 10.1145/358234.358246. url: http://doi.acm.org/10.1145/358234.358246.

Strehl, Karsten, Lothar Thiele, Matthias Gries, Dirk Ziegenbein, Rolf Ernst, and Jürgen Teich (2001). "FunState – An internal design representation for codesign". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9.4, pp. 524–544. doi: 10.1109/92.931229.

Tang, Liujuan, Vasily V. Titov, Eddie N. Bernard, Yong Wei, Christopher D. Chamberlin, Jean C. Newman, Harold O. Mofjeld, Diego Arcas, Marie C. Eble, Christopher Moore, Burak Uslu, Clint Pells, Michael Spillane, Lindsey Wright, and Edison Gica (2012). "Direct energy estimation of the 2011 Japan tsunami using deep-ocean pressure measurements". In: *Journal of Geophysical Research:*

Oceans 117.C8, p. 28. doi: 10.1029/2011JC007635. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2011JC007635. url: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2011JC007635.

Tardieu, Olivier, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri (2014). "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, pp. 53–66. isbn: 978-1-4503-2656-8. doi: 10.1145/2555243.2555245. url: http://doi.acm.org/10.1145/2555243.2555245.

Tardieu, Olivier, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, Mandana Vaziri, and Wei Zhang (Mar. 2016). "X10 and APGAS at Petascale". In: *ACM Transactions on Parallel Computing* 2.4. issn: 2329-4949. doi: 10.1145/2894746. url: https://doi.org/10.1145/2894746.

Teich, Jürgen (2008). "Invasive algorithms and architectures". In: *it–Information Technology* 50, p. 5. doi: DOI10.1524/itit.2008.0499.

Teich, Jürgen, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting (2011). "Invasive Computing: An Overview". In: *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Ed. by Michael Hübner and Jürgen Becker. New York, NY: Springer New York, pp. 241–268. isbn: 978-1-4419-6460-1. doi: 10.1007/978-1-4419-6460-1_11. url: https://doi.org/10.1007/978-1-4419-6460-1_11.

Teich, Jürgen, Jürgen Kleinöder, and Sandra Mattauch (Dec. 2015). *Invasive Computing Annual Report 2015*. Tech. rep. Transregional Collaborative Research Centre 89. Friedrich-Alexander-Universität Erlangen-Nürnberg, Karlsruhe Institute of Technology, Technical University Munich. url: http://invasic.informatik.uni-erlangen.de/publications/Annual-Report2015.pdf.

Teich, Jürgen, Jürgen Kleinöder, and Sandra Mattauch (Dec. 2016). *Invasive Computing Annual Report 2016*. Tech. rep. Transregional Collaborative Research Centre 89. Friedrich-Alexander-Universität Erlangen-Nürnberg, Karlsruhe Institute of Technology, Technical University Munich. url: http://invasic.informatik.uni-erlangen.de/publications/Annual-Report2016.pdf.

Teich, Jürgen, Alexandru Tanase, and Frank Hannig (2014). "Symbolic Mapping of Loop Programs onto Processor Arrays". In: *Journal of Signal Processing Systems* 77.1, pp. 31–59. doi: 10.1007/s11265-014-0905-0. url: https://doi.org/10.1007/s11265-014-0905-0.

Thoman, Peter, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer (2012). "Automatic OpenMP Loop Scheduling: A Combined Compiler and Runtime Approach". In: *OpenMP in a Heterogeneous World*. Ed. by Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 88–101. isbn: 978-3-642-30961-8.

Thomas, James (1999). *Numerical Partial Differential Equations*. Vol. 33. Texts in Applied Mathematics 978-1-4612-6821-5. Springer, New York, NY. doi: https://doi.org/10.1007/978-1-4612-0569-2.

Tiskin, Alexander (2011). "BSP (Bulk Synchronous Parallelism)". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, pp. 192–199. isbn: 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_311. url: https://doi.org/10.1007/978-0-387-09766-4_311.

Treibig, Jan, Georg Hager, and Gerhard Wellein (Sept. 2010). "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments". In: *2012 41st International Conference on Parallel Processing Workshops*. Los Alamitos, CA, USA: IEEE Computer Society,

pp. 207–216. doi: 10.1109/ICPPW.2010.38. url: https://doi.ieeecomputersociety.org/10.1109/ICPPW.2010.38.

