Technical University of Munich
Department of Informatics

Bachelor's Thesis in Informatics

Exploring Modern Runtime Systems for the SWE-Framework

Martin Bogusz

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Exploring Modern Runtime Systems for the SWE-Framework

Evaluation von modernen Laufzeitsystemen anhand des SWE-Frameworks

Author:	Martin Bogusz
Supervisor:	Prof. Dr. Michael Bader
Advisors:	M.Sc. Alexander Pöppl, M.Sc. Philipp Samfaß
Submission date:	15.09.2019

Statement

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

Date

Martin Bogusz

Abstract

Nowadays, processor development drives towards an increasing number of logical cores per processing unit. This leads to a growing need for concurrent execution to improve performance of applications. As synchronization and communication are complex tasks in a multi-core environment, parallelization frameworks are needed. In this thesis, we explored MPI, UPC++, Charm++, OpenMP and HPX by utilizing their concepts on a Tsunami approximation model - the SWE-Framework. Implementations were benchmarked on the Cool-MUC2 massively parallel processor with Intel "Haswell" nodes. We measured performance, computation and communication time for strong and weak scaling scenarios on up to 896 processing elements. Overall, MPI performed best in terms of performance and scaling. UPC++ demonstrated stable communication time with increasing number of ranks, but showed significantly higher reduction and synchronization costs. Overdecomposition of Charm++ Chares did not lead to performance improvement on load-imbalanced scenarios, as communication overhead exceeded migration benefit. HPX showed best performance when utilizing two concurrent tasks per processing core, but overall performed slower than all other frameworks. Concluding, the HPX implementation could be further improved by adapting to a better fitting parallel concept. Best performance results could be achieved by utilizing a hybrid UPC++/MPI solution.

Contents

Acknowledgement

I would like to thank my advisors Alexander Pöppl and Philipp Samfaß for their extensive support and guidance of this bachelor's thesis. Furthermore, I wish to thank my sister Barbara Bogusz and my friends Jonas Mayer and Catherine Feng for their constructive criticism of the manuscript.

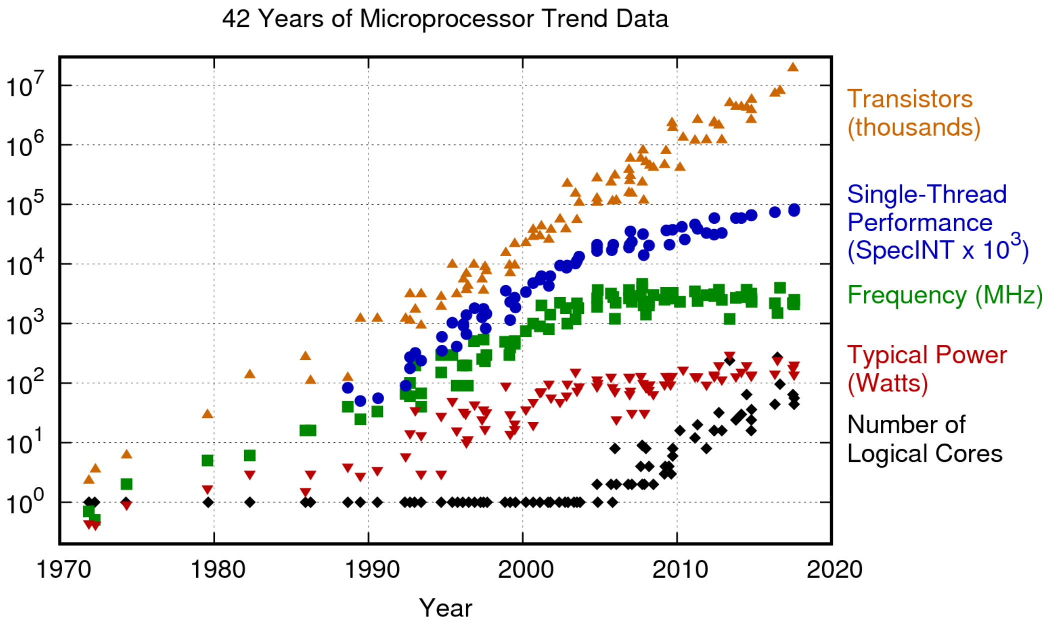
1. Introduction

Over the last 40 years, computational power of single processing units increased as dictated by Moore's Law. This tasks got increasingly more challenging, as processors evolved and therefore, the architectures became more complex and smaller. This lead to the point, where it is significantly more challenging to improve the performance of a single processor, than to increase the amount of logical cores. As shown in figure 1.1, this trend is continuing, while the performance gains of single-threads are declining.

From a software development point of view, it is now necessary to not only optimize code for single processor execution, but to think of ways to parallelize the algorithms, so action can be run simultaneously on many cores. While adjusting algorithms for parallel execution is already a challenging task, synchronization and communication of concurrently running cores present a demanding assignment itself. Especially, when computation runs on massively parallel supercomputers, where thousands of cores are available and distributed over a network.

For this purpose, various frameworks were developed, which created a programmable interface to use a specific concurrent concept to parallelize execution. The frameworks, could be then utilized to handle the communication, synchronization and prevent race-conditions between parallel running cores. While there is a variety of legacy frameworks, which have existed for over 20 years and are continuously improved and developed, the demand for solutions addressing the needs of modern systems, remains. Especially the always increasing gap between communication and processing power requires new approaches to ensure full utilization of contemporary multi-core processors.

In this thesis, we study state-of-the-art parallelization models by exploring



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Figure 1.1: Processor trend of the last 40 years. Source: [18]

corresponding frameworks. This is achieved by utilizing each framework for a real computational problem - the Shallow Water Teaching Code. It is a two-dimensional structured grid problem implementing a numeric solution to discretized Shallow Water Equations for Tsunami approximation. Resulting implementations are benchmarked on a massively parallel processor. The obtained results are evaluated regarding various measurands, including computing performance, scaling abilities but also implementation cost and convenience.

2. Prior and Related Work

This thesis analyzes different parallel implementations of the Shallow Water Code using OpenMP, MPI, UPC++, Charm++ and HPX. It is based on the SWE-Framework[20], which was introduced and designed by Alexander Breuer and Michael Bader in ”Teaching parallel programming models on a shallow-water code”[5]. The proposal describes a software package implementing a simple finite volume solver for Shallow Water Equations on a Cartesian grid for teaching purposes. Additionally, the introduced package provides parallel implementations using MPI, CUDA, OpenMP and hybrid solutions. The provided MPI and OpenMP approaches are used and evaluated in this thesis. The collection was later extended by UPC++ and Charm++ implementations, which are reused for the purpose of this work. Those extensions were introduced and tested in ”Performance Analysis of SWE Implementations based on modern parallel Runtime Systems”[17] by Jurek Olden.

While MPI and OpenMP are widely used and their scaling and performances capabilities well studied, this is not particularly true for other frameworks used in this thesis. Additionally, MPI serves as the baseline in most other studies, to which other frameworks are referenced and compared.
Several papers have studied UPC++'s PGAS implementation. In Shan et al. [21] UPC++'s one-sided communication was applied to an adaptive mesh framework. This resulted in a 60% performance increase compared to an implementation using MPI's two-sided communication and also outperformed the respective one-sided MPI solution. In Hashmi [9] a hybrid MPI/UPC++ model is proposed, which shall improve efficiency by combining proven concepts of message passing and the global address space model. A performance benchmark implementing a Gauss-Seidel based two-dimensional heat diffusion kernel shows up to 30% improvement over pure MPI or UPC++ imple-

mentations.

Charm++'s runtime model was utilized in few publications. In Sun et al. [22] Charm++ was used with a user Generic Network Interface (uGNI) on a new Cray XE/XK system. Benchmarking showed a 50% better message latency than the standard MPI implementation. Compared on communication-intensive application it was up to 70% faster than MPI. Bak et al. [3] showed how to use the Charm++ runtime system to mitigate load imbalances in a hybrid approach with OpenMP. The hybrid approach offered up to 46.5% improvement.

HPX in its present state is yet to be explored fully. There are few performance analyses of this relatively new framework. Grubel et al. [8] evaluated HPX applications by adjusting task sizes of asynchronous actions to minimize overheads and optimize performance. Kathami et al. [15] were able to implement an HPX solution to a N-Body problem improving scalability by 28% compared to an hybrid MPI + OpenMP implementation and up to 48% over single node OpenMP.

As shown, each of the frameworks can excel in a specific problem domain. However, we lack comprehensive reviews which are necessary for selecting a fitting parallelization model. This thesis will try to provide a solution to this deficiency.

3. Parallel Frameworks

3.1 OpenMP

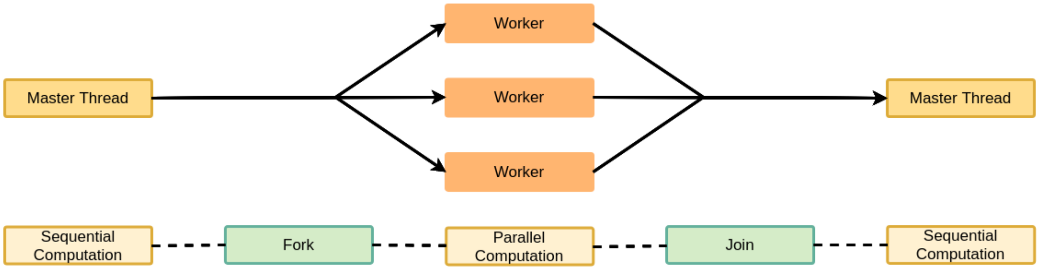


Figure 3.1: Example of a fork-join scheme.

OpenMP[6] represents the current standard for shared memory parallelization. It employs an API providing an abstract interface for platform-independent use. OpenMP utilizes C and C++ preprocessing macros, which are used as directives to parallelize designated sections of an application. The framework employs a simple, old-fashioned fork-join scheme. As shown in figure 3.1, a fork-join model consists of a main-thread and multiple worker-threads. The main-thread generates workers(fork), which simultaneously execute a task. After their termination(join) the main-thread continues sequential execution. The advantage of this framework lies upon the effective use of shared-memory in multi-core processor architectures. While other models require explicit communication, such as the message passing model, OpenMP can exploit the direct-memory access and cache-coherence to its full potential. Thus, it is likely to perform better on single nodes.

```
#pragma omp for
for(int n=0; n<10; ++n){
        printf(" %d", n);
}
```

Listing 3.1: Example of a parallelized loop with OpenMP.

The common use-case of OpenMP is *for-loop* parallelization as demonstrated in listing 3.1. The loop iterations would be divided and distributed on a specified number of workers, which predominantly equals the number of available processing cores. However, it is often necessary to access the same data, e.g. objects or variables in each loop iteration. This can lead to race-conditions in parallel execution.

```
int x = 5;
#pragma omp parallel private(x)
{
        x = x+1;
        printf(" %d", x) //Prints 6
}
```

Listing 3.2: Example of the private directive in OpenMP.

Hence, OpenMP provides directives to specify policies for shared data. Listing 3.2 demonstrates the use of the *private* directive. This semantic creates a copy of the specified entity for each worker. Additionally, reduction operations are supported, e.g. maximum and sum, for shared data as shown in listing 3.3. In our implementations, OpenMP is used in hybrid approaches. Those utilize OpenMP parallelization within a rank, i.e. shared-memory environment, and use a suitable distributed memory framework atop. Although OpenMP was first developed in 1998, it is still widely employed and continuously improved.

```
int x = 5;
#pragma omp parallel reduction(+:x)
{
        x = x+1;
}
printf(" %d", x); //Prints 6*(Number of Workers)
```

Listing 3.3: Example of the reduction directive in OpenMP.

3.2 MPI

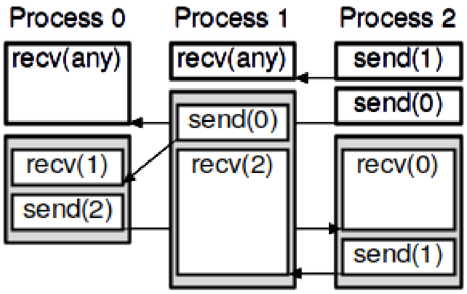


Figure 3.2: Example of message passing. Source: [24]

MPI [24] was created as a specification for a message passing interface. Nowadays, it is the standard approach in message passing for concurrent applications on distributed memory systems. It is a collection of proven and tested concepts of many message passing systems. MPI is supported by most architectures and various implementations are available, such as MPICH and OpenMPI. The idea of the message passing model is to have processes with separate address space. MPI specifies routines for such processes to communicate and synchronize. A common approach to utilize this model for parallel computing is to divide a problem into subtasks, which can be simultaneously carried out on a system. The tasks would synchronize and exchange data, e.g. parts of a Cartesian grid, by message passing. Tasks are represented in MPI by processes and belong to process-groups. In each group, every process is identified by a unique rank, which is a number between 0 and $n-1$, where n is the total number of processes in a group. An MPI application can therefore consist of multiple groups, which consist of multiple processes. A usual topology can therefore vary between a single processing-unit or a multi-node network. To synchronize and exchange data between processes, MPI specifies API calls - most importantly - routines for point-to-point communication, in blocking and non-blocking manner. This action is cooperative and therefore only occurs when the sending process invokes a *send* operation and the receiving process a *receive* operation. To distinguish different incoming messages, MPI specifies so called *tags*. That is to say, a *receive* call with a specified *tag* only matches with a message on the same communicator from a corresponding *send* call. Additionally, it is required to specify a destination and source, i.e. the rank of the sending and receiving process, in a communication-block.

```
MPI_send(address, length, destination, tag);
MPI_receive(address, length, source, tag);
```

Listing 3.4: MPI send and receive operations.

