Performance Analysis of SWE Implementations based on modern parallel Runtime Systems

Jurek Olden
Bachelor’s Thesis in Informatics

Performance Analysis of SWE Implementations based on modern parallel Runtime Systems

Performanzanalyse eines Lösers für Flachwassergleichungen basierend auf modernen parallelen Laufzeitsystemen

Author: Jurek Olden
Supervisor: Prof. Dr. Michael Bader
Submission Date: 15.10.2018
I confirm that this bachelor’s thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.10.2018

Jurek Olden
Abstract

As high-performance computing is starting to reach exascale and the amount of interconnected nodes of supercomputers is ever increasing, parallel applications must be extremely scalable, a property which is limited by the degree of parallelization the used algorithms provides, but which can also suffer greatly from load imbalances and processor downtimes due to the delay of data movement. For parallel runtime systems, it is highly desireable to promote the overlap of communication and computation, to evenly distribute load and to aid programmers in designing applications which exploit the given hardware as best as possible. This work presents the different notions and paradigms used by the parallel runtime systems UPC++, Charm++ and AMPI in order to achieve these goals. An overview of previous benchmarks is given, which already show significant performance increases in some applications by leveraging the features of the respective parallelization system. The Shallow Water Equations model will be introduced and the discretization, domain decomposition and parallelization strategy for implementing it will be elaborated on. The actual implementations using each of the aforementioned frameworks are described, which differ only in the way data is being communicated. Finally, the performance benchmarks of each implementation are illustrated, showing that all runtime systems are able to consistently perform on an equivalent level on par with the MPI reference implementation.
# Contents

Abstract iii

1 Introduction 1

2 Parallelization Frameworks 3
   2.1 OpenMP .................................................. 3
   2.2 MPI .......................................................... 3
   2.3 UPC++ ..................................................... 4
   2.4 Charm++ ................................................... 5
   2.5 AMPI ....................................................... 6

3 Shallow Water Equations 7
   3.1 Discretization and Domain Decomposition ..................... 7
   3.2 Update Scheme ............................................ 8
   3.3 Solver .................................................... 9

4 Implementation 10
   4.1 MPI ........................................................ 12
   4.2 UPC++ .................................................... 13
   4.3 Charm++ .................................................. 14

5 Benchmarks 16
   5.1 Input Data ............................................... 16
   5.2 Method ................................................... 16
   5.3 Results .................................................. 16

6 Conclusion and further Work 22

List of Figures 24

Bibliography 25
1 Introduction

Computer performance has been exponentially increasing for over fifty years [1]. When heat generation and gate delays became limiting factors for single core speed [2, Section 1.1], processing units grew in width, with multiple cores per socket, so called shared-memory processors. Nowadays, even multicore shared memory systems are being limited by memory speed and bandwidth, with the compute power growing two times faster than memory speed [3]. Current supercomputers mostly use 8 to 64 cores per socket [4], further growth in performance is therefore achieved by interconnecting multiple shared memory systems to multi node clusters. In order to harness the performance offered by modern high-performance computers consisting of up to millions of cores, each with a relatively low clock speed of around 2–3GHz [4], software has to exhibit extreme levels of parallelization. But even if a highly parallel algorithm for some problem exists, the cost of data movement is quite high; both in terms of delaying other threads and time that is not being spent computing. This means that software has to overlap communication and computation very efficiently to be able to scale to the amount of processing units available to it.

The currently dominating systems for parallelizing applications are OpenMP in the case of shared memory parallelization and MPI for multi-node parallelization [5]. Hybrid implementations are used as well, which communicate between nodes using MPI and perform loop-level parallelization within a node using OpenMP, which can improve performance when using larger core counts [5][6][7]. But the essential concept of MPI is simple point to point communication between isolated nodes, which makes it a non-trivial task to design systems such that they are able to take maximum advantage of data already available, delaying computations which needs data yet to be received and to properly manage load imbalances. Runtime systems such as UPC++ and Charm++ (as well as AMPI) use approaches differing from simple message passing between nodes in an effort to make it easier to design software with these desirable properties, as well as trying to offer a flat performance gain in certain situations. In this work, the terms ‘runtime system’ and ‘(parallelization) framework’ will be used interchangeably.