Ullmann, Jeffrey David (1975). "NP-complete scheduling problems". In: *Journal of Computer and System Sciences* 10.3, pp. 384–393. issn: 0022-0000. doi: https://doi.org/10.1016/S0022-0000(75)80008-0. url: http://www.sciencedirect.com/science/article/pii/S0022000075800080.

UPC Consortium (Nov. 2013). *UPC Language Specifications, v1.2*. Tech Report LBNL-59208. Lawrence Berkeley National Lab. url: https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf.

Valiant, Leslie G. (Aug. 1990). "A Bridging Model for Parallel Computation". In: *Communications of the ACM* 33.8, pp. 103–111. issn: 0001-0782. doi: 10.1145/79173.79181. url: https://doi.org/10.1145/79173.79181.

Vázquez, Mariano, Guillaume Houzeaux, Seid Koric, Antoni Artigues, Jazmin Aguado-Sierra, Ruth Arís, Daniel Mira, Hadrien Calmet, Fernando Cucchietti, Herbert Owen, Ahmed Taha, Evan Dering Burness, José María Cela, and Mateo Valero (2016). "Alya: Multiphysics engineering simulation toward exascale". In: *Journal of Computational Science* 14. The Route to Exascale: Novel Mathematical Methods, Scalable Algorithms and Computational Science Skills, pp. 15–27. issn: 1877-7503. doi: https://doi.org/10.1016/j.jocs.2015.12.007. url: http://www.sciencedirect.com/science/article/pii/S1877750315300521.

Weichslgartner, A., D. Gangadharan, S. Wildermann, M. Glaß, and J. Teich (2014). "DAARM: Design-time application analysis and run-time mapping for predictable execution in many-core systems". In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 34:1–34:10. doi: 10.1145/2656075.2656083.

Wildermann, Stefan, Michael Bader, Lars Bauer, Marvin Damschen, Dirk Gabriel, Michael Gerndt, Michael Glaß, Jörg Henkel, Johny Paul, Alexander Pöppl, Sascha Roloff, Tobias Schwarzer, Gregor Snelting, Walter Stechele, Jürgen Teich, Andreas Weichslgartner, and Andreas Zwinkau (2016). "Invasive computing for timing-predictable stream processing on MPSoCs". In: *it - Information Technology* 58.6, pp. 267–280. doi: https://doi.org/10.1515/itit-2016-0021. url: https://www.degruyter.com/view/journals/itit/58/6/article-p267.xml.

Witterauf, Michael, Frank Hannig, and Jürgen Teich (2019). "Polyhedral Fragments: An Efficient Representation for Symbolically Generating Code for Processor Arrays". In: *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE '19. La Jolla, California: Association for Computing Machinery. isbn: 9781450369978. doi: 10.1145/3359986.3361205. url: https://doi.org/10.1145/3359986.3361205.

X10 Performance Tuning (Nov. 2015). retrieved on February 6th 2020. url: http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html.

Yang, Xuejum, Zhiyuan Wang, Jingling Xue, and Yun Zhou (June 2012). "The Reliability Wall for Exascale Supercomputing". In: *IEEE Transactions on Computers* 61.6, pp. 767–779. issn: 1557-9956. doi: 10.1109/TC.2011.106.

Zwinkau, Andreas (Apr. 2018). "Resource-aware Programming in a High-level Language – Improved performance with manageable effort on clustered MPSoCs". PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik. doi: 10.5445/IR/1000083526.

List of Figures

List of Tables

Acronyms

Actor-MPI The MPI Actor Library 103, 105, 107, 166, 172–174

Actor-UPC++ The UPC++ Actor Library 21, 49, 71, 75, 89, 93, 95–103, 105–107, 148, 163, 164, 166, 169, 171–175, 180, 189, 190, 194

ActorX10 X10 Actor Library 31, 39, 49, 52, 67, 71, 75, 81, 83, 85–87, 89–91, 94, 96, 98, 100, 105, 106, 137, 146, 148, 163, 166–168, 171, 175, 180, 185, 190, 192, 194