Furthermore, MPI specifies collective communication routines such as *reduce* and *scan*, collective data movement routines including *broadcast* and *gather*[24] as well as global synchronization with *barrier*. Although one-sided communication was not foreseen in the original MPI specification, it was added in the MPI-2 extension. MPI lacks any kind of active messages. Additionally, the framework's low-level design does not employ a runtime system, which would allow multi-threading or management of processes. Hence, it is commonly used in hybrid approaches with shared memory parallelization, such as OpenMP.

3.3 UPC++

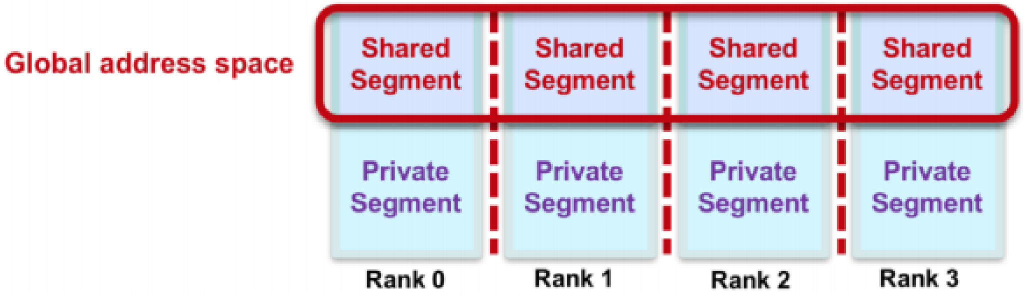


Figure 3.3: The PGAS model. Source: [1]

UPC++[25] is a full parallel framework based on the PGAS model. As shown in figure 3.3 PGAS, or Partitioned Global Address Space, is a memory model that assumes a global address space over all processing ranks. Additionally, a part of the address space is affine to a process and can therefore be optimized for access.

```cpp
upcxx::global_ptr<float> data = upcxx::new_array<float>(5);
//Defining an Array in the shared Segment
```

Listing 3.5: Example of shared data in UPC++.

UPC++ utilizes this model by providing an additional shared memory segment to the private address space that each process naturally has. As shown in figure 3.3, the shared segment can be accessed by any process. UPC++ provides various API calls to use the global space. The example code in listing 3.5 makes use of a provided special constructor to explicitly declare an array in the global segment. Such calls return a global pointer, which are used to reference objects in the shared segment. Also, objects of the local shared segments can be downcasted and used as usual C++ objects. Furthermore UPC++ implements Remote Memory Access(RMA) and Remote Procedure Calls (RPC). These concepts are responsible for communication and data-exchange between processes. RMA operations are explicit and one-sided, e.g. *rput* or *rget*. This means that there is no need for matching *send* and *receive* calls, and therefore are significantly different to MPI's two-sided point-to-point communication. In addition, the low overhead of RMA encourages fine-grained communication instead of heavy-weight data-exchange as used in message passing. Remote Procedure Calls are used to invoke functions on remote processes. They can be used to delegate parts of computation to other locations. Additionally, UPC++ implements an asynchronous future-concept for the RMA and RPC semantics. Futures are proxy-objects, which are promising to return a value in an unspecified amount of time. By calling *wait* on the future-object, a blocking task is invoked which waits until the value is returned. This semantic encourages the programmer to potentially overlap unrelated work during future completion. As MPI, UPC++ provides collective communication and synchronization semantics, including *reduce*, *broadcast* and *barrier*. UPC++'s design strongly emphasizes asynchronous execution and thus discourages the use of global barriers. The communication-layer is built atop the high-performant communication library GASNet-Ex [4]. GASNet-Ex is used by state-of-the-art PGAS implementing languages such as Unified Parallel C (UPC) and Co-Array Fortran. UPC++ does not implement any semantic for process creation or control, and therefore the number of ranks remains constant during the execution of the program.

3.4 Charm++

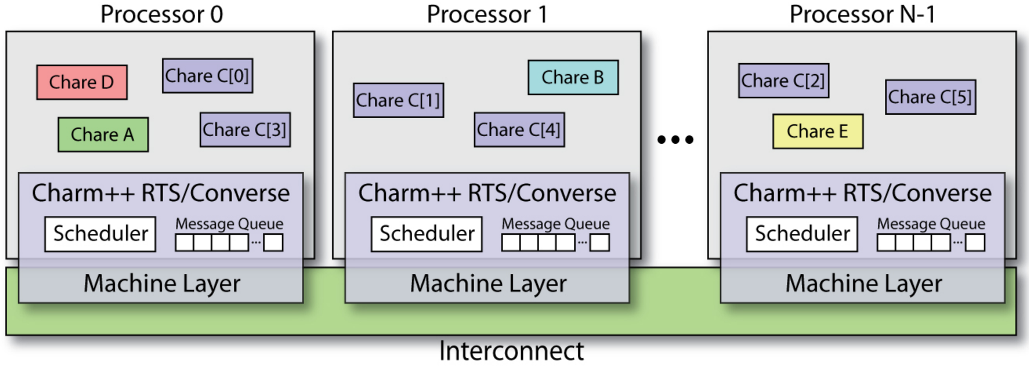


Figure 3.4: Charm++ runtime system. Source: [11]

Charm++ [14] is significantly different from the MPI or UPC++ parallelization approach. While both, MPI and UPC++, employ a more static parallelization model, in which each process is assigned to a certain task over the whole execution, Charm++ implements an asynchronous *message-driven* model. As a result, work is dynamically created and managed by its own runtime system (RTS). Charm++ further implements this concept by providing objects, so called Chares. A Chare is a capsuled C++ class, which is managed by the Charm++ runtime system and is migratable, thus it can be moved across processing units. A Chare-object contains member-functions and variables like any other C++ class, with the difference that they are callable in an asynchronous matter by other Chares. Actions of a migratable object are invoked by sending non-blocking messages. The RTS then schedules the object on a processor, hence message-driven scheduling occurs. To implement Chares, Charm++ employs interface files. In those, the programmer specifies entry methods and events, which are invoked by messages. Furthermore, Charm++ implements a variety of collective communication routines. While they provide the same functionality as MPI collectives, e.g. *MPI_Allreduce*, they are utilizing the message-driven concept. For such operations a call-back function is specified. Each participant then contributes its data to the collective by sending a message. The call-back is asynchronously invoked as soon as the global operation is completed. Additionally, the framework's design and runtime system enables the use of over-decomposition, i.e. dividing a computational task into more subtasks than processing-units available. For Charm++ this means to use more, but smaller Chares. This design leads to flexibility in scheduling and may improve performance through better cache utilization. In addition, the framework employs various load-balancing methods and enables the implementation of custom strategies. Charm++ divides load-balancing strategies in 3 groups - centralized, distributed and hierarchical balancing. In centralized approaches, load information and communication topology is gathered in a single-point, where migration strategies are evaluated based on the gathered information. Applications using distributed strategies only exchange load information among neighbouring nodes, while hierarchical approaches organize processors within groups, which are independently balanced. The default strategy used by Charm++ is a centralized "greedy balancing", i.e. assigning the heaviest computation to the least loaded processor. Other available strategies include centralized topology-aware, communication-aware and distributed neighbour balancing. On top, most load-balancers can be specified at runtime, and thus do not require recompilation. This encourages testing for optimal balancing strategies, which subsequently can be fine-tuned with over-decomposition.

3.5 HPX

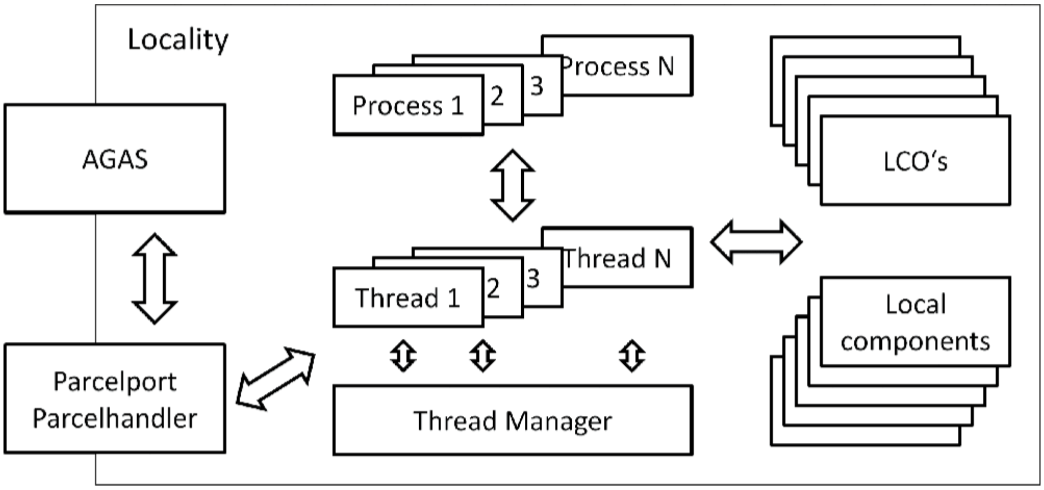


Figure 3.5: Overview of a HPX locality. Source: [13]

High Performance ParalleX(HPX)[10] is a library which comes with a runtime component that supports concurrent and parallel concepts. While it implements some message-driven semantics like Charm++ it is entirely different to any parallelization approach on the market. As the name suggests, the framework implements the so called ParalleX[13] execution model. It is fairly new and therefore designed to meet today's technological requirements by focusing on efficiency, scalability, fault tolerance, power consumption and programmability. In its own, it is not a radical new approach to solve those objectives but it is rather a collection of proven concepts derived from other models. Hence, the idea is to address common high-performance-computing bottlenecks such as starvation, high overheads and latency. In sum, HPX tries to improve efficiency by reducing the use of global barriers and synchronization by providing flow-control. It aims to increase scalability by employing message-driven remote procedure calls and emphasizes the use of fine-grained asynchronous tasks. As shown in figure 3.5, a key concept of ParalleX is to divide the system into local physical domains, which might be a single chip or a node, called *localities*. HPX then provides a runtime component for each respective locality and communication features for inter-locality operations.

3.5.1 Thread-Manager

As briefly noted, HPX tries to improve scalability by decreasing overheads of parallel tasks. This is achieved by implementing low overhead user-space threads and providing a thread-manager for scheduling. The thread-manager schedules tasks and maps them on the underlying processing-units or OS-threads. It supports various scheduling policies and uses a task-stealing policy as default, where for each underlying processing-core one task queue is maintained. Asynchronous tasks are evenly distributed over the queues by the thread-manager. If a queue is empty the respective core then "steals" tasks off other queues within a locality. Other scheduling policies feature static scheduling or a hierarchical approach. The resulting low overhead encourages the extensive use of threads. Local control objects provide various mechanisms to assign tasks and specify dependencies amongst them.

3.5.2 Local Control Objects

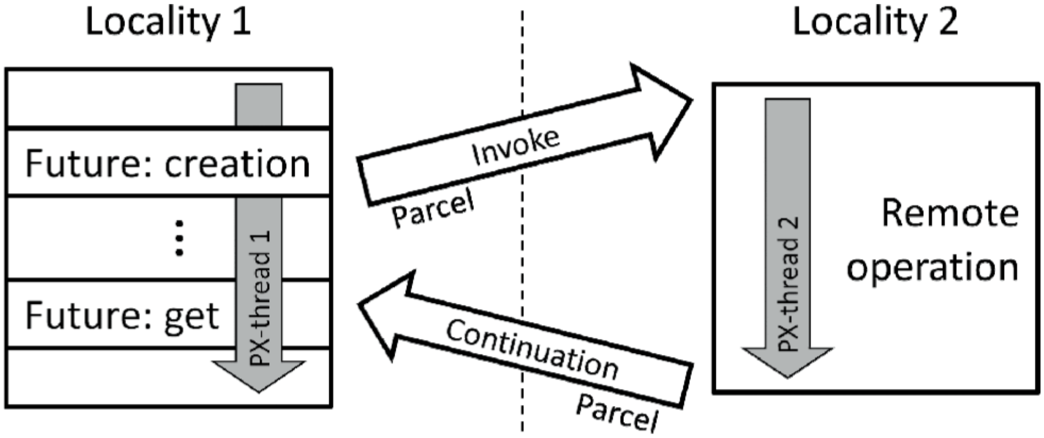


Figure 3.6: HPX remote action call. Source: [13]

Local control objects (LCO) are derived from the future-concept, as introduced in many parallel runtime systems. In essence, a local control object is returned by invoking any kind of HPX-specific asynchronous action. Such tasks, may even be executed on a remote location by sending a parcel as seen in figure 3.6, which will be evaluated more in-depth in the following subsections. The future-concept is represented by the *hpx::future* class. It is a templated C++ object-handle to a task which is promising to return a specified value when executed. By performing a *get* operation on the future a blocking call is invoked, which waits for the task to terminate. This semantic enables the programmer to explicitly control the workflow. He can decide whether to wait on a task to be completed, but also has the possibility to overlap unrelated work in the mean time. Furthermore, HPX provides semantics to group futures and wait on them collectively. An important feature extending the LCO-concept is flow-control. This functionality can be used to describe dependencies among asynchronous tasks and letting the runtime system dynamically schedule them when their requirements are met. HPX implements flow-control by providing *hpx::dataflow*. A *hpx::dataflow* is a function returning a future, which specifies an action to be invoked on completion of a single or a set of futures. This semantic allows to specify data-dependencies and to chain asynchronous tasks together while utilizing the HPX runtime system, i.e. thread-manager, without ever worrying about data-availability.