According to the designers, UPC++ is “well-suited for implementing elaborate distributed data structures where communication is irregular or fine-grained” [8]; it also displayed performance on par with equivalent MPI, Titanium or UPC implementations.
1 Introduction

while providing “better portability, interoperability and maintainability” [9].

Charm++ has already been devised in 1993, the goals at the time were to respond to issues of portability and code reuse, the complexity of parallel programming, the latency of accessing shared data and the irregularity of problems such as sparse matrix computations [10]. By now, Charm++ offers “several extremely sophisticated features, such as application-independent object migration, fault tolerance, power awareness, and automatic overlap of communication with computation” while claiming it “might dramatically improve your performance” when having scalability problems due to “dynamic behavior, load imbalance, or communication costs” [11].

The goal of this work is to present the different notions and paradigms used by the parallel runtime systems MPI, UPC++, Charm++ and AMPI, including an overview of previous benchmarks and analyses as well as other work on the topic. The Shallow Water Equations model will be introduced and the discretization, domain decomposition and parallelization strategy for implementing it will be elaborated on. Then the actual implementations using each of the aforementioned frameworks are described, which differ only in the way data is being communicated. Finally, the performance of each implementation will be tested. Because the nature of the problem leaves little room for complex designs, these benchmarks seek to offer an insight into the baseline performance of these frameworks and their core features. This will include the interoperability with loop-level parallelism using OpenMP, scaling properties and an analysis of the results.
2 Parallelization Frameworks

2.1 OpenMP

The OpenMP API consists of compiler directive, which enables the programmer to declare sections of the program to be run in parallel [12]. In these parallel sections a master thread forks off slave threads in a fork-join parallelizing model for data or work sharing, such as distributing iterations of a loop to different threads (Figure 2.1, Page 3)[13, Chapter 2]. Because OpenMP directly exploits the memory sharing of multi-core processing units and does not produce overhead by unnecessarily passing messages, it is often assumed to perform better than MPI on a single node [12]. The benchmarks (Figure 5.2, Page 17) do show that OpenMP performs at least as well as implementations based on explicit communication.

2.2 MPI

MPI, or the Message Passing Interface, defines a standard interface for message passing and parallelization functions. MPI design is an ongoing process since the early 1990s, to provide a common working ground for the then emerging high-performance computing. There are a variety of implementations available, for instance OpenMPI and MPICH, with the latter being in use. These implementations provide point-to-point communication between compute nodes over an interconnect, with an abundance of supported hardware ranging from standard network interfaces to proprietary solutions such as InfiniBand. Communication between nodes always has to be explicitly expressed, e.g. as a send operation to a fixed destination, with a counterpart receive operation at the destination. The more recent MPI standards also support more advanced features such

```c
1 int arr[size];
2 #pragma omp parallel for
3 for(int i = 0; i < size; ++i)
4     arr[i] = i;
```

Figure 2.1: Simple OpenMP loop-level parallelism example. Source: [13]
as one sided communication and high performance shared memory performance [14].
While MPI provides many features surrounding multi-process parallelization such
as locking, synchronizing, broadcasting, reductions etc., it does not extend beyond
passing messages between nodes [2, Section 2.2]. MPI is the mainstream solution for
multi-node computing, has a large community, active development and a broad footing
in the industry with many commercial partners [15]. For this work, MPI serves as
the reference implementation providing baseline performance measurements, against
which other implementations will be compared.

