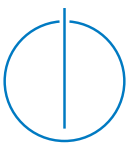
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Efficient I/O Strategies for Checkpointing and Visualization in sam(oa)^2

Christoph Honal



DEPARTMENT OF INFORMATICS

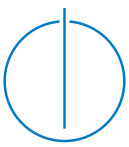
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Efficient I/O Strategies for Checkpointing and Visualization in sam(oa)^2

Effiziente IO-Strategien für Checkpointing und Visualisierung in sam(oa)^2

Author:	Christoph Honal
Supervisor:	Univ.-Prof. Dr. Michael Bader
Advisor:	Leonhard Rannabauer, M.Sc.
Submission Date:	15.03.2019



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

Munich, 15.03.2019 Christoph Honal

Acknowledgments

The author gratefully acknowledges the compute and data resources provided by the Leibniz Supercomputing Centre (www.lrz.de).

Abstract

The implementation of large-scale finite-element simulations on distributed systems requires specific memory management and disk I/O handling. This is due to the immense spatial and temporal resolutions and the grade of heterogeneous parallelization, i.e. hybrid MPI/OpenMP. In this thesis, a new approach for storing, visualizing and restoring (referred to as checkpointing) simulation data in the sam(oa)^2 project is presented, introducing a parallel storage system based on the HDF5 and XDMF file formats. The presented implementation measurably reduces disk space usage and file system strain, while improving run-time performance compared to previous output formats.

Contents

1 Introduction

1.1 The sam(oa)^2 Project

The *sam(oa)*^{2} software project, designed constructed and implemented by Oliver Meister in 2016 is a framework written in `Fortran` for fluid dynamic problems primarily in the field of geosciences [5]. It employs a finite-element or finite-volume approach to provide high scalability on typical high-performance systems, such as the LRZ CoolsMUC-2 cluster [3] used in this thesis. The numerical problems are formulated as a set of partial differential equations, which are solved by discretization using said finite-element methods.

1.1.1 Adaptive Refinement

One of the key features of this framework is the ability to dynamically adapt the domain's refinement in respect to element size and count, allowing for focus of computational power on critical simulation parts. This is achieved by partitioning the quadratic domain recursively into half-sized right-angled isosceles triangles, and traversing them according to a *Sierpinski space-filling curve* [5] as can be seen in figure 1.1.

Depending on the relative change or *importance* of an element, it may be split into two when additional spatial resolution is needed. Similarly, neighboring triangles may be merged when a low amount of change occurs, freeing computational power. These operations are referred to as *refine* and *coarsen*, generally performed in an *adaption* traversal.

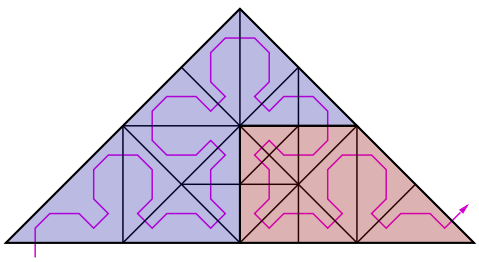


Figure 1.1: Adaptive triangular mesh, taken from [5], p. 16

Sections of the traversal curve are then distributed across all available compute resources to be traversed, while making sure that the load (i.e. the number of elements crossed by this section) is as balanced as possible. As a result of this strategy, the simulation maintains an efficient work distribution across the available compute hardware (see section 1.1.3). Consequently, in order for the framework to efficiently process and parallelize operations on cells, those operations have to be implemented using a traversal.

1.1.2 Shallow Water Equations

One of the applications of the *sam(oa)*^{2} framework is computing numerical solutions for the *Shallow Water Equations*, or *SWE* for short. These equations can be derived from the *Navier–Stokes equations* by depth-integrating [9], assuming some simplifications - such as that the spatial horizontal scale of the domain (i.e. its height) is much smaller than its vertical scale.

These equations can then be used to model large-scale tsunami simulations [4], requiring the solution of a *nonlinear Riemann problem*. Luckily, several numerical approaches exists, and some were implemented in the *SWE* module of *sam(oa)*^{2}. Here, the Riemann problem is solved amongst others by a *Roe Riemann solver* [4].

This specific use case of the framework is the starting point and environment for the work done in this thesis.

1.1.3 Parallelization Model

The *sam(oa)*^{2} framework uses a heterogeneous parallelization approach, consisting of thread-level and node-level parallelism. Thread-level parallelism is achieved with the *OpenMP* [8] (*Open Multi-Processing*) compiler extensions. Node-level work parallelism is made possible by using the *Message Passing Interface*, or *MPI* [6] for short.

Figure 1.2 shows an example configuration, using two nodes with two threads each, computing a problem split into sixteen sections.

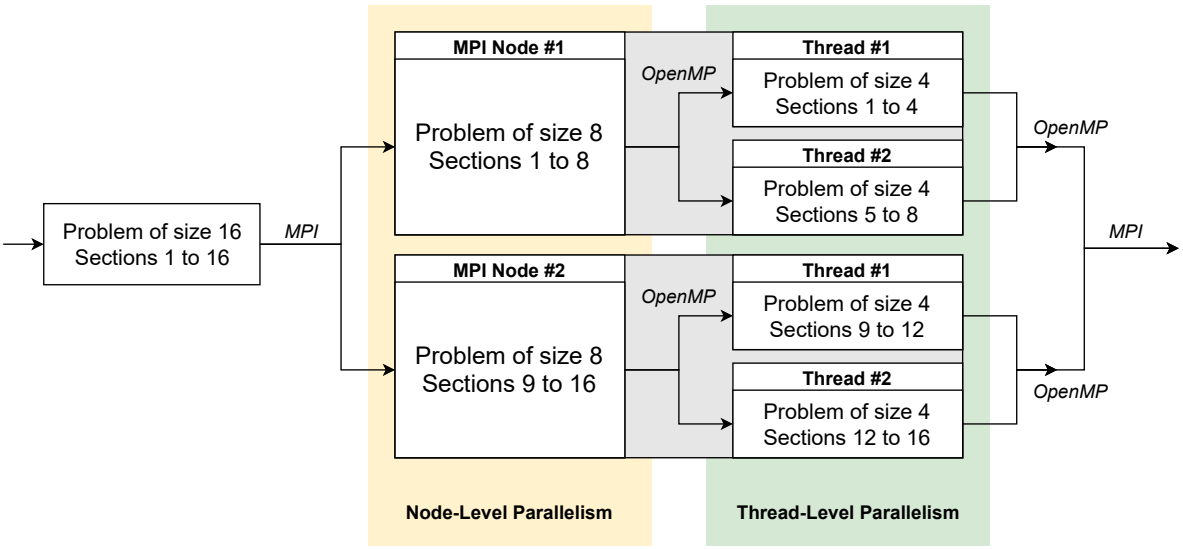


Figure 1.2: Example of heterogeneous parallelization

2 Scope of this Thesis

The stating point for all work done in this thesis is the existing *sam(oa)*^{2} project, and specifically the *SWE* module, as mentioned in chapter 1. Due to several shortcomings of the previously existing XML-based output writer (see 6), such as a disproportionately high amount of individual files, and general low performance, a new XDMF [16] and HDF5 [12] (see chapter 3) based writer was implemented (see section 4.3), which improves on said problems.