3.5.3 Active Global Address Space

As shown in figure 3.5, each locality is connected to the active global address space. The idea is similar to UPC++s PGAS approach - each entity can access the global address space to reference remote data and action. The main difference is that AGAS does not employ partitioned segments for each processor. It rather provides unique identifiers to reference global entities. Global entities can serve two functions. In the simplest case, a global entity is a locality. Thus, the identifier can be used remotely to invoke action on the corresponding physical location. In the other case, it is a global migratable object; so called component. A component is a capsuled class consisting of a set of actions. It is similar to Charm++s Chare, as components can be migrated across localities. The global identifier of each component is actively updated, so that the link is maintained even after the object moved to another physical location. Components can be manually migrated, however, it is natively supported by HPX's dynamic load balancer. Usually, a component object is implemented as a Client-Server class, where a local interface is provided which invokes a remote action call. Communication in the global address space is realized by parcel communication.

3.5.4 Parcel Transport Layer

Parcels are active messages, solely used by the runtime system to invoke asynchronous actions on members of the active global address space. Comparing to other message-driven implementations, parcels specify continuation. A continuation is a follow-up action for the remote entity to invoke after it finished the asynchronous task. This is an essential design decision, so that the future interface can be used equally for remote and local asynchronous tasks. As seen in figure 3.6, the remote operation is invoked by a parcel. As with local futures, the *get* operation can be called to enter a blocking call. The difference is however, that future completion is signalled by the continuation parcel. On each locality a respective Parcelhandler resolves incoming and outgoing parcels and notifies the thread-manager upon their arrival. Naturally, this concepts can be used with any other LCO, such as *hpx::dataflow*.

3.5.5 C++ parallel Algorithms

In addition to its framework, HPX provides a full collection of the parallel algorithms specified in the C++17 standard. The algorithms are divided in 4 subgroups: Non-modifying, modifying, sorting and index-based. The most common non-modifying algorithm is *for_each*, which iterates over a container and applies a function on each element. Other non-modifying operations are *search* and *scan*. Modifying algorithms include *copy*, *replace*, *transform* and *fill*. Index-based algorithms represent iterative *for-loops*. The algorithms are fully integrated in the HPX framework, as they are implemented atop of its runtime system. Listing 3.6 demonstrates the use of a parallel index-based loop. As shown, an execution policy needs to be specified. The standard foresees different policies, however, HPX supports only parallel and sequential execution.

```
hpx::parallel::for_loop(
        hpx::parallel::execution::par,//Execution policy
        0,          //Start Value
        100,        //End Value
        [](int x)   //Function called each iteration
        {
                hpx::cout << " " << x ;
        }
);
```

Listing 3.6: Example of a parallel for-loop in HPX.

3.6 Frameworks by Comparison

MPI and UPC++ employ a similar approach. While their communication differs, as MPI makes use of message passing and UPC++ utilizes a partitioned global address space, both rely on a static distribution of data and tasks across processing units and lack a runtime system. This design leads to the creation of heavy-weight concurrent processes which frequently exchange data or synchronize. Charm++ and HPX are significantly different, as they are higher-level parallelization frameworks utilizing a runtime system. Charm++'s unit for parallel computation is a Chare, which can be dynamically created at runtime. Chares are exposed to Charm++'s runtime system which schedules and migrates them across the available processing elements and nodes. In addition, synchronization is not required as every action is invoked by active messaging. The HPX model builds on the full exposer of its runtime system. It emphasizes dynamic fine-grained asynchronous execution within a physical domain over the use of static work distribution on processes. Additionally, moving work to data is preferred over exchanging data between localities.

4. SWE-Framework

The SWE-Framework[20] is a codebase gathering different parts for simulation of wave based scenarios. It utilizes a discretized model of Shallow Water Equations and provides different components and helper structures for simulation.

4.1 Shallow Water Equations

Shallow Water Equations are a set of hyperbolic partial differential equations forming a non-linear system, which is used to model wave motion in space and time.

These equations (4.1) are derived from conservation laws - the equation for water height h derives from the law of conservation of mass. Water velocity in horizontal and vertical direction, hu and hv, from the law of conservation of linear momentum. g is the gravitational constant - $9.81\frac{m}{s^2}$.

$$\frac{d}{dt}\begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \frac{d}{dx}\begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix} + \frac{d}{dy}\begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix} = 0 \tag{4.1}$$

This system assumes no coriolis forces, friction or bathymetry. To model those factors a source-term $S(x, y, t)$ has to be added as seen in equation 4.2. The implementation of the Shallow Water Equations in the SWE-Framework only models bathymetry b, i.e ocean depth, as an additional factor (4.3).

$$\frac{d}{dt}\begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \frac{d}{dx}\begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix} + \frac{d}{dy}\begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix} = S(x, y, t) \tag{4.2}$$

$$S(x,y,t) = \begin{bmatrix} 0 \\ -\frac{d}{dx}(ghb) \\ -\frac{d}{dy}(ghb) \end{bmatrix} \tag{4.3}$$

$$\frac{d}{dt}q + \frac{d}{dx}F(q) + \frac{d}{dy}G(q) = S(q,x,y,t) \tag{4.4}$$

$$\begin{aligned} Q_{i,j}^{n+1} - Q_{i,j}^{n} = {} & \frac{\Delta t}{\Delta y}(F(q(x_{i+\frac{1}{2}},y,t_n)) - F(q(x_{i-\frac{1}{2}},y,t_n))) \\ & + \frac{\Delta t}{\Delta x}(G(q(x,y_{j+\frac{1}{2}},t_n)) - G(q(x,y_{j-\frac{1}{2}},t_n))) \end{aligned} \tag{4.5}$$

The shallow water model is solved by applying finite volume discretization[16]. This approach transforms the continuous domain into a discrete space of equally sized rectangular cells with a continuous time domain. Further discretization of the time domain leads to an explicit time stepping scheme as shown in equation 4.5, which is the computational model used in the SWE-Framework. The solution of the q-terms are obtained by solving one-dimensional Riemann problems between neighbouring cells.

4.2 Domain Decomposition and Discretization

$$Q = (h, hu, hv)^t \tag{4.6}$$

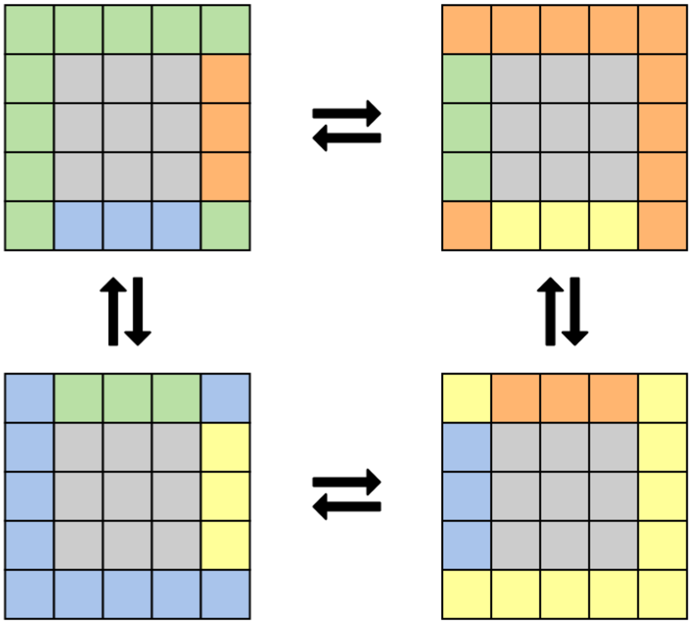


Figure 4.1: Visualization of a Ghostlayer exchange.

The domain consists of a rectangular grid of cells. Each cell is assigned three states: h as water height and hu, hv as momentum in vertical and horizontal direction. The border of the domain is represented by the Ghostlayer. A Ghostlayer in the most simple way represents the end of the simulation domain. Before each iteration, the value of the Ghostlayer is set accordingly to the scenario. Thus, a water outflow, inflow or reflection from the simulation border can be simulated. Additionally, the Ghostlayer can be used to connect independent domains together. As shown in 4.1, the cell states of the Ghostlayer of neighbouring domains are exchanged at the beginning of each iteration. This concept is mainly utilized in parallel execution, where each independent domain represents one patch of the whole simulation grid.

4.3 Updating Scheme

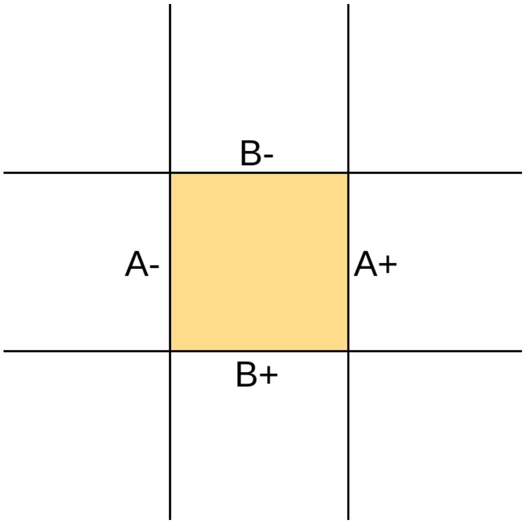


Figure 4.2: Visualization of the net-updates.

The SWE-Framework implements a scheme to update cell states for each step of the simulation. As shown in equation 4.5, first the Riemann-Problems between neighbouring cells are solved in horizontal and vertical direction. Each solution computes net-updates which are used to update the domain for the next iteration, i.e. timestep. The SWE-Framework supports different approaches for updating, but mainly uses dimensional splitting.

$$\begin{aligned} Q^*_{i,j} &= Q^n_{i,j} - \frac{\Delta t}{\Delta x}(A^+\Delta Q^n_{i-\frac{1}{2},j} + A^- Q^n_{i+\frac{1}{2},j}) \\ Q^{n+1}_{i,j} &= Q^*_{i,j} - \frac{\Delta t}{\Delta y}(B^+\Delta Q^*_{i,j-\frac{1}{2}} + B^- Q^*_{i,j+\frac{1}{2}}) \end{aligned} \tag{4.7}$$

Figure 4.3: Dimensional Splitting

The dimensional splitting approach divides the solution domain into a horizontal and a vertical sweep. A respective solver (see 4.4.3) solves the Riemann-Problems and calculates updates for each pair of cells. As shown in figure 4.2, A^+ and A^- represent the horizontal and B^+ and B^- the vertical updates. Subsequently, the updates are used to calculate the cell states for the next simulation step, i.e. water height and the respective momentum of each cell. This updating scheme is used in all parallel implementations of the SWE code.

Timesteps for each simulation step are derived from the maximum wave speed, which is returned by the solver after computing the updates. This is necessary, so that the computed wave does not break the Courant-Friedrichs-Lewy (CFL) condition, i.e. does not travel further than one cell per iteration. Since the solver only determines the maximum wave speed of neighbouring cells it must be reduced over the whole domain, thus, in case of a divided domain, over all parts.

4.4 SWE Components

A simulation requires few components which are provided by the SWE-Framework. In essence, it consists of a simulation scenario, a computational block and main function. An overview of the system design is shown in figure 4.4.

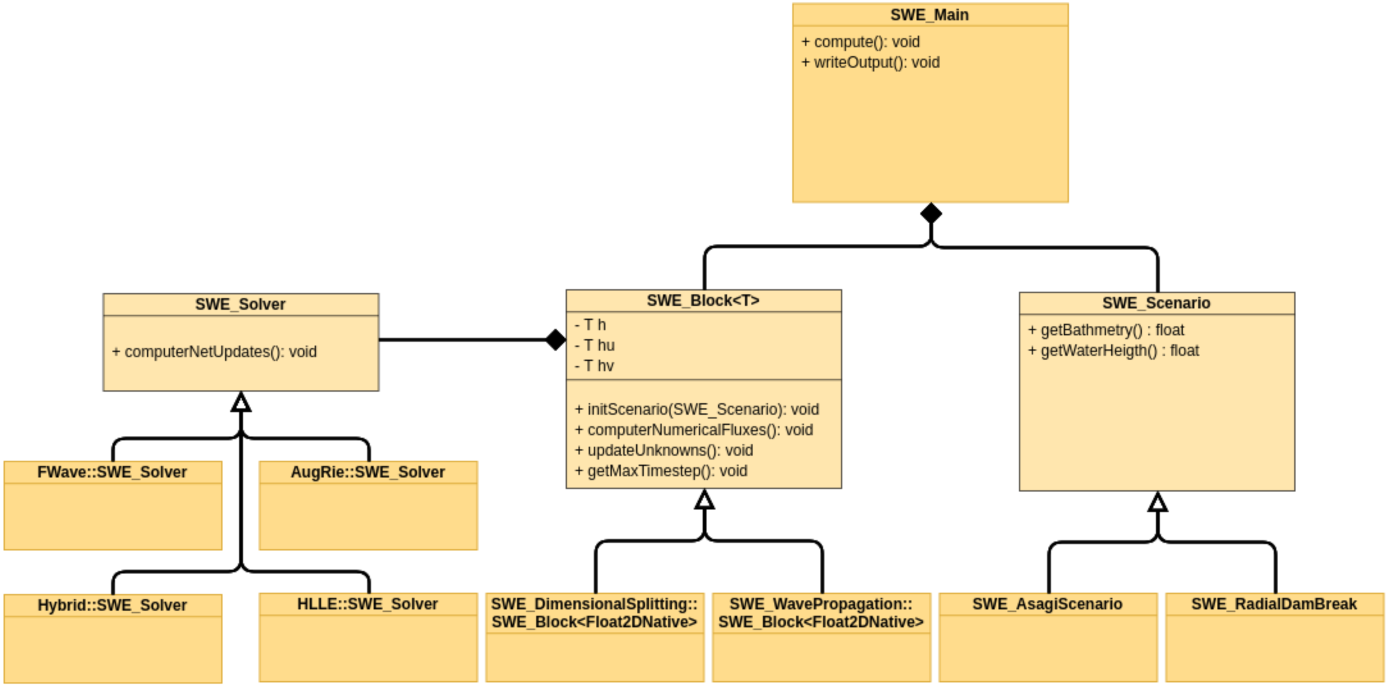


Figure 4.4: Pseudo-UML diagram for the SWE-Framework.