2.3 UPC++

UPC++ is a “C++11 library that supports APGAS programming” [8]. APGAS stands
for Asynchronous Partitioned Global Address Space, an abstraction which unifies the
isolated memory of each individual node into a single virtual global address space (Fig-
ure 2.2, Page 5). The partitioning refers to the locality of global address segments to
certain nodes, with each node also having access to their private local memory segments.
The global memory is accessed via global pointers which support pointer arithmetic but
may not be dereferenced; the shared data must be allocated inside a shared segment
by using UPC++ functions, UPC++ processes may check if the data at some pointer is
stored in their local shared segment and subsequently access it directly using a C-style
pointer. Furthermore, UPC++ never implicitly blocks, instead using the concept of
‘futures’, which represent a result that will be available at some later point. Futures may
be queried for completion and be waited on, multiple futures can also be conjoined.
These abstractions greatly simplify concepts such as one-sided communication as well
as distributed data structures and enable programs to be designed as if they were
operating on shared memory. The needed communication can then be dynamically
scheduled in the background and computations be chained as data becomes avail-
able. See [16] and [17] for details. UPC++ communications “run at close to hardware
speeds” [8], this is achieved by “utilizing the low-overhead GASNet-EX communication
library” [8]. This library was designed since global address space frameworks such
as UPC++ depend on low latency communications even more than frameworks based
on message passing and most high-performance communication solutions are tied to
specific hardware. GASNet is therefore highly portable and suited to be used by global
address space frameworks, with its performance overhead minimal as shown in initial
benchmarks [18]. GAS-style frameworks have already been shown to outperform MPI,
though it has to be noted that these results where achieved when shared data structures
and fine-grained overlapping of communication and computation, i.e. problems GAS-
style frameworks were explicitly designed for [8][19][20]. The findings in [20] consist of
benchmarks of elementary operations and not a production-level implementation of an actual problem. The results of [19] were obtained from microbenchmarks and global Fast Fourier Transformations, the author stressed again the importance of overlapping communication with computation.

2.4 Charm++

Charm++ abstracts parallelity in a radically different way than MPI or UPC++ by leveraging object orientation. Instead of explicitly managing ranks and sending data, Charm++ is ‘message-driven’. Parallel operations are achieved by calling ‘entry’ methods on parallel objects called ‘Chares’, which in practice is a function call as usual, the function itself is declared to be an entry method. Internally, this produces a message which carries data as well as the code to be executed. This way, the processor never waits on data to be received, but the underlying scheduler (called Converse) chooses the next available message for processing. The specifics of a Chare, e.g. its entry methods, are defined in a separate interface file, which acts like a second header file. Before compilation, these interface files have to be parsed by the Charm++ compiler, which generates two helper files that must be included in the Chares source code. These files contain auto-generated methods used by the runtime as well as a base class from which
the Chare must inherit.

When calling entry methods on a Chare, the necessary data is passed as a parameter, either marshalled by the Charm++ framework which requires this data to be made serializable by the ‘PUP’ framework [21, Section 6], or by using custom messages (Section 4.3, Page 14)[21, Section 10]. Since entry methods on Chares always have a void return value, program design and expressing parallel control flow differs significantly from other frameworks. Charm++ uses the so called ‘Structural Dagger (SDAG)’ to impose a control flow upon Chares, which allows loops, conditional statements and waiting on specific entry methods to trigger function calls (Section 4.3, Page 14). Charm++ is able to interoperate with sequential code, which actually makes it quite simple to integrate parallelization within an existing codebase, especially if it was already using objects. Only the objects which are supposed to be work in parallel have to be transformed and a ‘Main Chare’ containing the program entry point has to be created.

The advantage of using Chares is that they fully encapsulate the data needed for the computations happening in their member functions and are therefore perfectly suited to be migrated across processing units. This extremely promotes load balancing, which is the core benefit of Charm++. There are results showing the extraordinary scaling properties of Charm++ on tests performing LU-decomposition, Fast Fourier Transformation as well as molecular simulation code and more [22].

2.5 AMPI

Advanced MPI is a MPI implementation such as OpenMPI or MPICH, based on Charm++ and therefore exploiting its load balancing capabilities. Programs may be compiled against AMPI and potentially make performance gains without being rewritten to support Charm++. The AMPI execution model core idea is to ‘overdecompose’ the task, making the communication more fine-grained and therefore leaving more space for load balancing, potentially better cache usage and better overlap of computation and communication. This is achieved by splitting processing elements into virtual processors, which may each carry a MPI rank. These virtual processors are realized with user-level threads and may be migrated across physical processors. All of these benefits are automatically performed by the scheduler, such that any existing MPI code base may take advantage of them without implementing the needed pipelining, load balancing etc. Charm++ provides multiple ready-to-use scheduling strategies; custom schedulers can be integrated as well. There are already a number of production level applications and benchmarks written to work with AMPI, see [23]. AMPI performance gains have been illustrated in [24].
3 Shallow Water Equations