Additionally, it was ensured that the generated output files could be visualized using the *ParaView* [2] software (see section 4.5) just as before. This was done to ease migration from the previous file format. The module was also extended to be able to read generated output files back in, in order to continue an aborted or crashed simulation. This allows for reentrant simulations, and thus for more efficient compute resource usage, and is referred to as *checkpointing* (see section 4.4).

A special focus was laid on parallel performance, or at least parallel compatibility. This ensures smooth integration of the new module into the existing heterogeneous OpenMP [8] and MPI [6] environment, resulting in a performance several times better than the existing solution (see chapter 5).

3 Utilized Storage Technologies

3.1 The Hierarchical Data Format (HDF)

The *Hierarchical Data Format* version 5, or *HDF5* for short, is a binary data format developed and supported by the *HDF Group* [10]. Its high performance, parallel capabilities and excellent handling of huge datasets did popularize it in all sorts of scientifically and industrial applications [12].

3.1.1 History

The HDF file format was originally developed at the *National Center for Supercomputing Applications* [13] at the *University of Illinois at Urbana-Champaign* [14] as a general and portable scientific data format [10]. The latest version 5 offers many advantages above the previous version 4; such as a unified and simplified API, and support for much larger datasets. Nevertheless, version 4 continues to receive support to maintain backwards compatibility.

3.1.2 Internal structure

A HDF5 file consists of groups and datasets. As the verb *hierarchical* in its name implies, these elements are organized in a graph-like structure; similar to the tree of a traditional file system such as FAT32 or NTFS. In fact, a HDF5 file could be understood as a kind of sophisticated *Microsoft Windows cabinet file* or *Unix Tarball*, but storing raw datasets instead of files. However, unlike traditional file systems, HDF5 groups may be linked not only in the shape of a hierarchical tree, but also in an arbitrary graph. This allows for cyclic linkage and data deduplication.

The HDF5 file itself is structured in two main parts, a small section of metadata containing symbols and type information, and a large raw data store. Due to the individually accessible metadata, the structure of the file and the location of any dataset can be read with minimal effort, avoiding having to search in the potentially huge data section. Datasets can be of any dimension and rank, and contain data of any standard types such as integer, floating point numbers and character strings, or even user defined composite types.

3.1.3 Example

```
1  HDF5 "example.h5" {
2    GROUP "/" {
3      DATASET "Foo" {
4        DATATYPE H5T_IEEE_F32LE
5        DATASPACE SIMPLE { ( 3, 2 ) / ( 3, 2 ) }
6        DATA {
7          (0,0): -0.8, -0.5,
8          (1,0): 0, 0.5,
9          (2,0): 0.5, -0.75
10       }
11     }
12     DATASET "Bar" {
13       DATATYPE H5T_STD_I32LE
14       DATASPACE SIMPLE { ( 1, 3 ) / ( 1, 3 ) }
15       DATA {
16         (0,0): 0, 1, 2
17       }
18     }
19   }
20 }
```

Listing 3.1: Text definition of an exemplary HDF5 file

To further clarify the structure of a HDF5 file, a short example is given in text form in 3.1. This file contains two datasets, `Foo` and `Bar`. `Foo` stores three records of size two and type 32 Bit Float (see lines 3-11), whereas `Bar` stores one record of size three and type 32 Bit Integer (see lines 12-18).

The schema 3.1 provides a visual presentation of the previously defined HDF5 file. Note the metadata and the raw data sections; marked in blue and red respectively.

3.2 The Extensible Data Model and Format (XDMF)

The XDMF format is a widely supported composite XML-based file format that may be validated according to its formal definition [18]. A XDMF file is used to further augment one or more HDF5 files by providing semantic annotations. Thus, it is inherently coupled to the HDF5 file format. The additional semantic information may be used by data processing programs like *ParaView* [2] to correctly visualize the

Example HDF5 File		
Metadata		
General Data		
Dataset Foo Description		
	Type	32 Bit Float
	Size	3 x 2
Dataset Bar Description		
	Type	32 Bit Integer
	Size	1 x 3
Dataset Foo		
-0.8	-0.5	
0	0.5	
0.5	-0.75	
Dataset Bar		
0	1	2

Table 3.1: Visual representation of the file defined at 3.1

underlying HDF5 datasets.

3.2.1 Motivation

Although HDF5 already contains a metadata section, XDMF encodes additional information which goes far beyond the HDF5 metadata section. Usually, a HDF5 file only contains the absolute minimum of metadata necessary to locate and read datasets correctly, but no semantic information about how to interpret the stored data. This is where XDMF comes into play, providing typically a specification on how to interpret the raw HDF5 data as geometric data and associated attributes across a special or temporal domain.

A single XDMF file may reference multiple HDF5 files, essentially acting as a consolidating index structure. This is especially useful when selectively loading steps from a series stored across multiple HDF5 files is required. The application may lookup the HDF5 dataset positions (in the so called *heavy* data) using the *light* XDMF index, resulting in higher performance due to reduced disk I/O.

3.2.2 Example

In order to show the relation between XDMF and HDF5, the file 3.2 augments the example HDF5 data file 3.1 from section 3.1. Using this file, the HDF5 data may be

interpreted as a series of spatial grids across a temporal axis. Each grid object may be different, however, this file only defines one time step. This lone time step contains a single triangle, which is defined using the `Foo` HDF5 dataset as vertex data, and the `Bar` dataset as topology definition, i.e. vertex interconnection information.