4.4.1 Scenarios

The *SWE_Scenario* is an abstract class, which specifies properties of the simulation, such as width and height of the domain. Additionally, it is possible to set initial conditions, such as water height and momentum, and bathymetry. Negative bathymetry represents the cell depth, while positive bathymetry values are used to model land patches. As shown in figure 4.4, concrete implementations of the abstraction are provided, e.g. *SWE_RadialDambreak* and *SWE_AsagiScenario*. For example, *SWE_RadialDambreak* is used to specify an emerging wave from the center of the simulation domain.
SWE_AsagiScenario is used to input geological data. It is based on the ASAGI[19] library and takes a bathymetry and displacement file as input and is commonly used to simulate real-life Tsunami scenarios, such as Tohoku and Chile.

4.4.2 Block

The *SWE_Block* is a templated abstract class, which provides an interface to compute the Shallow Water Equations. As described earlier, the simulation domain is defined by a Cartesian grid. Each cell state, i.e. water height and momentum, is stored in a respective two-dimensional float object, which is respresented in most implementations by the *Float2D* helper structure. The block uses a provided solver object to compute the numerical fluxes for each step of the simulations. The framework provides two main implementations of the *SWE_Block* - *SWE_DimensionalSplittingBlock* and *SWE_WavePropagationBlock*. Those implementations differ in the calculation-order of the Riemann-Problems as described in section 4.3.

4.4.3 Solver

The SWE-Framework collects a number of solvers. Those are used to compute occuring one-dimensional Riemann-Problems[23] between two cells. The solvers then return net-updates for the states of both cells as well as the maximum computed wave speed. Solvers differ in properties regarding computational power, accuracy and editable features. Most common are the *F-Wave Solver*, *Augmented Riemann Solver* and *Hybrid Solver*. The *Hybrid Solver* combines the *F-Wave Solver* and the *Augmented Riemann Solver* and switches depending on the internal state of the cells. Furthermore, the

F-Wave Solver requires less computational cost than the *Augmented Riemann Solver*, however is less accurate and cannot handle inundation. In this thesis, the *HLLEFun Solver*[2], which is a vectorized implementation of the *Augmented Riemann Solver* is used for most benchmarks. For benchmarks with load imbalances the *Hybrid Solver* is of good use, as it uses different computational methods depending on wet and dry cells.

4.4.4 Main

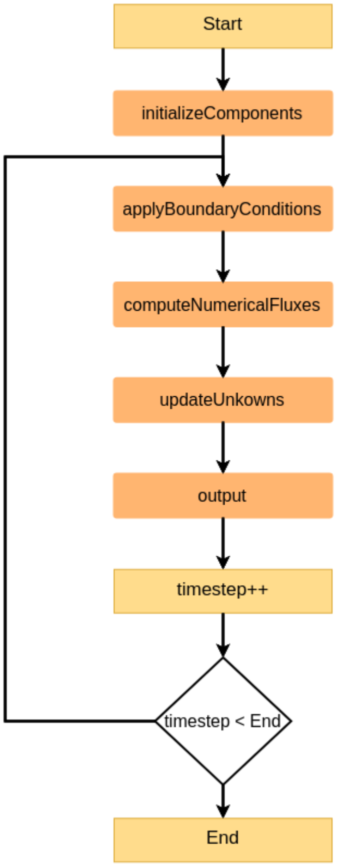


Figure 4.5: Flowchart of a sequential execution in the SWE-Framework.

As shown in figure 4.5, the main is responsible to link all components together and invoke computation of the Shallow Water model. It accomplishes its task by initializing the respective scenario and block, controlling IO-operations, such as writing output and checking command-line arguments as well as invoking each step of computation on the *SWE_Block*.

5. Implementation

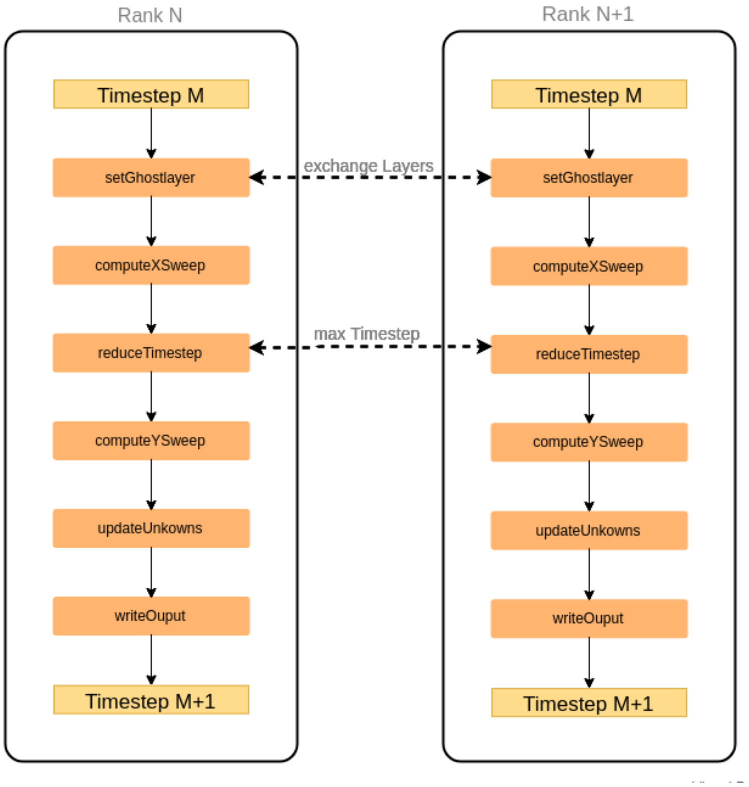


Figure 5.1: Flowchart of a concurrent execution in the SWE-Framework.

For the parallel implementation the existing sequential solution of the Shallow Water model was used as a base. This foundation was operated and extended. The concurrency concept is described with figure 5.1. The domain is divided into equally sized blocks, which are managed by a respective rank. As shown, each rank runs concurrently and synchronizes with neighbouring *SWE_Blocks* on two events. Firstly, blocks update their respective boundary condition at begin of every timestep. If it is neighbouring another block an exchange of the Ghostlayer is invoked - each parallel framework holds a respective implementation of this exchange. Secondly, after computing the horizontal sweep a reduction operation of the maximum possible timestep needs to be performed over all ranks. For each parallel framework a respective *SWE_Block* and a main function are implemented, which solve the addressed difficulties in their native approach.

5.1 OpenMP

As explained, OpenMP is not used for distributed parallelization but to parallelize within a shared memory domain. Hence, this framework is used to optimize execution within one rank. This is achieved by parallelizing the computation of the net-updates in horizontal and vertical direction. The code in listing 5.1 shows the OpenMP instructions used to parallelize the *for-loop*, which computes the X-Sweep, i.e. the net-updates in horizontal direction. Note that the instruction also reduces the maximum horizontal wave speed from all rows. OpenMP parallelization is available for every parallel *SWE_Block* except for HPX's implementation.

```
#pragma omp for reduction(max : maxWaveSpeed) collapse(2)
for (int x = 0; x < nx + 1; x++) {
        for (int y = 0; y < ny + 2; y++) {
                solver.computeNetUpdates (
                        h[x][y], h[x + 1][y],
                        hu[x][y], hu[x + 1][y],
                        b[x][y], b[x + 1][y],
                        hNetUpdatesLeft[x][y],
                        hNetUpdatesRight[x + 1][y],
                        huNetUpdatesLeft[x][y],
                        huNetUpdatesRight[x + 1][y],
                        maxWaveSpeed
                );
        }
}
```

Listing 5.1: Parallelized horizontal sweep with OpenMP.

5.2 MPI

Every MPI block holds a list of the neighbouring blocks, more precisely neighbouring ranks, which are set in the beginning of the simulation. This information is necessary to exchange the Ghostlayer cells. The transfer is implemented by using coupling *send* and *receive* calls as shown in example 5.2. To distinguish each call, a respective tag is used depending on the border and information transmitted, i.e. h, hu, hv. The *receive* operation blocks until a matching message arrives. In this implementation the *receive* requests for every border are bundled and waited for collectively. The timestep reduction is implemented by the provided *MPI_Allreduce* method, which reduces an input value over all ranks using a specified operation. In this case, the minimum function *MPI_MIN*.

```
//Sending the Left Ghostlayer
int startIndex = ny + 2 + 1;
MPI_Isend(
        h.getRawPointer() + startIndex, ny, MPI_FLOAT,
        neighbourRankId[BND_LEFT],
        MPI_TAG_OUT_H_LEFT, MPI_COMM_WORLD, &req
        );
MPI_Request_free(&req);
```

```
//receiving the Left Ghostlayer
int startIndex = 1;
MPI_Irecv(
        h.getRawPointer() + startIndex, ny, MPI_FLOAT,
        neighbourRankId[BND_LEFT],
        MPI_TAG_OUT_H_RIGHT, MPI_COMM_WORLD, &recvReqs[0]
        );
```

Listing 5.2: Example of a Ghostlayer exchange in MPI.

5.3 UPC++

The UPC++ approach makes use of its partitioned global address space by implementing *Float2DUpcxx*, which is a derived version for the shared segment of the *Float2D* helper structure. This structure is used to store and exchange the cell states of each block across ranks. Each block holds the global references of its neighbouring blocks cell states. The exchange is then performed by copying the cell states of each border into the respective global segment. This is solved by using UPC++'s *rget* for the left and right border, and the strided version *rget_strided* for the top and bottom border.This is significantly different to the exchange semantics used in the MPI implementation, as *rget* is a one-sided call. As shown in example 5.3, *rget* uses the reference of the respective neighbour. For each exchange operation one *upcxx::future* is returned. The futures are then synchronized for completion with *upcxx::when_all*. It shall be noted, that this implementation requires a global barrier before invoking the exchange, as there is no guarantee in a one-sided call that neighbouring blocks have finished computation and are ready for the Ghostlayer exchange. The timestep reduction is achieved by UPC++'s counterpart of *MPI_Allreduce*.

```
//retrieving global reference of the left neighbour
BlockConnectInterface<upcxx::global_ptr<float>> iface =
                neighbourCopyLayer[BND_LEFT];

upcxx::global_ptr<float> srcBaseH =
                iface.pointerH + iface.startIndex;

upcxx::global_ptr<float> srcBaseHu =
                iface.pointerHu + iface.startIndex;

upcxx::global_ptr<float> srcBaseHv =
                iface.pointerHv + iface.startIndex;

//invoking one-sided get operation
auto leftFutH = upcxx::rget(srcBaseH, &h[0][1], ny);
auto leftFutHu = upcxx::rget(srcBaseHu, &hu[0][1], ny);
auto leftFutHv = upcxx::rget(srcBaseHv, &hv[0][1], ny);
leftFuture = upcxx::when_all(leftFutH, leftFutHu, leftFutHv);
```

Listing 5.3: Example of a Ghostlayer exchange in UPC++.

5.4 Charm++

In Charm++'s implementation each *SWE_Block* represents a transmittable Chare-object. Additionally, the main class is also representing a Chare, which holds an array of all computation blocks. Each Chare is provided with an interface file, in which the entry functions and message-driven events can be specified. For the *Charm_Blocks* this means to hold a computations function, a respective event for each Ghostlayer and a reduction method for the maximum possible timestep. The computation method fulfils the same purpose as the main function of a rank in UPC++ or MPI. The difference is that Ghostlayer exchange happens in an event-based matter. As shown in listing

5.4, a Chare first sends a message to its neighbours containing its respective Ghostlayer, and then waits for incoming messages of the neighbouring blocks. This section is completed upon receiving all necessary borders (some Chares are facing the end of the domain and therefore do not require a message on this side). The completion invokes the next section, which is the computation of the horizontal sweep. Upon computing the horizontal sweep, the reduction operation is triggered. As shown in listing 5.5, this operation is implemented by specifying a call-back function which is invoked when the reduction operation finishes. The actual collective operation is invoked by calling *contribute(...)*. It shall be noted, that the call-back-function needs to be defined in the respective interface file.

```
overlap {
        when receiveGhostLeft(copyLayer *msg)
        if (!msg->isDummy) {
        serial { processCopyLayer(msg); }
        }
        //Repeat for other borders
        //Leave overlap when all borders received

}
serial {
        //The xSweep will trigger the reduction
        //and accumulate compute time
        xSweep();
}
```

Listing 5.4: Example of a Ghostlayer exchange in Charm++.

```
//Call-back function for the finished reduction
CkCallback cb(CkReductionTarget(SWE_DimensionalSplittingCharm,
                                reduceWaveSpeed), thisProxy);
//Contributing to the collective operation
contribute(sizeof(float), &maxTimestep,
                CkReduction::min_float, cb);
```

```
//Callback function for reduction
void SWE_DimensionalSplittingCharm::
        reduceWaveSpeed(float globalTimestep) {
        //set the globalTimestep locally
        maxTimestep = globalTimestep;
}
```

Listing 5.5: Example of a reduction operation in Charm++.