ADER-DG Arbitrary High-Order Discontinuous Galerkin 3

AGAS Active Global Address Space 44

APGAS Asynchronous Partitioned Global Address Space 31, 32, 81, 175

μArch micro-architecture 146, 148, 168

AVX-2 Advanced Vector Extensions 2 129, 166

AVX-512 Advanced Vector Extensions 512 7, 8, 166

BSP bulk synchronous parallel 4, 10, 11, 121, 124, 132, 137, 145, 146, 148, 163, 166, 180

ccNUMA cache-coherent Non-Uniform Memory Access 9

CFL condition Cauchy-Friedrichs-Lewy Condition 115, 116, 124

CI *i*-Core Custom Instruction 64, 153–155, 157, 158, 160

C*i*C core *i*-let controller 61, 62

DFG German Research Foundation 51

DMA Direct Memory Access 57, 180

DSE Design Space Exploration 67, 68

DSL Domain-Specific Language 41, 64

DSP digital signal processor 160, 161

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg 52

FIFO First In First Out 75

FPGA field programmable gate array 64, 160

FSM Finite State Machine 78, 79, 83, 85, 91, 101, 105, 107, 142, 143, 175, 180, 185

GASNet-EX Global Address Space Networking for Exascale 22, 103, 172, 173

GEBCO General Bathymetric Chart of the Oceans 138

HAM Hybrid Application Mapping 66

HPX High Performance ParalleX 41, 44, 122, 127, 129, 131, 132, 135, 136, 220

***i*-Core** Invasive Core 60–62, 64, 65, 153–155, 157–161, 175, 179, 223

***i*-let** invade-let 53, 56–58, 61, 62, 155–159, 223

***i*MPI** Elastic MPI 59

***i*NoC** invasive Network-on-Chip 60, 61, 63

InvadeX10 Invasive X10 Language 52, 55, 57, 59, 155

InvasIC Transregional Collaborative Research Center Invasive Computing 4, 31, 51, 53, 55, 59, 64, 81, 105, 145, 153, 161, 175

***i*RTSS** invasive Runtime Support System 52, 53, 55, 66, 69, 153

ISA Instruction Set Architecture 62, 157

KIT Karlsruhe Insitute for Technology 52

LEON 3 Sparc LEON 3 60–62, 64, 65, 106, 155, 157, 160, 161

LPC Local Procedure Call 27, 92, 95, 96

LRZ Leibniz Supercomputing Centre 60, 129, 172

LUT lookup table 160, 161

MIMD Multiple Instruction Multiple Data 8

MISD Multiple Instruction Single Data 8

MPI Message Passing Interface 3, 13–18, 21, 22, 41, 44, 105, 121, 122, 127, 128, 130–132, 135, 138, 166, 175, 180

MPSoC Multiprocessor System-on-Chip 31, 60

NA network adapter 61, 62

NERSC National Energy Research Scientific Computing Center 89, 166, 202

NoC Network-on-Chip 60–62

NUMA Non-Uniform Memory Access 21, 81, 95

OctoPOS OctoPOS Parallel Operating System 52, 55–59, 155, 156

μ**Op** micro-operation 154

PDE partial differential equation 112, 113, 116, 142, 175

PE Processing Element 56, 62, 64, 65, 153

PGAS Partitioned Global Address Space 10, 21, 31, 41, 44, 89, 139, 179

μ**Program** Micro-Program 64, 154, 155

RDMA Remote Direct Memory Access 21

RMA Remote Memory Access 128

RPC Remote Procedure Call 23–27, 89, 90, 92, 95, 96, 128, 171

sam(oa)^2 Space-Filling Curves and Adaptive Meshes for Oceanic and Other Applications 60

SIMD Single Instruction Multiple Data 3, 8, 11, 144, 146

SIMT Single Instruction Multiple Thread 8

SISD Single Instruction Single Data 7, 8

SPMD Single Program Multiple Data 8, 13, 14, 21, 89, 94, 194

SWE Shallow Water Equations Software Package 48, 106, 111, 121, 122, 125, 127, 130, 135–138, 140, 143–146, 148, 150, 163, 164, 166, 168–170, 172, 174, 175, 201

SWE-PPM Shallow Water Equations Parallel Programming Models Evaluation Software Package 41, 121, 122, 126, 127, 129, 131, 133, 134

SWE-X10 Shallow Water Equations in X10 31, 120, 121, 137–140, 142–150, 153–155, 158, 161, 163–169, 175, 176, 179, 180, 201

TCPA tightly coupled processor array 60–62, 64, 65, 153

TLM tile-local memory 61, 64, 65, 153, 154, 158, 160

TUM Technical University of Munich 52

UMA Uniform Memory Access 9

UPC++ Unified Parallel C++ 10, 21–29, 31, 37, 41, 44, 71, 89, 91, 92, 95, 96, 98, 103, 106, 122, 127, 128, 130–132, 163, 167, 171, 172, 175, 194

VLIW Very Long Instruction Word 62

WCET worst-case execution time 68