The Shallow Water Equations are the standard approach for tsunami simulation (Compare [25][26]). They are derived from classical conservation laws and represent a system of nonlinear hyperbolic partial differential equations [27]. Given in the following differential form, \( h \) denotes the water height, \( hu \) and \( hv \) the water momentum in horizontal and vertical direction respectively and \( b \) the bathymetry, i.e. the ocean depth relative to the geoid. \( g \) is the gravitational constant (\( g = 9.81 \text{ m s}^{-1} \)). \( S(t,x,y) \) is the optional source term which may be used to model additional factors such as friction and the coriolis force. For this implementation, the source term is used to model the bathymetry using the source term in equation (3.2). In the following, the abbreviation ‘SWE’ is used for the actual implementation of the Shallow Water Equations model.

\[
\begin{bmatrix}
h \\
hu \\
hv 
\end{bmatrix}_t + \begin{bmatrix}
hu \\
hu^2 + \frac{1}{2}gh^2 \\
huv 
\end{bmatrix}_x + \begin{bmatrix}
hv \\
huv \\
hv^2 + \frac{1}{2}gh^2 
\end{bmatrix}_y = S(t,x,y) 
\tag{3.1}
\]

\[
S(x,y) = \begin{bmatrix}
0 \\
-ghB_x \\
-ghB_y 
\end{bmatrix} 
\tag{3.2}
\]

The approach to numerically solving the shallow water equations is to discretize the domain and to reduce the differential equations to one-dimensional Riemann problems at the edges of neighbouring cells (see [28]). Then, the ‘Wave Propagation’ scheme is used to update the values of the cells (see [29]).

3.1 Discretization and Domain Decomposition

The discretization scheme used by SWE is finite volume discretization. This transforms the continuous domain to a discrete domain \( \Omega \) consisting of rectangular cells.

\[
\Omega := \bigcup_{i,j} C_{i,j} \quad \text{with states} \quad Q_{i,j} = [h, hu, hv]^T 
\]

The arising one-dimensional Riemann problems at the cell edges are then solved in a black-box manner by a suitable Riemann solver, which provides the net updates for two
neighbouring cells given the cell states. This has to be done for all four edges of every cell, to provide net updates coming from the left, right, above and below. To model the boundaries of the simulation domain, an additional layer of grid cells around the domain is used, the ‘ghost layer’. These cells are initialized with the same bathymetry as the cells they are bounding to the inside. Before each iteration of the simulation, water height and momentum is set corresponding to the current state of the neighbouring cells in order to let water flow out of the simulation domain. For parallelization purposes, the domain may be decomposed into rectangular blocks (Figure 3.1, Page 8). In this case, block boundaries which do not border the simulation domain but another block use the corresponding ghost layers to connect with the neighbouring block. This is denoted by the coloring in figure 3.1, the grey parts are the ‘outer’ ghost layers, connecting ghost layers are in the colour of the respective neighbour. The corner cells may be disregarded, since they are not needed for updating the inner cells (see 3.2).

### 3.2 Update Scheme

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

Equation (3.3) shows how cells are updated from state $Q^n$ to state $Q^{n+1}$ using net updates in horizontal $(A^\pm \Delta Q$, left and right) and vertical $(B^\pm \Delta Q$, above and below) orientation. The update scheme also depends on cell size, which is uniform over the whole domain, and the timestep. The maximum timestep for a cell edge is derived from the local maximum wavespeed and is returned by the solver after computing
the net updates at that edge. Therefore, the maximum timestep has to be reduced over all edges and forces all simulation blocks to communicate before being able to update their cells. Figure 3.2 visualizes how a simulation block progresses one timestep: Neighbouring cell states are passed to the solver which returns a net update for each cell corresponding to one of the net updates used in equation (3.3). In the case of a vertical (red) edge, the resulting net update of the left cell would be plugged in as $A^-$ when updating this cell. Figure 3.2 also shows that ghost layers are needed to generate a valid state for the outermost cells, as well as the irrelevance of the corner cells. This also means that no exchange of corner cell states between diagonally opposite blocks is needed.