5.5 HPX

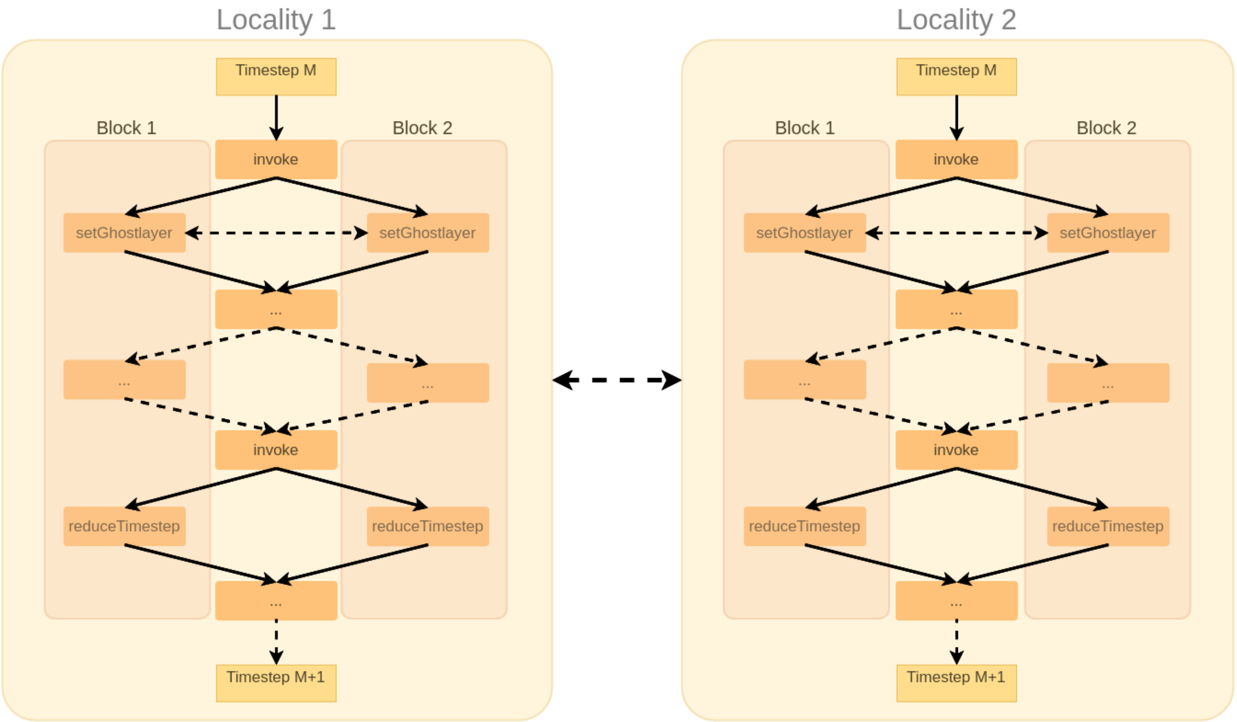


Figure 5.2: Flowchart of the HPX implementation.

The approach utilizing HPX is comparatively different. While it still uses the concept of blocks, it bundles them on each locality where they are handled collectively. To solve this task, a component class was implemented (in the first version it was implemented using HPX components but was changed to a normal class due to bad performance) which holds a handle to each simulation block. This class invokes a computation for every block running on the respective locality. Each action is invoked in a HPX asynchronous matter and is distributed by its runtime system on the available processing units of the given node. As shown in figure 5.2, the component class synchronizes completion of each task and then invokes the next step. The exchange of the Ghostlayer is divided into two cases. Transmission between blocks sharing a locality is done by explicitly copying the respective border. Blocks, which are neighbouring different localities, are using the HPX's channel component. It is similar to MPI's two-sided point-to-point communication with the distinction that the sender can choose to ”fire and forget” the sending call. The exchange between blocks is always handled by a communicator class which provides the functionality described above. Additionally, the communicator makes use of the *hpx::dataflow* semantic. As figure 5.6 displays, it automatically invokes the copy of the Ghostlayer when all borders are received. The reduction operation is designed within two consequential steps. First, the timestep is reduced within each locality by the managing class, and then each locality sends its timestep by using the channel component to locality 0. Locality 0 reduces the timestep and distributes the result. While this implementation does not use the native reduction operation of HPX, it solves the reduction task without the need of a global barrier, as the channel implementation automatically synchronizes this task. The channel communication is automatically wired by a helping communication structure as shown in listing 5.7.

```
hpx::dataflow(
hpx::util::unwrapping([] (T border,Boundary n,int nx, int ny,
                Float2DNative * h,Float2DNative * hu,
                Float2DNative * hv,Float2DNative * b,
                bool bat) -> void{
  if (n == BND_LEFT) {
        if(!bat){
                for(int i= 0; i < border.size; i++){
                        (*h)[0][i + 1] = border.H[i];
                        (*hu)[0][i + 1] = border.Hu[i];
                        (*hv)[0][i + 1] = border.Hv[i];
                }
        }else {
                for(int i= 0; i < border.size; i++){
                        (*b)[0][i + 1] = border.B[i];
                }
        }
  }
//Repeat for other borders
}),recv[n].get(hpx::launch::async),n,nx,ny,h,hu,hv,b,bat);
```

Listing 5.6: Example of using flow-control for Ghostlayer exchange in HPX.

```cpp
struct LocalityChannel
{
   std::vector<hpx::lcos::channel<T>> send,recv;
LocalityChannel(std::size_t rank, int localityRanks)
{
  static const char* master_name = "master";
  static const char* slave_name = "slave";
  if(rank == 0){
    for(int i = 1; i < localityRanks; i++){
      recv.push_back(
        hpx::find_from_basename<ch_type>(slave_name, i));
      send.push_back(ch_type(hpx::find_here()));
      hpx::register_with_basename(master_name, send[i-1], i);
    }
  }else {
    recv.push_back(
      hpx::find_from_basename<ch_type>(master_name, rank));
    send.push_back(ch_type(hpx::find_here()));
    hpx::register_with_basename(slave_name, send[0], rank);
  }
}
void set(T&& t)
{
  for(auto &channel : send){
    channel.set(hpx::launch::apply, std::move(t));
  }
}
std::vector<hpx::future<T>> get()
{
  std::vector<hpx::future<T>> ret;
  ret.reserve(recv.size());
  for(auto &channel : recv){
    ret.push_back(channel.get(hpx::launch::async));
  }
  return ret;
}};
```

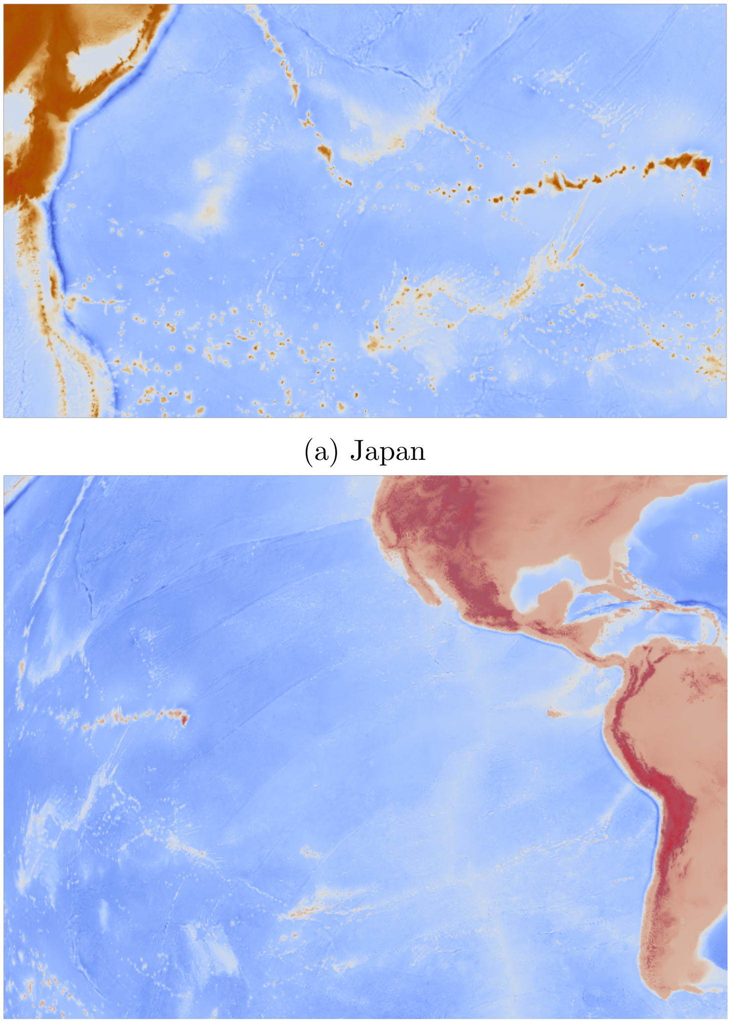
Listing 5.7: Example of the communicator class for timestep reduction.

6. Findings

6.1 Benchmark Environment

The benchmarks were performed on the Linux-Cluster ”CoolMUC-2” provided by the Leibniz Rechenzentrum. It is a Massive Parallel Processor architecture (MPP) consisting of 384 Intel Xeon E5-2697 v3 ”Haswell” nodes connected by FDR14 Infiniband. Each node gathers a total of 28 processing cores and 64GB RAM. However, the maximal node count per job is limited to 60 nodes or a total of 1680 processing units. Furthermore, all parallel frameworks were compiled with GCC 8 and all besides HPX with Intel 19.0. Benchmarks comparing all implementations make use of the GCC compiled versions. UPC++ is the only framework which supports a native Infiniband backend. Although HPX describes the possibility of using an experimental Infiniband parcel-porter we were not able to compile this feature. Therefore, the implementation using MPI, Charm++ and HPX were compiled with MPI backend. A detailed description of the exact versions and GitHub-revisions can be found in the appendix.

6.2 Input Data and Measurands



(a) Japan

(b) Chile

Figure 6.1: Bathymetric profiles of the simulation domain. Source: [7]

For all simulations we used real geographic data of the 2011 earthquake Tohoku, Japan. Furthermore, the *HLLEFun Solver* was used for all benchmarks, besides load-balancing. The benchmarks evaluate the performance by measuring different terms regarding computational speed, time and communication. In the following, *wall time* refers to the computation time excluding

IO-operations and initializations. However, it includes communication and synchronization of the blocks, such as global barriers, reduction and Ghostlayer exchange. The *GFLOPS per second* refer to the billion floating point operations per second, which were measured by the total amount of floating operations performed of all solvers divided by the wall time. The *reduction time* refers to the total time used by all blocks to reduce the maximal possible timestep, while the *communication time* refers the duration of the Ghostlayer exchange. The *barrier time* refers to the total time duration of global barriers and is only available for the UPC++ and MPI implementations. These measurands do not take time overlap of the ranks into consideration.

6.3 Results

6.3.1 Strong Scaling

The strong scaling benchmark shall evaluate scalability of the parallel frameworks by increasing the amount of processing elements on a fixed problem size. Thus, we used the Tohoku scenario with a fixed grid size where for each processing unit one rank was spawned. We chose a grid of 3500×2000 for up to 8 nodes, and 7000×7000 for up to 32 nodes As shown in figure 6.2, MPI provides best scalability in terms of wall time and computational power. Although UPC++'s communication time is not greatly affected by increasing processing elements due to the global address space, it lacks a good reduction implementation and therefore cannot reach better results. Additionally, the barrier implementation of MPI is significantly faster than UPC++'s. As expected, Charm++'s reductions are worse in scaling. A reduction operation is not a standard tool of the message-driven model and thus the support for it is likely to be very limited. Additionally, Charm++'s Chares are not synchronized and thus slower computation blocks greatly effect the duration of a reduction operation. Communication in the HPX implementation displays a great bottleneck. Although Ghostlayer exchange within a node is implemented with explicit copy operations which produce no communication overhead, inter-node communication is much slower compared to the other frameworks as shown in figure 6.2. This might be due to HPX design policy which supports fine-grained communication. Contrary to the other frameworks, where reduction is performed over each rank, HPX's implementation requires significantly less communication for the reduction operation and therefore performs better with a low node count.