Additionally, a cell-bound attribute `SomeAttribute` is defined and set to an immediate value defined in the XDMF file itself. This practice is not recommended for large amounts of attribute data, as it blows up and pollutes the XDMF file, which would defy the idea of a lean and fast indexing structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<Xdmf Version="3.0">
  <Domain>
    <Grid GridType="Collection" CollectionType="Temporal">
      <Grid>
        <Time Value="7" />
        <Geometry GeometryType="XY">
          <DataItem Format="HDF" NumberType="Float" Dimensions="3␣2">
            example.h5/Foo
          </DataItem>
        </Geometry>
        <Topology TopologyType="Triangle">
          <DataItem Format="HDF" NumberType="Int" Dimensions="1␣3">
            example.h5/Bar
          </DataItem>
        </Topology>
        <Attribute Name="SomeAttribute" Center="Cell">
          <DataItem Format="XML" NumberType="Int" Dimensions="1">
            42
          </DataItem>
        </Attribute>
      </Grid>
    </Grid>
  </Domain>
</Xdmf>
```

Listing 3.2: Example XDMF file referencing the HDF5 file defined at 3.1

The rendering 3.1 shows an approximation of the interpretation of these XDMF and HDF5 files, which could be generated by an application such as *ParaView*. Note the cell

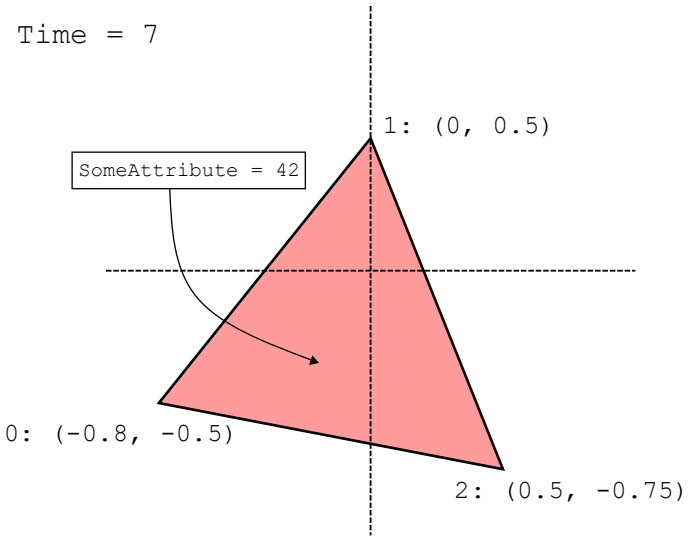


Figure 3.1: Approximate rendering of the XDMF file defined at 3.2

bound arbitrary attribute `SomeAttribute` and the time information.

3.2.3 Implementation Notes

Although the XDMF community provides an API for `C++`, `Python`, and `Fortran` [17], for this thesis a custom XDMF output library was developed. This is due to the insufficient flexibility of the official XDMF API at the time of writing, as fine-grained control over the generated HDF5 and XML files was needed. Also, the official XDMF library only provides rough MPI parallelization support, where again precise control was needed (see chapter 5).

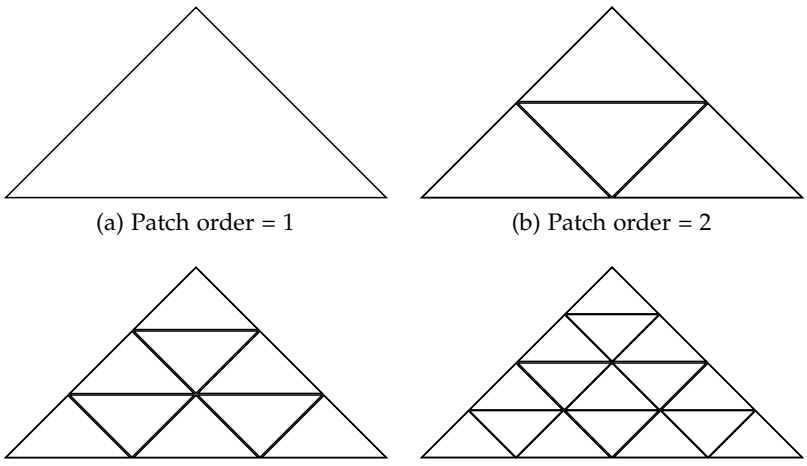
The XML file was written by utilizing raw `Fortran` text output, and the HDF5 files were generated and read back using the much more sophisticated, albeit significantly more complex HDF5 API [11]. Similarly, reading XML files was made possible by the excellent *Fortran Library for XML* [1] (*FoX* for short), which implements a full XML DOM parser.

4 Storing and Restoring Domains

4.1 Patch Geometry

Patch geometry describes the concept of further dividing all cells into sub-cells. The amount and method of tessellation is the same for each cell, which then becomes a *patch*.

Only those patches are individually traversed by the sam(oa)^2 framework, not the sub-cells.



(a) Patch order = 1 (b) Patch order = 2

(c) Patch order = 3 (d) Patch order = 4

Figure 4.1: Patch geometry for different patch orders

Figure 4.1 shows a tessellated cell with different grades of subdivision, referred to as *patch order*. Each patch contains $c = p^2$ cells, p being the patch order.

The XDMF writer does not output patch information for visualization purposes, but stores it separately in order to correctly reconstruct a domain when needed.

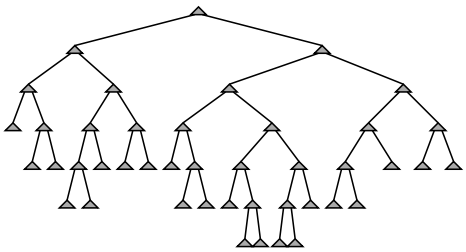


Figure 4.2: Refinement tree of domain shown in figure 1.1, taken from [5], p. 16

4.2 Persistent Refinement Trees

The way a domain is partitioned into cells is referred to as its *Refinement tree*. This stems from the fact that the state of refinement can be expressed using a *binary tree*. Figure 4.2 shows the refinement tree for the introductory partitioning example shown in figure 1.1. Only the leaves of this tree are actual cells contained in the domain, intermediate nodes can be interpreted as cells which have been previously refined.

To ensure the independent cell processing required for parallelization, the framework traverses a cell using only the data associated with this exact cell. The framework does not explicitly provide the refinement tree, nor the traversal curve (see section 1.1.1). In fact, the refinement topology of a domain is implicitly stored by the order of cells on a series of processing stacks [5]. Only the level of refinement of the currently traversed cell is accessible, this corresponds to the depth of a leaf in the refinement tree.