3.3 Solver

Multiple solvers for the one-dimensional edge-local Riemann problems are available (see [28]), for SWE the \textit{f-wave solver}, the \textit{Augmented Riemann solver} and the \textit{Hybrid solver} is used. The properties of these solvers concerning this work are the computational cost, features and correctness of these solvers. The Augmented Riemann method is more expensive than the f-wave solver, but can handle inundation and is closer to the exact solution. The Hybrid solver combines the f-wave and Augmented Riemann solver; it will by default use the Augmented Riemann solver, but falls back to the f-wave solver if both cells are wet (no inundation) and the differential in water height and velocity is within a certain limit. For benchmarking, the hybrid solver is interesting regarding the Charm++ and AMPI implementations, since it is a possible reason for load imbalances.
4 Implementation

The implementation of the shallow water equations simulation is loosely based on an existing framework made by the Chair for Scientific Computing at the Technical University of Munich [30], but was almost entirely rewritten for the purpose of accommodating the design of UPC++ and Charm++.

The core component of all implementations is the abstract SWE_Block interface, which represents a patch of arbitrary size of the computation domain (Figure 3.1, Page 8) and holds all cell state data within this patch. Additionally, a 2D-Array wrapper, Float2D, and classes to handle input data are used. The pseudo-UML diagram (Figure 4.1, Page 11) shows an abstract overview of the design. From SWE_Block, specialized classes are derived which implement the actual computation of cell net updates, the wave propagation update scheme (Section 3.2, Page 8) and the communication components specific to UPC++, MPI and Charm++. In order to make the interface compatible with UPC++ and its global pointers in terms of data storage, using simple getter/setter functions etc. — considering that the other frameworks use C native pointers —, SWE_Block is additionally templated to use data in the form of Float2DNative or Float2DUpcxx wrapper objects.

Float2DUpcxx objects allocate memory in the shared memory segment of a UPC++ rank, such that it may be accessed by other ranks, while Float2DNative objects wrap standard pointers. Both classes are derived from Float2D, which implements the standard array operation ‘[]’ for accessing elements. A specialty of the Float2D wrapper is that it stores the data column-major, meaning that the second index is stored with stride 1.

The initial cell states for the simulation are provided by derivations of the SWE_Scenario interface, which act as a proxy to data sources in the form of netCDF input files, hardcoded data or dynamically generated data. The backend for processing netCDF input files is Asagi, which natively handles details such as the mapping of coordinates to the appropriate cell of the input grid and calculating water height from the bathymetry and an ocean floor displacement (See [31]).

In all implementations, the main method is in charge of computing the amount and dimensions of the SWE_Blocks, their origin within the simulation domain and the cell width. These parameters are derived from the layout of the input domain, the resolution which was requested at the start of the program and the available number of processing
Figure 4.1: Pseudo-UML Design Overview
elements. Generally, one rank is used per processing element (per node in the hybrid case) and one simulation block is spawned per rank. In the Charm++ implementation, the amount of simulation block Chares corresponds to the hardware layout the same way, the internal scheduler then distributes the Chares over the physical processors. All needed simulation blocks are then instantiated with the corresponding parameters. A SWE_Scenario is also instantiated, and the simulation blocks are initialized to it: Each simulation block knows its origin within the simulation domain as well as the cell width and count in both dimensions, such that they are able to request values from the scenario object at the proper coordinates corresponding to its cells.

The parallel program flow during the actual simulation is quite simple, since each block can perform all computations within an iteration on its own, only the global maximum timestep has to be broadcasted over all blocks before updating (compare $\Delta t$ in equation (3.3)). The exchange of copy layers takes place before each new iteration and implicitly synchronizes all blocks (Figure 4.2, Page 12).

### 4.1 MPI

For MPI communication, only the rank ids of the sender/receiver and a shared message tag are needed. The message tags are defined as global constants and identify the
value (out of the water height and horizontal/vertical momentum) and the border position (left, right, bottom, top). Combined with the rank of the sender/receiver, this uniquely identifies any message. Since copy layers (Section 3.1, Page 7) may be oriented horizontally w.r.t. to the computational domain depending on the corresponding border, the stride between the relevant elements in memory have to be considered. MPI offers custom datatypes to solve this problem. Figure 4.3 shows how a horizontal copy layer is being sent. In lines one to three, the custom MPI Vector is defined: It consists of nx (number of cells in x-direction) elements, a single block and the stride is ny + 2 (number of cells in y-direction plus two ghost cells). In lines five to seven, a non-blocking MPI send directive is used to send one HORIZONTAL_BOUNDARY to the bottom neighbour. The data starts at index (1,1) of the Float2D array storing the water height, which corresponds to the bottom left cell of the patch of the computational domain the current block is representing. The MPI request is immediately discarded since a simulation block does not care about when its copy layers are received, it only needs to wait for the reception of its own ghost layer values (Figure 4.2, Page 12). For this, MPI_Waitall() is used after all MPI_Irecv() calls were made in order to block until all copy layers are received.