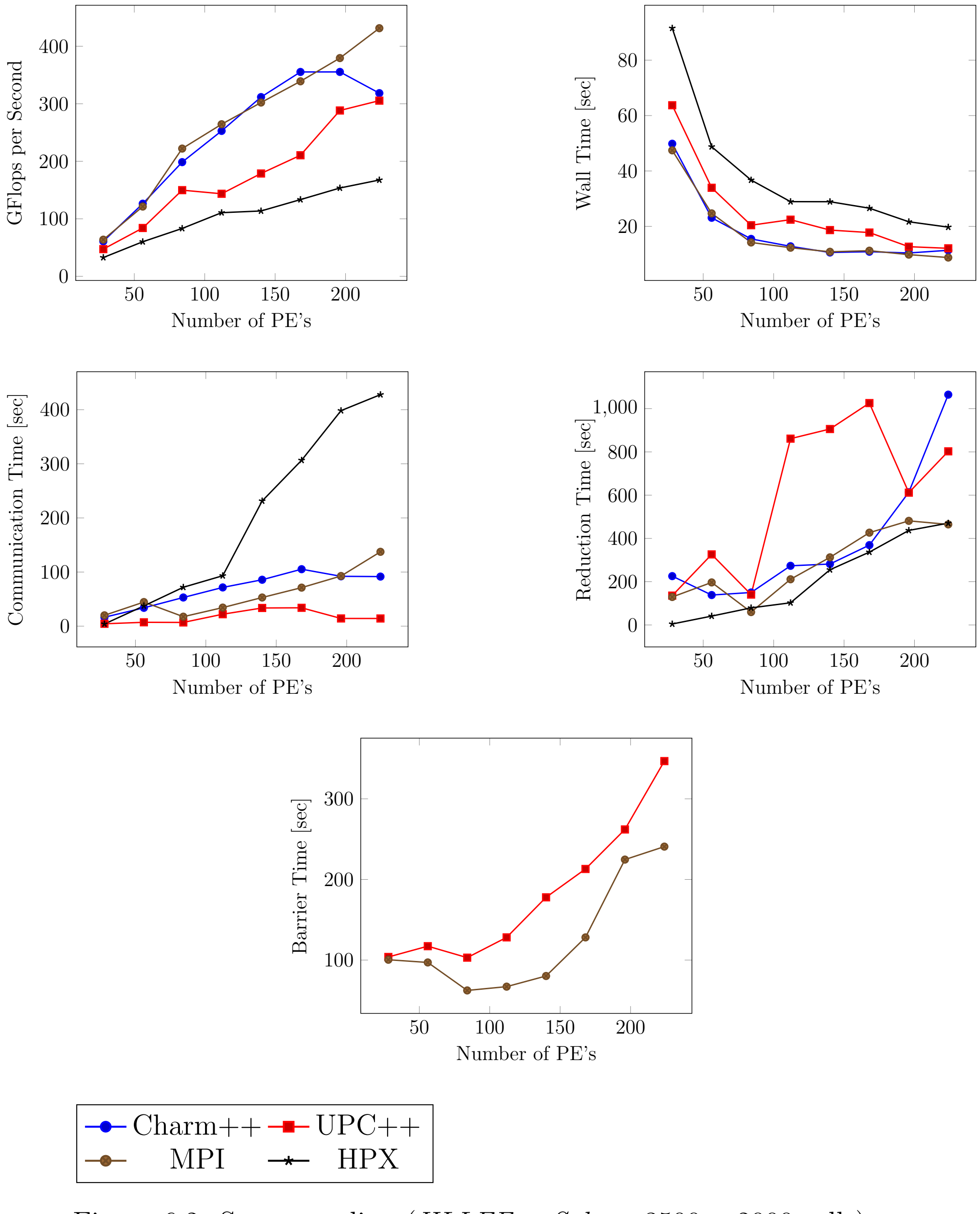


Figure 6.2: Strong scaling (*HLLEFun Solver*, 3500 × 2000 cells).

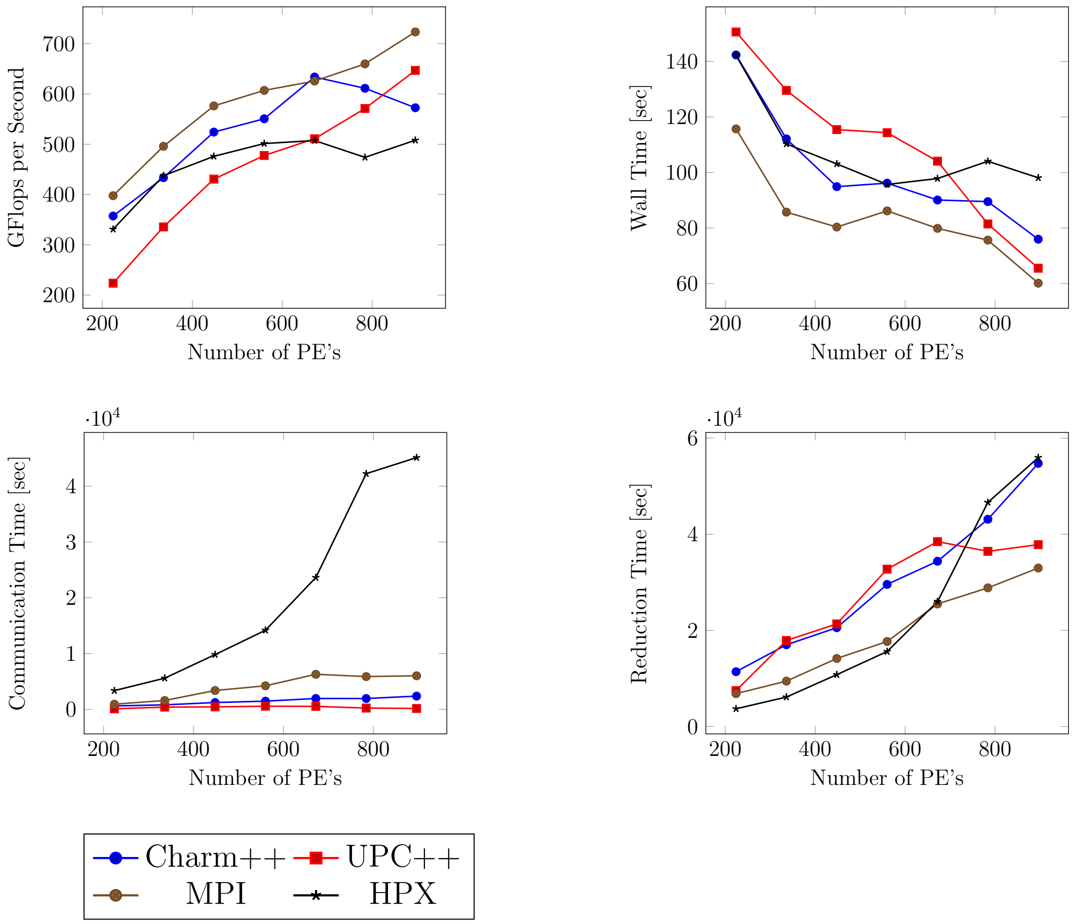


Figure 6.3: Strong scaling (*HLLEFun Solver*, 7000 × 7000 cells).

6.3.2 Weak Scaling

Weak scaling benchmarks evaluate scalability by increasing the total amount of processing elements but keeping a fixed problem size per processing unit. This was achieved by balancing the grid size with the respective node count as shown in table 6.1.

Number of PE's	Grid size
28	2000 × 2000
56	3200 × 2500
112	4000 × 4000
224	6400 × 5000

Table 6.1: Domain resolution used in weak scaling.

As shown in figure 6.4, each framework shows similar weak scaling behaviour. The graph displaying computational power shows an overall increase in GFLOPS/s. However, the growth ratio is moderate, which leads to a growing wall time. This growth ratio can be explained with the almost exponential increase in communication and reduction time.

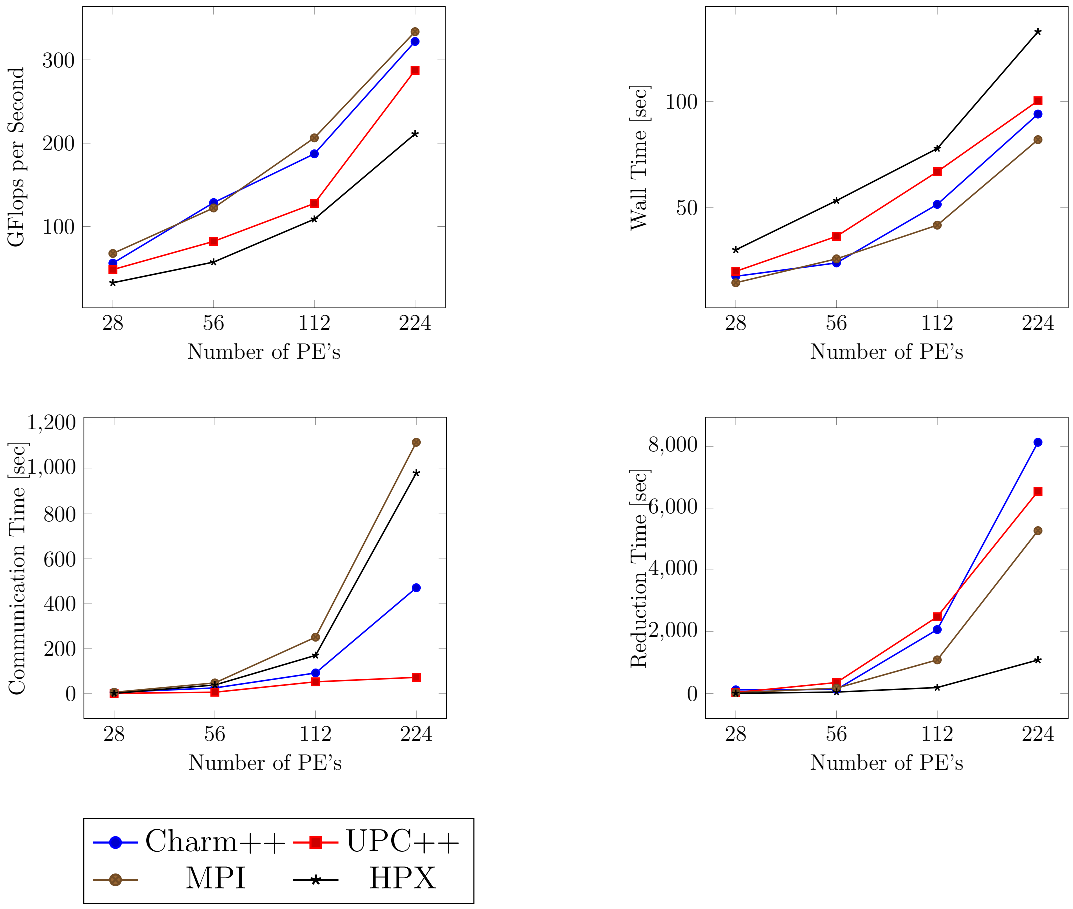


Figure 6.4: Weak scaling (*HLLEFun Solver*).

6.3.3 Load Balancing

In this benchmark, we tested the Charm++'s ability of load-balancing. Imbalances were created by using the *Hybrid Solver* in combination with the Tohoku scenario. Charm++'s implementation was executed with a over-decomposed amount of Block-Chares. The Load-Balancer of Charm++'s runtime could then distribute the workload across localities or migrate them to even out imbalances. For this benchmark the default greedy load-balancer was used. As shown in figure 6.6, over-decomposition does not improve performance. This shows, that the extra communication overhead and scheduling-effort is higher than the load imbalance bottleneck created in this benchmark.

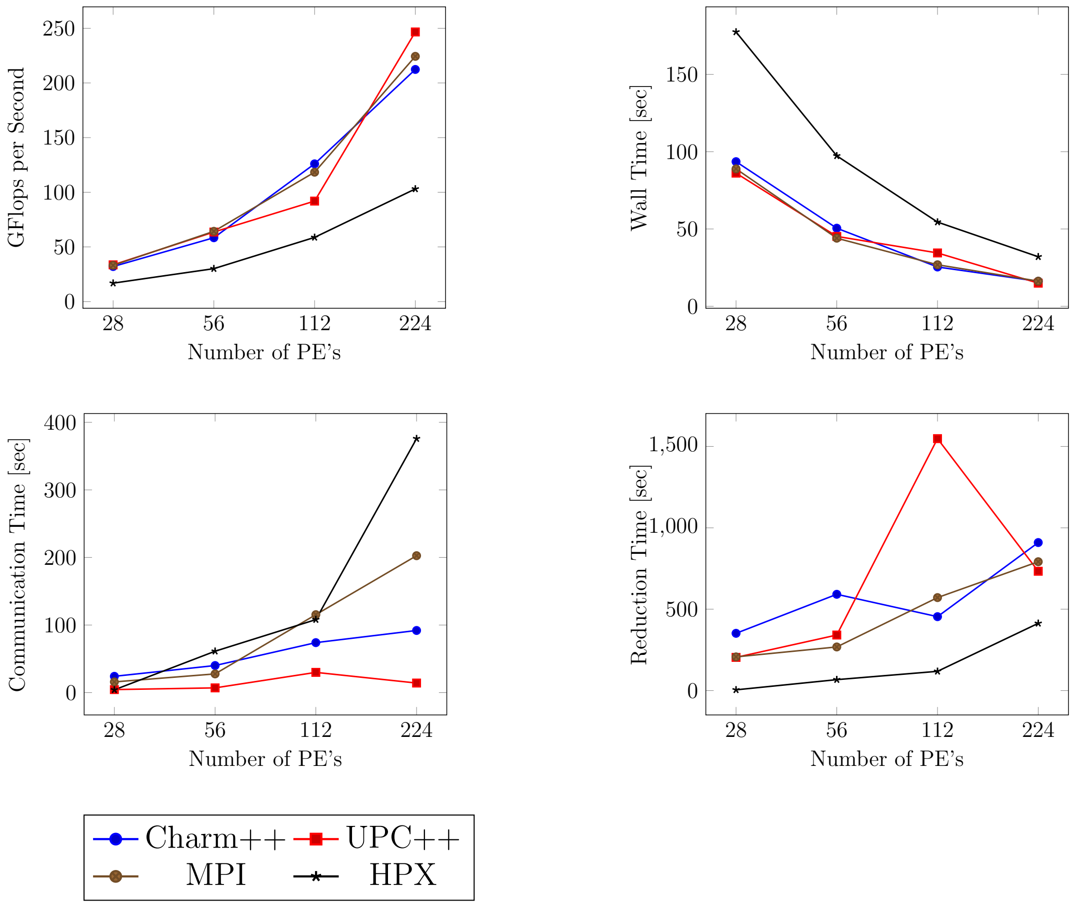


Figure 6.5: Strong scaling (*Hybrid Solver*, 3500 × 2000 cells).