The restoration functionality has to be implemented in form of an recursive adaptive traversal, and always starts with a non-partitioned domain (see section 4.4). Because of this algorithms recursive nature, there needs to be a system to check whether an arbitrary cell is adequately refined.

To provide such functionality, an index to look up cells based on their level of refinement and position is stored alongside the cell data. Several different data structures and techniques were evaluated in terms of access speed and disk space requirement. It was found that a *hash table* provided great access performance compared to not using any indexing structure. Not having any indexing structure would require scanning the entire list of cells. The hash table does require too much additional disk space, like storing the full refinement tree would.

4.2.1 Position-Based Cell Hashing

The employed hash function maps the spatial position and orientation of a cell to a 32 Bit signed integer. It is numerically stable for refinement levels up to about 29,

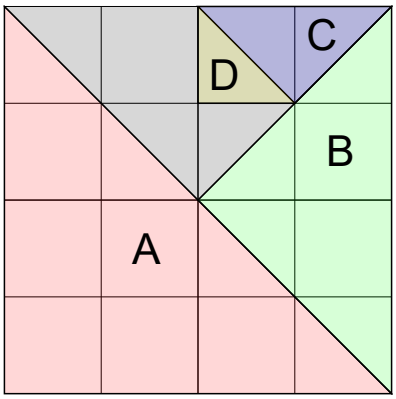


Figure 4.3: Sample domain with quadtree used for hashing

because each level of possible refinement (i.e. maximum domain refinement depth) on average requires one Bit, and the cell orientation requires two Bits. Although the hash is computed internally using 64 Bit logic, later steps of the implementation only handle the lower 32 Bits.

Figure 4.3 shows a sample domain with some degree of refinement. Because of the right-angled nature of all cells, and the way they are split down the middle, it is possible to fit every cell into a quadtree as shown. The quadrants are then numbered anti-clockwise from $p_i = 0$ to 3, and each possible eight orientations and shapes (see *Plotter Type* in table 4.2) of a cell is assigned an integer from $w = 0$ to 7.

To compute the hash key of a cell, first its bounding sub-quadrant is determined by a recursively descending binary coordinate comparison. During this decent to depth n, the indices of each surrounding quadrant at depth i are joined into a string of bits. Following that, the cells shape index w is joined with this bit-string:

$$k(E) = w \vee \bigvee_{i=0}^{n} (p_i << (i * 2 + 3)) \tag{4.1}$$

With the operator $<<$ denoting a bit-wise left shift. In figure 4.3, the theoretical hash key values of cells A to D would be as follows:

$$k(A) = 0 \vee 7 = 0b\ 111 = 7 \tag{4.2}$$

$$k(B) = 0 \vee 6 = 0b\ 110 = 6 \tag{4.3}$$

$$k(C) = (1 << 3) \vee 0 = 0b\ 01\ 000 = 8 \tag{4.4}$$

$$k(D) = (1 << (2 + 3)) \vee (2 << 3) \vee 7 = 0b\ 01\ 10\ 111 = 55 \tag{4.5}$$

In the actual implementation, the hash function additionally accounts for several special cases and guarantees a nonzero result.

4.2.2 Double Hashing

The hash table itself is implemented using *double hashing* to resolve collisions. Given the maximum amount p of cells to store, the size h of the hash table is computed by

$$h = nextprime(2p) \tag{4.6}$$

with $nextprime(n)$ being a function which returns the smallest prime that is greater than n. The factor of 2 ensures a maximum *load factor* of 50%, ensuring good access performance by reducing the amount of hash collisions. The actual hash function $h(k, i)$ is given by

$$h(k, i) = (h_1(k) + i * h_2(k)) \ mod \ h \tag{4.7}$$

$$h_1(k) = k \ mod \ h \tag{4.8}$$

$$h_2(k) = k \ mod \ prevprime(h) \tag{4.9}$$

with $prevprime(n)$ being a function which returns the biggest prime that is smaller than n, similar to the $nextprime(n)$ function described above. The parameter i denotes the number of previous hash collisions.

4.3 Storing Cell Data

4.3.1 Storage Layout

As demonstrated in chapter 3, the new writer uses a hybrid file format consisting of one XDMF index file and several HDF5 database files.

Each MPI node stores the cell data of all sections assigned to this node in local memory buffers. These buffers are then written to the corresponding HDF5 datasets in parallel using concurrent MPI-IO via parallel HDF5 (see chapter 5).