4.2 UPC++

The UPC++ implementation makes use of one-sided communication. Because data is not explicitly sent but global pointers are used instead, neighbouring blocks need to exchange global pointers, starting indices and strides of their copy layers at the end of initialization, before the simulation loop starts. Because of the usage of one-sided communication, there is also a need to explicitly synchronize all simulation blocks after updating the cells, since a block may not have updated all cells when another block is ready to get values from it (see 4.2). For the communication, rget() and rget_strided
are being used, the necessary pointers, indices, strides and counts obviously stay unchanged during the whole simulation. To allow the UPC++ runtime to handle the data flow at its own discretion, the futures returned by `rget()` and `rget_strided()` are conjoined and collectively waited on after issuing all communication requests. UPC++ also provides a simple reduction mechanism equivalent to `MPI_Allreduce`, which is used for finding the global timestep.

### 4.3 Charm++

Since Charm++ does not work with the notion of ranks, but calls entry methods on ‘Chare’-Objects which are then executed in parallel (Section 2.4, Page 5), the control flow is handled within the simulation blocks (simulation chares) themselves. To accomplish this, the ‘Structural Dagger (SDAG)’ (Section 4.3, Page 14) paradigm is used. The excerpt of the `SWE_Block` interface file illustrates this: At line 10, the Chare is declared to be an array, since multiple blocks are being spawned. This happens inside the Main Chare using the corresponding Charm++ methods. Line 11 is the start of the entry method which contains the simulation loop (Figure 4.2, Page 12). It always starts by sending the blocks own copy layers out to the blocks neighbours. This means selecting the relevant cell values and manually copying them into a message. Messages are the best performing method to pass data according to the Charm++ documentation, they have to be declared inside an interface file (line 1). Since the message is built by hand, the striding is handled by copying elements one by one, each time skipping the stride. Then the SDAG `overlap` statement is used such that the Chare ‘listens’ for any incoming copy layer message (line 11, other `when` statements are omitted). Such an incoming message triggers the block to process the message; once all needed messages are processed, the block proceeds to calculate the net-updates for each cell. Once again, the global timestep has to be reduced. Charm++ carries out reductions using callback functions: Chares may contribute to a reduction while also specifying a callback function, which is called as soon as the reduction is complete. In this case, the callback function is declared at line 24, its implementation progresses the SDAG code at line 20 by calling the `reductionTrigger()` after updating the global timestep.
4 Implementation

message copyLayer {
  Boundary boundary;
  bool containsBathymetry;
  float b[];
  float h[];
  float hu[];
  float hv[];
};

array [1D] SWE_DimensionalSplittingCharm {
  entry void compute() {
    while(...) {
      serial { sendCopyLayers(); }
      overlap {
        if(boundaryType[BND_LEFT] == CONNECT) {
          when receiveGhostLeft(copyLayer *msg)
            serial { processCopyLayer(msg); }
        }
      }
      serial { computeNumericalFluxes(); }
      when reductionTrigger()
        serial { updateUnknowns(maxTimestep); }
    };
    entry [reductiontarget] void reduceWaveSpeed(float maxWaveSpeed);
    entry void receiveGhostLeft(copyLayer *msg);
  }
};

Figure 4.4: Block Chare Interface Excerpt
5 Benchmarks

5.1 Input Data

For the benchmarks, historical data of the 2010 Chile and the 2011 Tohoku, Japan earthquakes are used. The data consists of the bathymetric profile of the domain as well as the displacement induced by the earthquake. Figure 5.1 shows the bathymetric profiles of the corresponding domains.