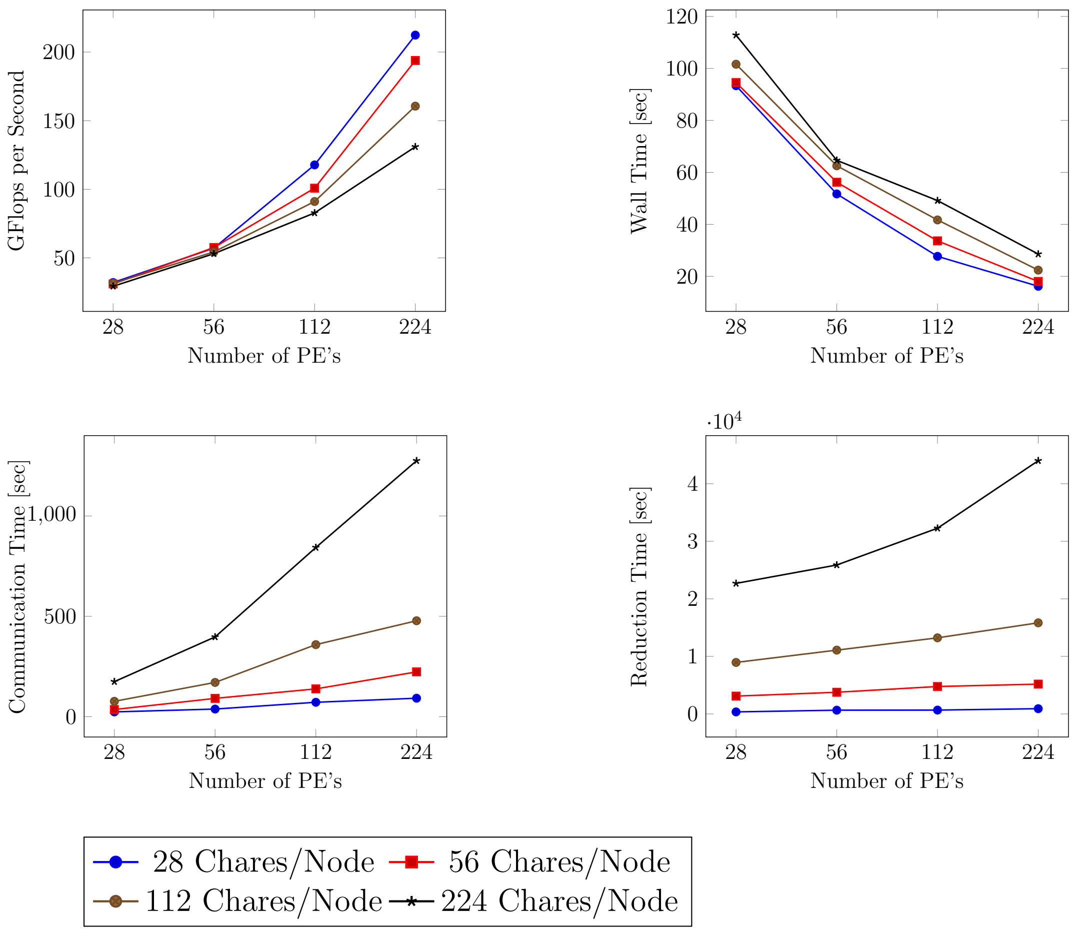


Figure 6.6: Strong scaling of Charm++ with different Chare counts (*Hybrid Solver*, 3500 × 2000 cells).

6.3.4 HPX

In this scenario, we simulated a strong scaling for HPX with different block counts per computational node. As shown, a higher number of blocks per node leads to an increase in performance on a single node. This behaviour can be explained by HPX's design being optimized for medium sized parallel tasks. Additionally, communication of blocks within a locality creates no communication overhead, as exchange is implemented with copy operations. Furthermore, simulations with increased block counts are exposed to higher total communication overhead, which increase with rising node count. This results from more communication channels created between neighbouring localities. The average best block count is 56 blocks per node. This result can be explained by the fact that the cluster supports two Hyper-threads per processing core, which is a total of 56 threads per node.

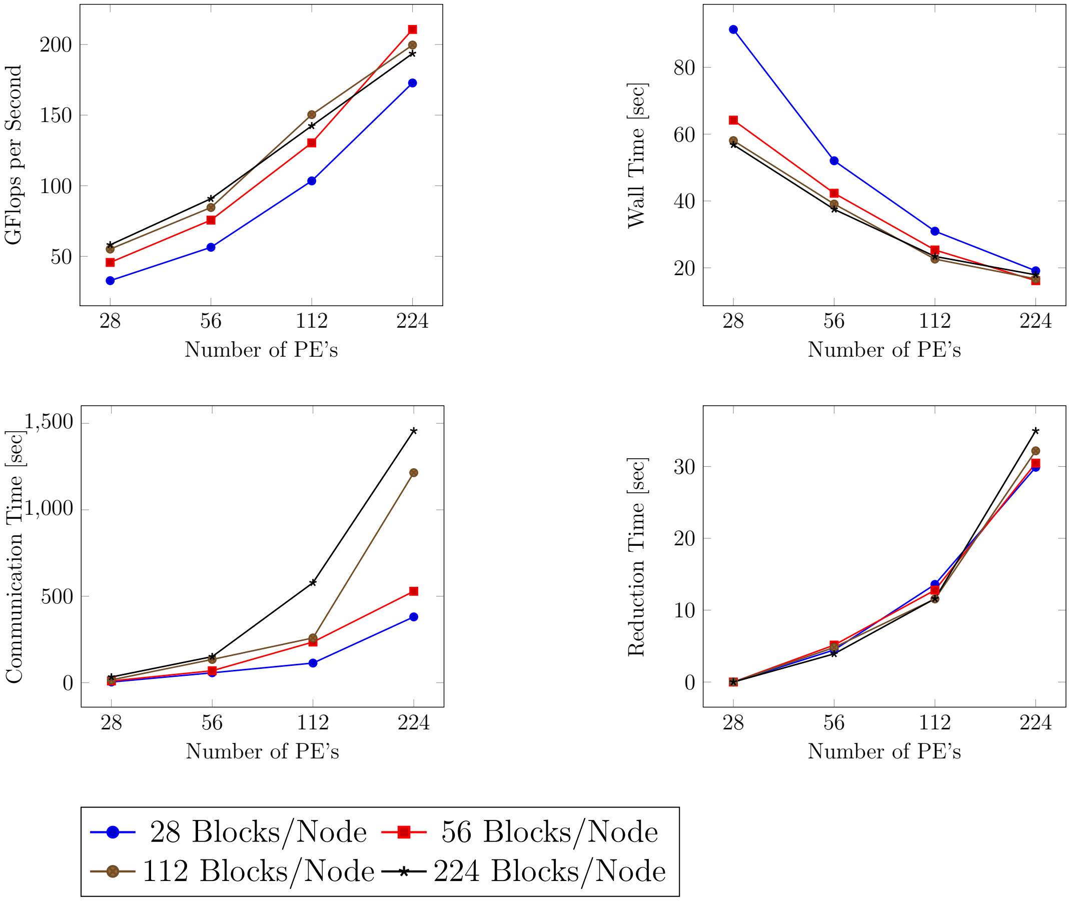


Figure 6.7: Strong scaling of HPX with different block sizes (*HLLEFun Solver*, 3500 × 2000 cells).

6.3.5 Execution Trace

In this scenario, we compare the execution trace collected by the Intel Trace Analyzer. The Intel Trace Analyzer and Collector [12] is a graphical tool for analysis and debugging of concurrent applications. It is mainly used to trace MPI calls, however, it allows to specify user-defined events in the code, which can be tracked additionally. We specified custom events for each function of the respective SWE_Block. By compiling with a tracing-flag, the application is set to produce trace-data at runtime. After execution, it can be displayed with the Intel Trace Analyzer. We used this tool to visualize the execution trace of each processor over time.

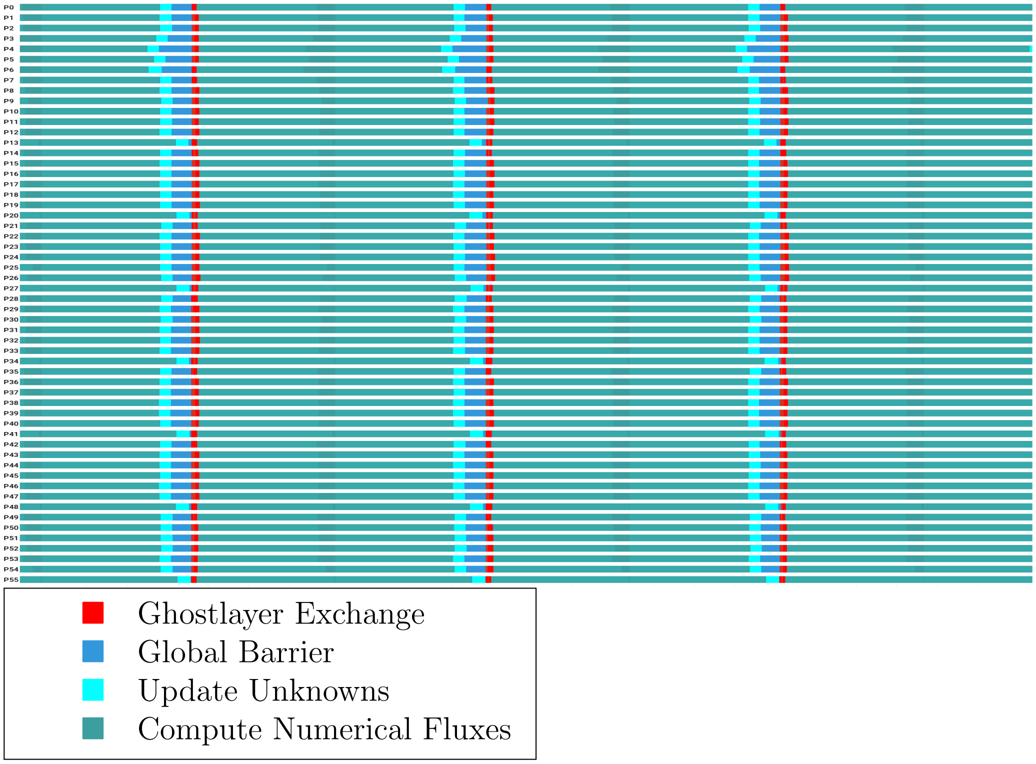


Figure 6.8: Execution trace of MPI.

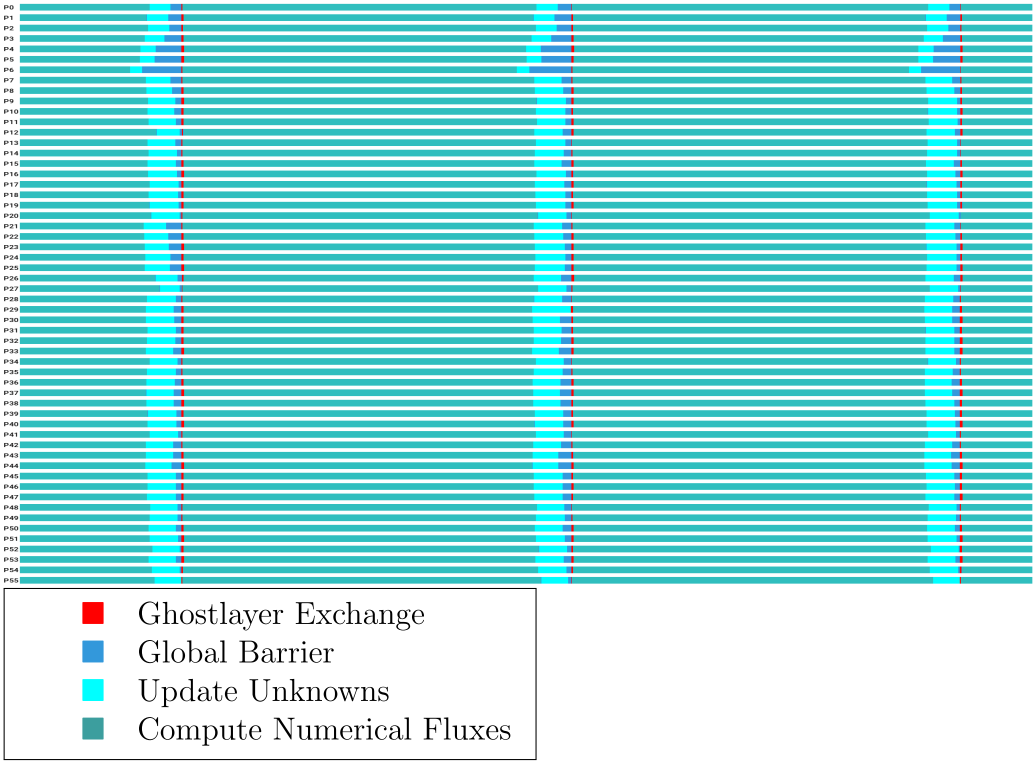


Figure 6.9: Execution trace of UPC++.