Because the memory layout of these buffers was deliberately chosen to match the internal HDF5 storage layout, writing to a dataset is very fast. Data only has to be copied from memory onto the disk, not converted or reordered.

HDF5 Database

The generated HDF5 files are internally structured using HDF5 groups. The domain data is stored in several datasets, as shown in table 4.1. All datasets are made up of 32-Bit records of type integer or floating point number.

According to the table 4.1 and the corresponding symbol definition in table 4.2, the disk space usage of a single step encoded in a HDF5 file is about

$$S = ((8 + v)c + 2h + dcv) * 32\ Bit \tag{4.10}$$

This expression can be further simplified by assuming the default values used in the *SWE* implementation, according to table 4.2:

$$S = (17c + 2h) * 32\ Bit \tag{4.11}$$

Using the definition of h (see table 4.2) and writing the number of cells as the product of the number of patches as p, and the number of cells in a patch as r, the equation becomes

$$S = (17pr + nextprime(2p)) * 32\ Bit \tag{4.12}$$

Thus, the disk space for one step grows linearly with the amount of patches and the amount of cells in a patch. Because the maximum amount p of patches is calculated by

$$p = 2^k \tag{4.13}$$

with k being the refinement depth, the disk space grows exponentially in respect to the refinement depth.

XDMF Index

The accompanying XDMF file stores additional meta information as described in section 3.2. In addition to defining the vertex geometry and topology, and exposing the HDF5 cell attributes (see table 4.1), it adds several more attributes as according to table 4.3. This allows for continuation of the simulation, using the stored data to reconstruct the domain and simulation state.

4.4 Reconstructing Domain Data

The input module parses the XML file and reads the metadata. It reinitializes the application with the previous command line arguments stored in the XML file.

Dataset path	Name	Data type	Width	Length	Notes
`/step/g`	Geometry	Float	d	c * v	Vertex coordinates
`/step/p`	Topology	Integer	v	c	Vertex indices
`/step/t`	Refinement	Integer	2	h	Checkpoint data
`/step/a/b`	Bathymetry	Float	1	c	
`/step/a/d`	Depth	Integer	1	c	Refinement level
`/step/a/f`	Water Momentum	Float	2	c	
`/step/a/h`	Water Height	Float	1	c	
`/step/a/k`	Water Level	Float	1	c	
`/step/a/l`	Plotter Type	Integer	1	c	Cell triangle type
`/step/a/o`	MPI Rank	Integer	1	c	

Table 4.1: Layout of a HDF5 output file.

Symbol	Explanation
v	Number of vertices per element. Set to 3 vertices for triangles in SWE.
d	Number of spatial dimensions per vertex. Set to 2 dimension for a plane in SWE.
c	Number of cells in the domain. This includes sub-patch cells.
h	Size of the topology hash table. Equal to $h > 2 * p$ where h is the smallest fitting prime and p the number of patches in the domain.

Table 4.2: Explanation of the symbols used in table 4.1

Name	Context	Notes
CommandLine	Global	The command line string the application was invoked with
FileStamp	Global	The file name of the output data
Time	Step local	The current simulation time of this step
Step	Step local	The number of this simulation step
DeltaTime	Step local	The current time step

Table 4.3: Additional attributes defined in the XDMF output file.

It also sets required runtime meta data, such as the current step number and elapsed simulation time.

4.4.1 Cell Adaptation

The cell adaption traversal checks whether the current cell exists in the checkpoint database. This is done by first hashing the cell as described in section 4.2.1, and then looking it up in the previously mentioned hash table. If the cell is found, it is marked as completed and subsequently populated with simulation-related data, such as water height or bathymetry information.

If the cell is not found, it is refined by splitting it into two halves. This recursive procedure runs until each cell is marked as complete.

This traversal is able to be run in parallel, because each cell can be evaluated independently and the HDF5 dataset allows for concurrent read access.

4.5 Visualization and Verification

4.5.1 The ParaView Software