5.2 Method

All benchmarks were performed on the cluster ‘CoolMUC-2’ at the Leibniz-Rechenzentrum at the Technical University of Munich. The cluster consists of 384 Nodes, each fitted with 28-way Intel Xeon E5-2697v3 (Haswell) CPUs and 64GB RAM. The nodes are using a FDR14 Infiniband interconnect, which is supported and used by all parallel frameworks. Measured time corresponds to the completion time of the simulation, excluding initialization and I/O. The cluster supports allocations of up to 60 nodes, totalling to 1792 processing units. The MPI implementation in use is MPICH.

5.3 Results

The first benchmarks compare the performance of all frameworks and OpenMP on a single node, where OpenMP is expected to perform near optimally. As can be seen

(a) Japan 2011
(b) Chile 2010

Figure 5.1: Bathymetric Profiles of the Simulation Domains
in 5.2, all implementations scale almost linear. Charm++ scales slightly better than other implementations, but is slower. This is most likely due to the implementation needing to pack the messages ‘by hand’ and does not use native methods to deal with striding such as a MPI_Vector custom datatype or rget_strided(), which know strides and sizes beforehand and may deal with the request in any way the runtime considers most efficient. As the core count increases, the domain is being split into smaller patches and the copy layers shrink as well. Then, Charm++ performs on par with the other frameworks as well, additionally the later benchmarks do not support significant scaling advantages of Charm++, therefore the interpretation is plausible. The MPI implementation does not lag behind the reference OpenMP implementation, this is to be expected since MPICH is able to take full advantage of KNEM [14]. The

![Figure 5.2: Shared Memory Performance (Tohoku 2011, 2000×2000 cells)](image)

results of the strong scaling benchmark (figure 5.4) using the smaller Tohoku domain once again shows equal performance and almost linear scaling, the runtime is very close to being halved for each doubling of the processor count. Performance gains are limited as soon as the simulation blocks are getting very small, using 448 processing elements, each regular block only consists of 8 by 4 cells. The next doubling of the processor count provides a speedup of 1.67 compared to 1.85–2 for the previous steps.
The UPC++ implementation shows strange behaviour: with certain configurations (e.g. 56 or 224 processing elements), the runtime increases to very large values which do not match other test results using the same executable. This behaviour is reproducible but the reasons are not clear as of yet, the computed simulation results are also correct and equal those being generated by the other implementations.

![Strong Scaling (Tohoku 2011, 3500×2000 Cells)](image)

Figure 5.3: Strong Scaling (Tohoku 2011, 3500×2000 Cells)

Strong scaling using the larger Chile domain continues the same trend: equal performance and strong scaling which starts to slow down significantly as soon as the simulation blocks drop to a small size, in this case 8 by 6 cells. The UPC++ implementation behaved as before and could only complete the first configuration within a reasonable timeframe.

The results of the weak scaling benchmark (figure 5.5) were obtained by doubling the resolution of the simulation for each doubling of processing elements. For the first test using 28 cores, 2000 × 2000 cells were simulated. The next step used 3200 × 2500 cells, 4000 × 4000 and finally 6400 × 5000 cells. Internal timers show that the computation time does indeed stay largely unchanged as is expected, but in comparison with the strong scaling benchmark, the communication load doubles for every further test.
5 Benchmarks

The strong scaling benchmark using a hybrid approach (figure 5.6 scales quite bad, which is expected since only one of the 28 cores on each node is doing inter-node communication.

![Figure 5.4: Strong Scaling (Chile 2010, 7770×5828 Cells)](image)

The strong scaling benchmark using a hybrid approach (figure 5.6 scales quite bad, which is expected since only one of the 28 cores on each node is doing inter-node communication.)
Figure 5.5: Weak Scaling (Chile 2010, Variable Resolution)
5 Benchmarks

Figure 5.6: Strong Scaling, Hybrid Implementation (Chile 2010, 7770×5828 Cells)
6 Conclusion and further Work

A goal of both Charm++/AMPI and UPC++ is to be easily deployable and portable. Compiling the libraries themselves worked fine on the LRZ cluster. To compile source code against either of them, some additional configuration work has to be integrated in the build process, but all of this is properly documented. The Charm++ interface files pose an additional challenge, as these files have to be ‘compiled’ before the actual compilation takes place. Refactoring becomes more complicated too, since entry methods have to be declared in both the header as well as the interface file (except for SDAG entry methods). UPC++ provides great documentation including a detailed language specification. Charm++ and AMPI documentations sometimes rely a lot on examples, which don’t necessarily contain all needed information. Both frameworks do interoperate well with the Slurm Workload Manager and deploy over nodes without any additional configuration; A disappointment was the GASNet communication layer of UPC++ not supporting Intel OmniPath, therefore the usage of an older cluster was required.