As expected, MPI and UPC++ show similar behaviour. Both implementations map exactly one SWE_Block per processing unit and make use of global barriers, so it is no surprise that events are almost perfectly aligned among processing elements. As shown in figure 6.10, HPX displays quite the opposite. Although each locality joins the asynchronous action frequently, it is up to the runtime system to distribute the tasks among processing units. This leads to a chaotic distribution, which increases with block count. Charm++'s message-driven approach shows a similar outcome. As active messages do not require synchronization, it results in unordered execution. Over-decomposition could amplify this behaviour and provide similar results to HPX's implementation.

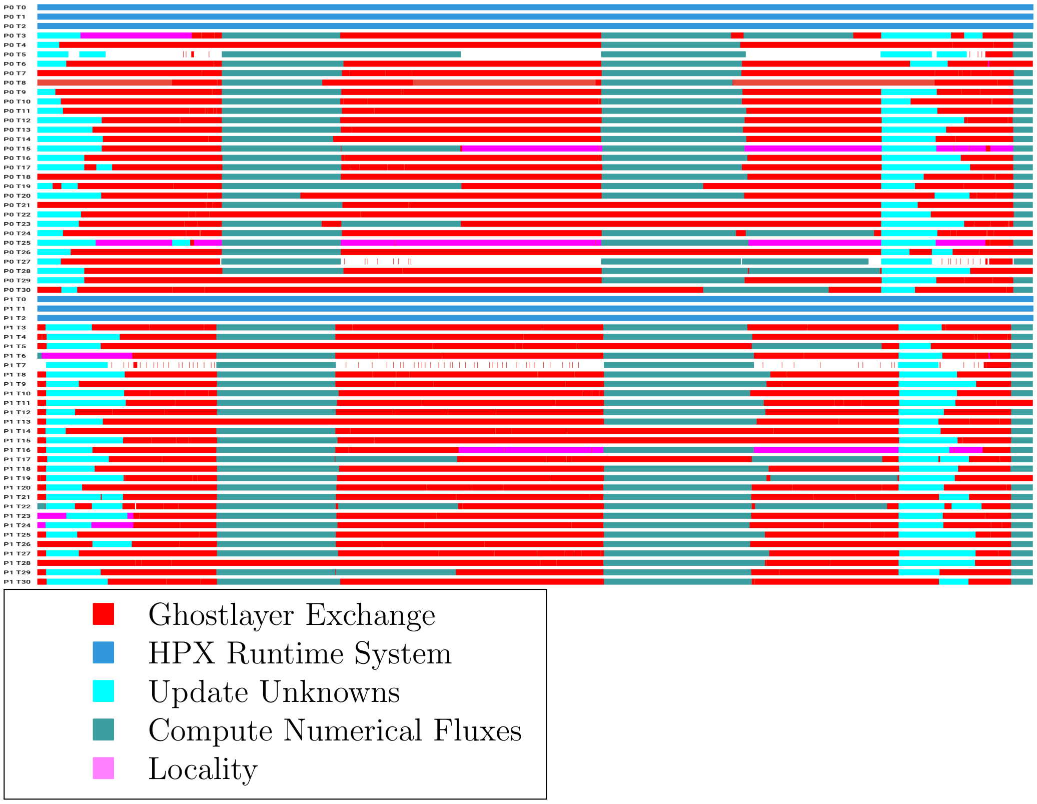


Figure 6.10: Execution trace of HPX.

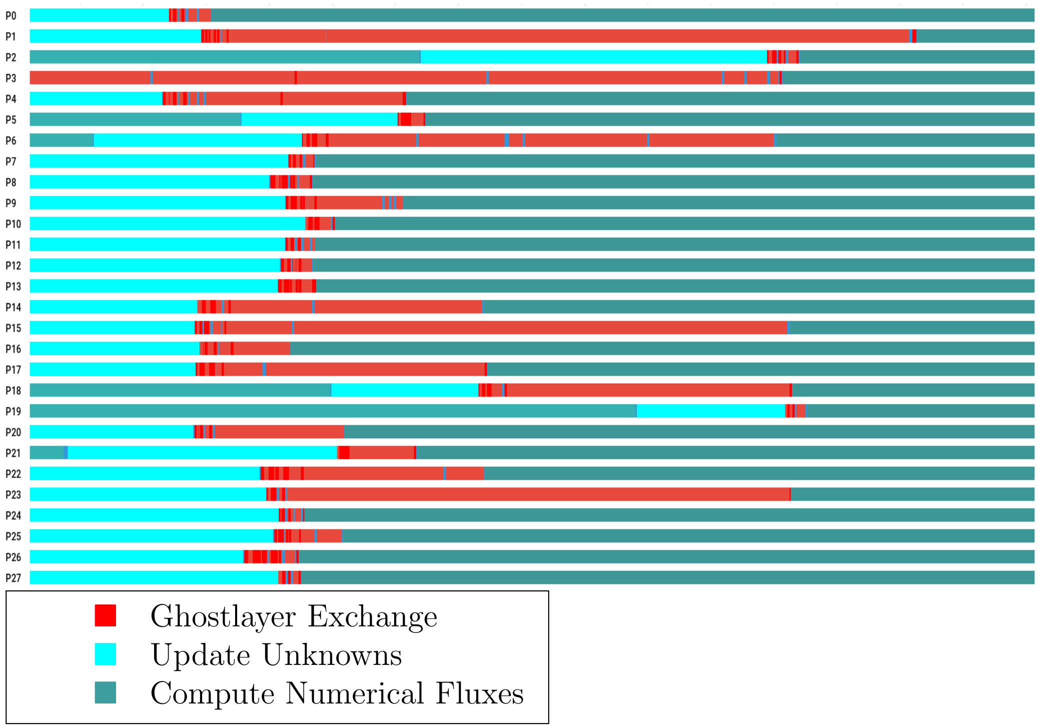


Figure 6.11: Execution trace of Charm++.

7. Conclusion and Further Work

Overall, MPI showed the best results. It can reasoned that MPI is optimized for static distributed work-loads and thus our parallelization approach fit MPI best. UPC++ provides a great documentation, which collects detailed explanations to each command and functionality. Additionally, it gathers examples describing the most important concepts of the framework. To resolve less frequent issues, one can always ask in the small, but very active google group. Furthermore, UPC++'s communication library GASNet-Ex provides native Infiniband support for the implementation. In addition to its programming model, this lead to very promising results in terms of point-to-point data-exchange. Unfortunately, those results were overshadowed by relatively slow global synchronization and reduction. For future implementations the use of synchronization could be reduced by improving the parallelization model used in our solution or by experimenting with a MPI-UPC++ [9] hybrid model which combines the best of each respective world. Charm++'s documentation makes a great effort to explain the programming and execution model as well as basic programming concepts, e.g. Chare arrays and Interface files. However, it relies more on programming examples to explain the actual functionalities. Charm++ does not provide Infiniband support and thus it was used with a MPI backend. Overall, Charm++ performed well, it showed good scaling and performed in many cases comparable to MPI. Over-decomposition of blocks did not improve performance in load imbalanced scenarios, however, it showed potential improvement when block sizes are very small and therefore migration costs of a Chares are low. HPX comes with a huge library, which provides a great amount of features. Its design requires C++11 coding standard (or higher) which is utilized throughout the entire framework. This enables a very modern programming environment, which is unique for parallelization frameworks. HPX is fairly new and still in a development state, thus the community is rather small and only a number of environments and architectures are supported. A big disappointment was that HPX does not support the newest Intel compiler, which forced us to compile with GCC where we could not utilize vectorization and achieve full optimization for the used Intel cluster architecture. Studies show very good scalability and thus we were very keen to use HPX's runtime model. However, we had trouble utilizing given features to the full potential. Our main approach was to utilize HPX components and therefore create migratable block objects. Unfortunately, this created so much communication overhead that it was not a sustainable solution to our problem. This may cohere with reported performance losses when using the MPI parcel-porter, which was a necessity in our benchmark-environment. Thus, our final solution did not use HPX components. Additionally, our parallelization model relies a lot on data-exchange. This is contrary to the ParalleX model which favours a "move work to data" policy and fine-grained parallelization, as we saw in better performance on small computation partitions, i.e. blocks. That said, a more HPX supportive parallelization model would improve performance. In the future, this could be implemented by using HPX's distributed views or utilizing distributed versions of HPX parallel algorithms.

Bibliography

[1] John Bachan et al. “UPC++: A High-Performance Communication Framework for Asynchronous Computation”. In: *Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium (to appear)*. 2019.

[2] Michael Bader et al. “Vectorization of an Augmented Riemann Solver for the Shallow Water Equations”. In: *Proceedings of the 2014 International Conference on High Performance Computing and Simulation (HPCS 2014)*. Ed. by Waleed W. Smari and Vesna Zeljkovic. mediatitle: Proceedings of the 2014 International Conference on High Performance Computing and Simulation (HPCS 2014)¡br¿editor: Smari, Waleed W.; Zeljkovic, Vesna¡br¿. IEEE, Aug. 2014, pp. 193–201.

[3] Seonmyeong Bak et al. “Multi-level load balancing with an integrated runtime approach”. In: *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press. 2018, pp. 31–40.

[4] Dan Bonachea and Paul H Hargrove. *GASNet-EX: A High-Performance, Portable Communication Library for Exascale*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2018.

[5] Alexander Breuer and Michael Bader. “Teaching parallel programming models on a shallow-water code”. In: *2012 11th International Symposium on Parallel and Distributed Computing*. IEEE. 2012, pp. 301–308.

[6] Leonardo Dagum and Ramesh Menon. “OpenMP: An industry-standard API for shared-memory programming”. In: *Computing in Science & Engineering* 1 (1998), pp. 46–55.

[7] GEBCO. *General Bathymetric Chart of the Oceans*. URL: `https://www.gebco.net/`.

[8] Patricia Grubel et al. “The performance implication of task size for applications on the hpx runtime system”. In: *2015 IEEE International Conference on Cluster Computing*. IEEE. 2015, pp. 682–689.

[9] Jahanzeb Maqbool Hashmi, Khaled Hamidouche, and Dhabaleswar K Panda. “Enabling Performance Efficient Runtime Support for Hybrid MPI+ UPC++ Programming Models”. In: *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2016, pp. 1180–1187.

[10] Thomas Heller et al. “Hpx–an open source c++ standard library for parallelism and concurrency”. In: *Proceedings of OpenSuCo* (2017), p. 5.

[11] University of Illinois. *Introduction to the Charm++ Runtime System*. URL: `http://charmplusplus.org/tutorial/CharmRuntimeSystem.html`.

[12] *Intel Trace Analyzer and Collector*. URL: `https://software.intel.com/en-us/trace-analyzer`.

[13] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. “Parallex an advanced parallel execution model for scaling-impaired applications”. In: *2009 International Conference on Parallel Processing Workshops*. IEEE. 2009, pp. 394–401.

[14] Laxmikant V Kale and Sanjeev Krishnan. “CHARM++: a portable concurrent object oriented system based on C++”. In: *OOPSLA*. Vol. 93. Citeseer. 1993, pp. 91–108.

[15] Zahra Khatami et al. “A massively parallel distributed n-body application implemented with hpx”. In: *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE. 2016, pp. 57–64.

[16] Jaime Miguel Fe Marqués. “Introduction to the Finite Volumes Method. Application to the Shallow Water Equations.” In: ().

[17] Jurek Olden. “Performance Analysis of SWE Implementations based on modern parallel Runtime Systems”. BA thesis.

[18] Karl Rupp et al. “Years of microprocessor trend data”. In: *Figure available on webpage http://www. karlrupp. net/wp-content/uploads/2015* 6 (40).

[19] T. C. of Scientific Computing. *a parallel Server for Adaptive GeoInformation.* URL: `https://github.com/TUM-I5/ASAGI`.

[20] T. C. of Scientific Computing. *The Shallow Water Equations teaching code.* URL: `https://github.com/TUM-I5/SWE`.

[21] Hongzhang Shan et al. “Experiences of applying one-sided communication to nearest-neighbor communication”. In: *2016 PGAS Applications Workshop (PAW).* IEEE. 2016, pp. 17–24.

[22] Yanhua Sun et al. “A ugni-based asynchronous message-driven runtime system for cray supercomputers with gemini interconnect”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium.* IEEE. 2012, pp. 751–762.

[23] Eleuterio F Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction.* Springer Science & Business Media, 2013.

[24] David W Walker and Jack J Dongarra. “MPI: a standard message passing interface”. In: *Supercomputer* 12 (1996), pp. 56–68.

[25] Yili Zheng et al. “UPC++: a PGAS extension for C++”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium.* IEEE. 2014, pp. 1105–1114.

A. Testing Environment

Name	Version	GitHub-Revision
Intel Compiler	19.0.4.243	-
GCC Compiler	8.2.0	-
Intel MPI	2019 Update 4	-
Charm++	v6.90	c3d50efa1ec4649b29c60fc734ba1458d3da1861
UPC++	v2019.3.2	fc6e0d4440be1ab2efe3306729048fb22fb78b1a
HPX	v1.3.0	a943fd2c5d8b90d2f45be919850eefa9c31788e8

Figure A.1: Versions Overview