The *ParaView* application, developed and supported by *KitWare, Inc.* [2], provides a highly flexible system for scientific data visualization and analysis. Conveniently, the XDMF format is supported out of the box, without the need of any additional software packets.

4.5.2 The HDFView Utility

The *HDFView* application is being developed and distributed by the *HDF5 Group* [12]. It provides a graphical interface to display and manipulate the internal structure of HDF5 files.

5 Parallelization

5.1 The Message Passing Interface (MPI)

The *Message Passing Interface*, *MPI* for short, is a software library used to communicate between nodes in cluster. It is standardized by the *MPI Forum* [6] and typically implemented by the compiler manufacturer.

5.1.1 Collective and Individual I/O

MPI-I/O is a subsystem of the MPI library, providing general and parallel access to the file system. It is capable of exposing special features of the underlying file system, such as true concurrency supported by various file systems used on compute clusters.

There are two modes of operation supported by MPI-I/O when accessing a file, *independent* and *collective* access. Using independent access, the MPI I/O system immediately performs the requested I/O operation.

In collective mode on the other hand, all participating MPI nodes are required to participate on the same I/O operation, which is then aggregated and run in one step by the MPI-I/O system. For example, small consecutive write operations may be joined into one large block written to the disk. This tremendously reduces the amount of I/O operations performed, but only contributes to performance if all nodes participate at roughly the same time. Otherwise, the nodes will have to wait for each other.

5.1.2 Parallel HDF (PHDF)

Parallel HDF5 is a version of the HDF5 library developed with MPI support in mind. It is specifically optimized to be run on distributed systems supporting concurrent file access via MPI-I/O.

This enables a distributed application running on multiple nodes to perform parallel read and *write* operations on the same file resource.

Figure 5.1 compares the performance of the two MPI-I/O output modes, using the same scenario as in section 6.2. Clearly, the benefit of grouping I/O operations does not compensate for the additional synchronization time needed in this use-case.

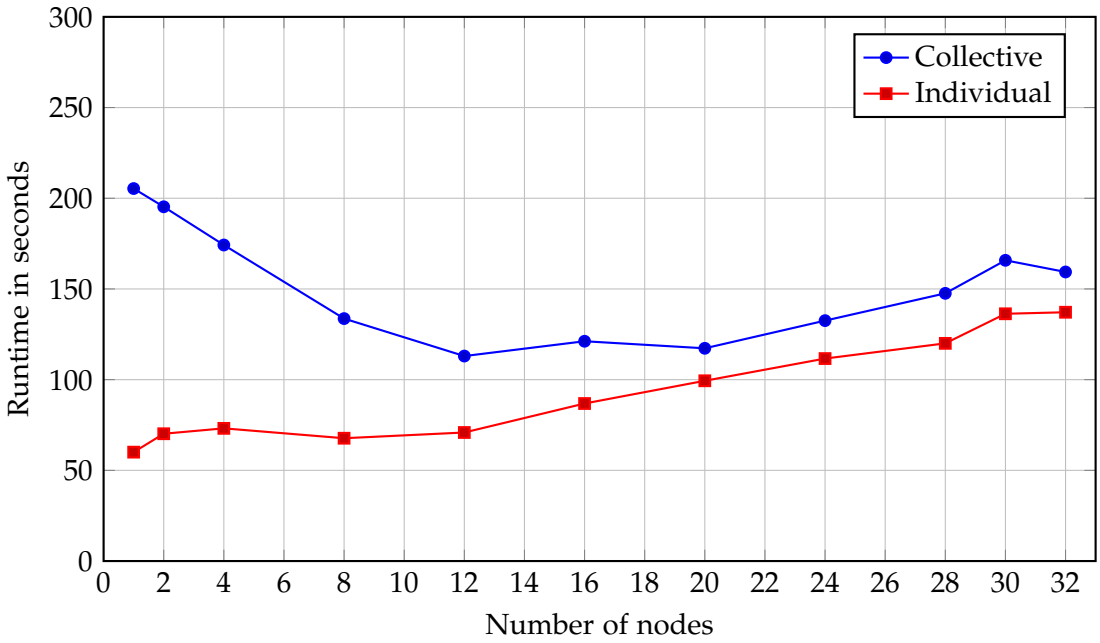


Figure 5.1: Runtime comparison of different MPI output modes.

5.2 Parallel Distributed Hash Tables

5.2.1 Motivation

The hash table described in section 4.2 has to be read- and writeable by all nodes, while still guaranteeing integrity. Ideally, this would happen in parallel. However, due to the nature of a hash table and its collision handling, a clear order of operations has to be establishes to guarantee data integrity.

If only a single node was used to store the hash table in memory, this node could run out of available memory. Also, because write and read requests to the hash table need to be atomic - to guarantee its integrity - and thereby synchronized using locking, parallel access performance would drop.

5.2.2 MPI Remote Memory Access (RMA)

MPI traditionally supports *synchronous* communication, i.e. blocking send and receive operations between two and multiple nodes, and *asynchronous* communication in form of *Remote Memory Access*, *RMA* for short.

MPI RMA allows for data modification on another node without the target note

actively having to execute any logic. This mechanism is referred to as *passive target synchronization*. Instead of explicitly participating in the data exchange, the target node marks a section of memory (a *window*) as public, and the MPI library handles all read and write operations concerning this window [15].

To combat the issues concerning the implementation using only a single node, the hash table is segmented into as many parts of the same size as there are nodes, and then those segments are distributed across the pool of nodes. Each node can then access the full hash table by computing which node stores the requested segment, and then accessing it using atomic MPI RMA operations. Although this still requires exclusively locking the target node, this leads to a higher degree of concurrency due to parallel inter-node communication, while still guaranteeing integrity.