As concepts, both PGAS and the Charm++ approach are very interesting, PGAS makes designing complex parallel applications and control flows much less cumbersome and provides all the tools to efficiently exploit the available hardware: No UPC++ function blocks implicitly such that the programmer is always aware of parallel flow, futures can be used to keep track of the readiness of data and the virtual global address space simplifies shared data structures and optimizing fine-grained parallelization structures. Charm++ is very expressive, object orientation is a natural approach to modeling many logical structures in an application, the same is true for parallel applications. The encapsulation properties of Chares combined with the message driven backend and a powerful scheduler are the core traits which enable scalability. AMPI provides these features for already existing MPI applications and has been shown to perform at least as well as MPICH on a basic level.

The main takeaway from this analysis is that all tested frameworks do indeed perform at the level of the reference MPI implementation. There is not much room for improvement, since the SWE model is perfectly suited to plain message passing. None of the benefits UPC++ provides can be exploited, because the inherent alternation of computation and communication allows no overlap at all; the computation has to unconditionally wait on the completion of the communication. UPC++ futures are
not used to their full potential, since they can’t be integrated in the computational part for the same reasons. Load balancing does not happen at large scale because the rank/chare mapping is a one to one relation which means that vacating a processing unit will never lead to a performance gain. AMPI can not take advantage of the overdecomposition it applies by design: virtual processes could not be advantageously migrated, which the distribution of compute times across all ranks clearly shows; therefore the implementation was also unable to consistently outperform other runtime systems. A custom scheduling policy which intelligently considers cell states might help with adapting to the application, such as clustering simulation blocks with mostly positive (island) bathymetry on a small number of processing units, thus leaving more computing power for where water is actually moving.

For future work on SWE, some issues are still to be resolved, most prominently the erratically behaving UPC++ implementation. Even though the hybrid approaches did not promise any performance gains considering the results of the shared memory benchmark and the benefits of overdecomposition, it remains to be seen how a fully parallelized hybrid implementation compares to the current results. Considering further research on alternative runtime systems, it might be very interesting to implement an application suited for a global address space framework using both MPI and UPC++, to examine the practical advantages of the approach. Also, performance gains have been claimed for all of the runtime systems, though what is apparently missing is a fully optimized, production level application implementing complex parallel flow in order to go beyond microbenchmarks and test performance in a ‘real life’ context.
# List of Figures

2.1 Simple OpenMP loop-level parallelism example. Source: [13] . . . . . . . . 3  
2.2 Partitioned Global Address Space (PGAS) . . . . . . . . . . . . . . . . . 5  
3.1 Patch Domain Decomposition . . . . . . . . . . . . . . . . . . . . . . . . 8  
3.2 Update Scheme . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9  
4.1 Pseudo-UML Design Overview . . . . . . . . . . . . . . . . . . . . . . . . 11  
4.2 Exemplary parallel flow with only two ranks . . . . . . . . . . . . . . . . 12  
4.3 MPI Copy Layer Exchange . . . . . . . . . . . . . . . . . . . . . . . . . . 13  
4.4 Block Chare Interface Excerpt . . . . . . . . . . . . . . . . . . . . . . . . 15  
5.1 Bathymetric Profiles of the Simulation Domains . . . . . . . . . . . . . . . 16  
5.2 Shared Memory Performance (Tohoku 2011, 2000×2000 cells) . . . . . . . 17  
5.3 Strong Scaling (Tohoku 2011, 3500×2000 Cells) . . . . . . . . . . . . . . 18  
5.4 Strong Scaling (Chile 2010, 7770×5828 Cells) . . . . . . . . . . . . . . . 19  
5.5 Weak Scaling (Chile 2010, Variable Resolution) . . . . . . . . . . . . . . 20  
5.6 Strong Scaling, Hybrid Implementation (Chile 2010, 7770×5828 Cells) . 21
Bibliography


Bibliography