5.3 Limitations

Unfortunately, at the time of writing, parallel HDF5 only allows for node-level concurrency, but does not support multithreaded access. Thus, all HDF5 related operations have to be performed in a single thread on each node.

To still enable parallel output data preparation across threads, all threads write into a shared memory buffer, which is then written out to the disk when all threads are finished.

6 Results

All tests were performed on the LRZ *CoolMUC-2* cluster, consisting of *Intel Xeon E5-2697 v3 ("Haswell")* CPUs, with 28 cores per node and 64 GB of RAM per node [3]. For every simulation step, an output step was written.

6.1 File System Usage and Strain

One of the main goals of the XDMF-based output format was to reduce the number of output files which were generated, compared to the previously existing XML writer. Because the HDF5 format allows for parallel access via MPI, the information in these files was consolidated into a single file. Figure 6.1 compares the amount of files for a hypothetical test case.

Figure 6.2 compares the total amount of disk scape required for the same hypothetical test case.

6.1.1 Interpretation

Because the XML writer generates a file for each *output step*, each *MPI node*, and each *section* on every node, the amount of files generated grows exponentially. The XDMF writer on the other hand only uses a constant amount of files, configurable by the user. This tremendously reduces the stress on the file system, as those systems are typically built to handle about 2^{16} to 2^{32} files.

The total disk space used by the XDMF writer is less than that used by the XML writer. This is mainly due to HDF5 being a binary data format, and thus having a higher data density than the text-based XML format. Also, the total disk space used by the XML writer is, just as the amount of files, independent of node count.

6.2 Strong Scaling

For the purpose of testing, the *Radial dam break* scenario of the *SWE* module was selected. Dynamic cell adaption was disabled to ensure reproduceable and uniform results.

Figure 6.3 shows a rendering of this scenario, simulating 2000 seconds. Subsection (a) displays the bathymetry, black denoting deep (10 meters below sea level) and gray

shallow (5 meters below sea level) shores. In this case, a circular pit exists in the center of the domain. Subsections (b) to (f) show the progress of a 10 meter high water column with a smaller diameter than the pit slumping down and creating waves on the pit borders. Red denotes a high water column height, and blue a lower one.

Strong scaling evaluates how the program runtime behaves when a fixed problem of size S is computed by a varying amount N of nodes. Thus, each node has to compute a sub-problem of size

$$S_n = \frac{S}{N} \tag{6.1}$$

A perfectly scaling program would complete after

$$T_n = \frac{N}{T_1} \tag{6.2}$$

seconds, with T_1 being the program runtime on a single node.

6.2.1 Runtime

In the test case plotted in figure 6.4, a domain containing

$$S = 2^{20} = 1048576 \tag{6.3}$$

cells was simulated for 10 steps. Figure 6.4 shows the total runtime in seconds for the pre-existing XML writer, the new XDMF-based writer. Figure 6.5 shows a configuration without any output enabled. The XDMF writer was tested in two configurations, with and without emitting checkpoint data required for restoring the simulation in a potential subsequent run.

Interpretation

The diagram shows a considerable performance loss for all writers in comparison to the configuration without any output enabled (see figure 6.5). This is expected, because the required I/O operations to store the generated data on the disk are expensive, compared to the in-memory calculations making up the rest of the program.

The XML writer experiences a performance penalty for each additional node, which for this test case is greater than the performance gain achieved by added computational power. This leads to a total runtime *increase* as the number of participating nodes increases. Even though the XML writer is able to write simultaneously to the disk from every node by writing to multiple files, the sheer amount of files and the accompanying file system strain dampens the theoretical performance gain noticeably.

The XDMF writer does not suffer from the same performance penalties as the XML writer, because it writes only to a single HDF5 file. MPI-I/O enables all participating nodes to write to the file *at the same time*, reducing the need for operation sequencing due to file system locking. As figure 6.4 shows, the runtime of the XDMF writer stays more or less constant. This demonstrates that the performance loss due to a more complex disk I/O with an increasing amount of nodes can be compensated by the performance gain in other sections of the program.

6.2.2 Parallel Efficiency

The parallel efficiency can be computed by comparing the theoretical linear scaling performance to the actual scaling performance. Thus, the parallel efficiency is given by

$$E_i = \frac{T_1}{N * T_i} \tag{6.4}$$

A program with ideal scaling (i.e. linear) would have a parallel efficiency of 1. Figure 6.6 shows the scaling efficiency, similar to the graph in section 6.2.

Interpretation

While the program exhibits an efficiency of around 70% - 90% without any output enabled (see figure 6.7), the performance degrades when using any output writer. The XDMF writer manages to outperform the XML writer by about one order of magnitude.

6.2.3 Speedup Ratio

The speedup ratio is a metric for how much faster the program becomes when more nodes are added and is computed by

$$R_i = \frac{T_1}{T_i} \tag{6.5}$$

Figure 6.8 shows the speedup ratio, similar to the graph in section 6.2 and section 6.2.2. This metric further supports the conclusions made in the previous sections.

6.3 Weak Scaling

Weak scaling evaluates the program runtime while giving each node a sub-problem of the same size to compute. Thus, the total problem size is directly dependent on the number of participating nodes.

6.3.1 Runtime

In the test shown in figure 6.10 and figure 6.12, a simulation containing

$$S = 2^{10+k}, k = [0; 5] \tag{6.6}$$

cells was computed by $N = 2^k$ nodes. This simulation used the same *dam break* scenario as the previous test (see section 6.2). Each node had a sub-problem containing

$$S_n = \frac{2^{10+k}}{2^k} = 2^{10} \tag{6.7}$$

cells to compute. Figure 6.10 compares the runtime of different output modes, figure 6.11 serves as a comparison where no output writer is enabled.

Interpretation

Similar to the strong scaling tests performed in section 6.2, the XDMF writer generally displays a similar scaling behavior to the XML writer. The XDMF writer manages to perform consistently better.

6.3.2 Parallel Efficiency

Analogously to section 6.2.2, the parallel efficiency for weak scaling can be computed by comparing the theoretical linear scaling performance to the actual scaling performance. Thus, the parallel efficiency for weak scaling is given by

$$E_i = \frac{T_1}{T_i} \tag{6.8}$$

Just as before, a program with ideal scaling (i.e. linear) would have a parallel efficiency of 1.

Interpretation

As can be seen in figure 6.12, the XDMF writer exhibits a similar scaling behavior as the XDMF writer. However, it manages to scale consistently better. Figure 6.13 serves as a comparison to a configuration without any output writers active.

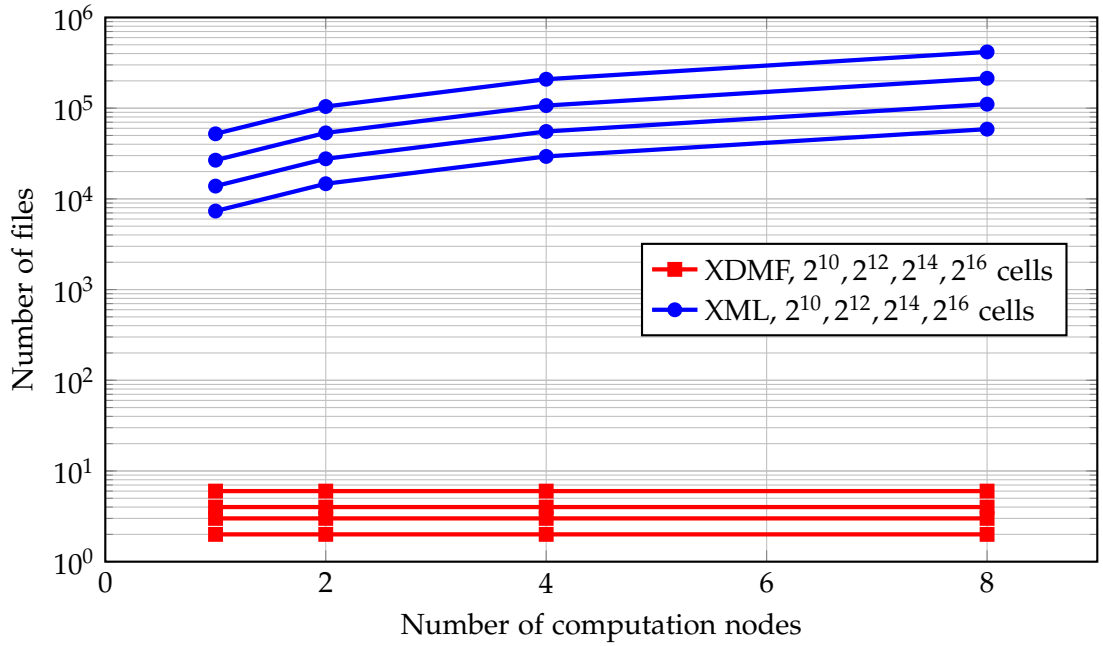


Figure 6.1: Number of files generated by different writers

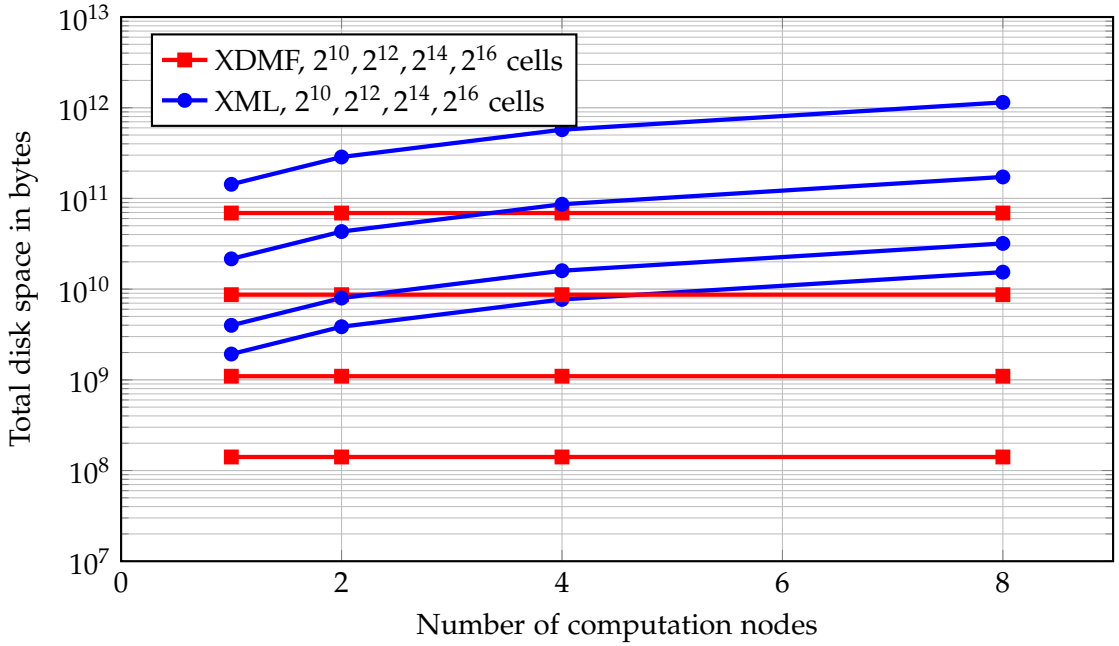
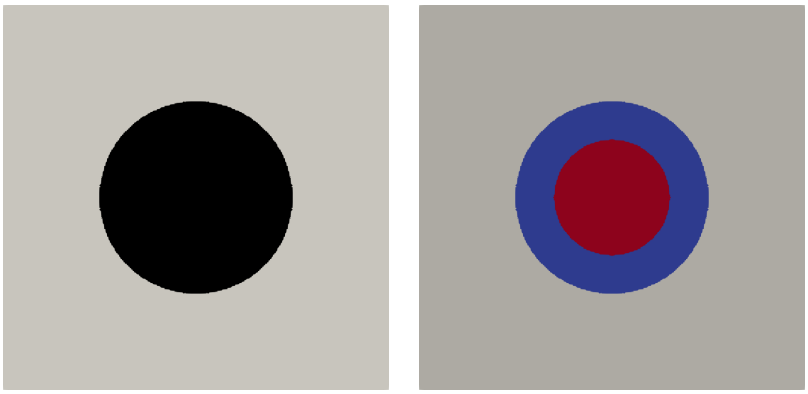
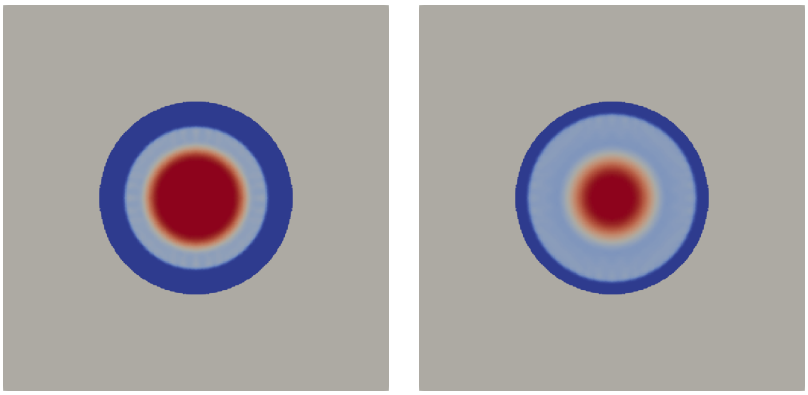


Figure 6.2: Total disk space used by different writers



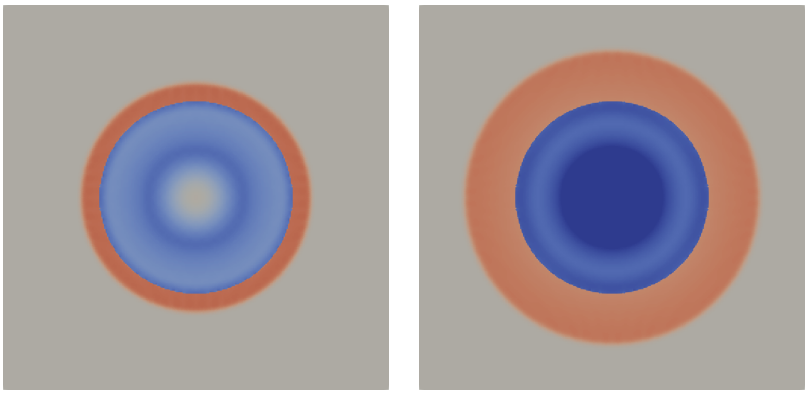
(a) Bathymetry

(b) Water Height, t = 0 s



(c) Water Height, t = 300 s

(d) Water Height, t = 600 s



(e) Water Height, t = 1300 s

(f) Water Height, t = 2000 s

Figure 6.3: Radial dam break scenario

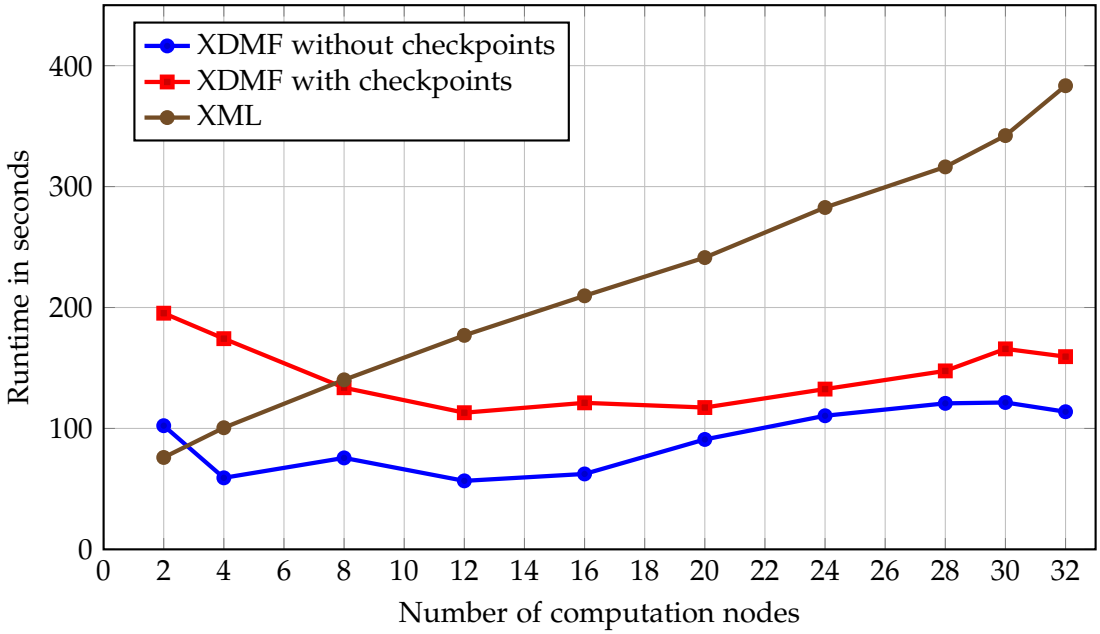


Figure 6.4: Strong scaling: Runtime of different writers

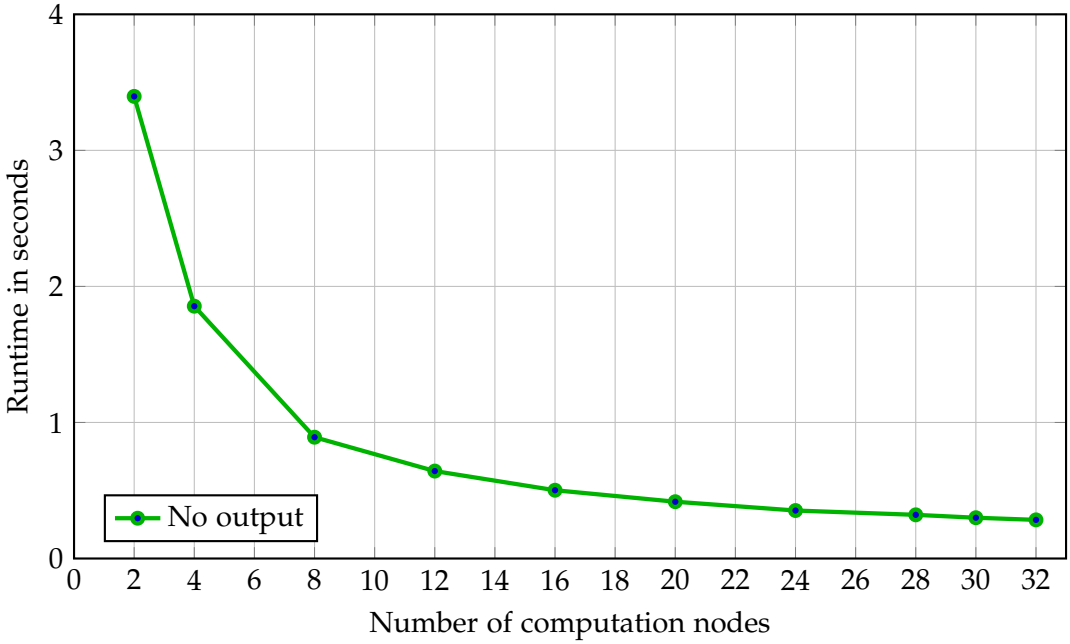


Figure 6.5: Strong scaling: Runtime without any writer

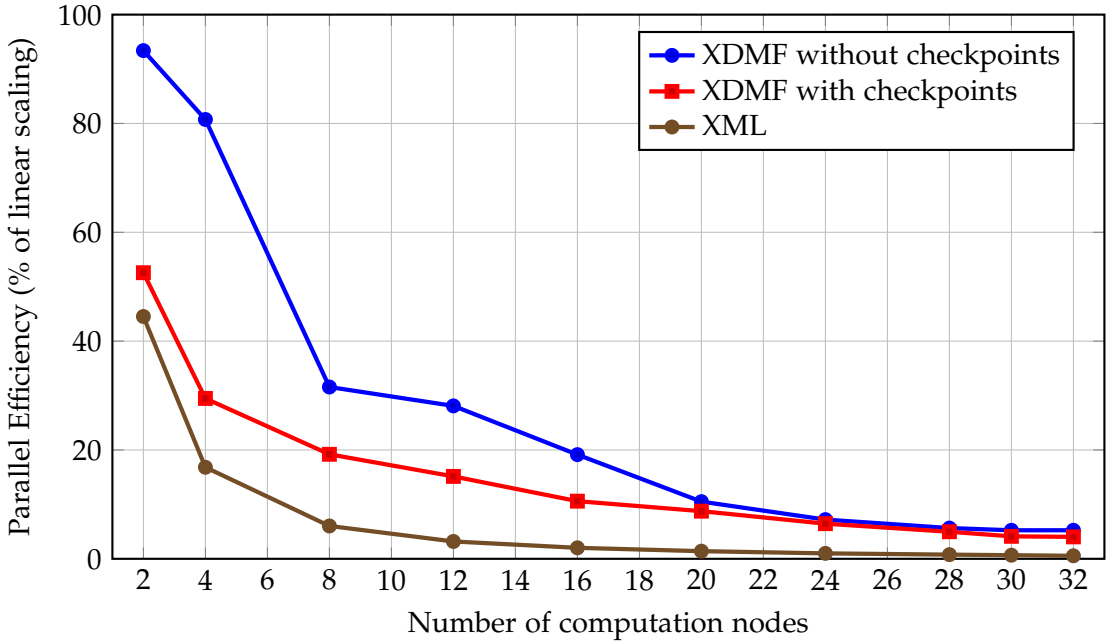


Figure 6.6: Strong scaling: Parallel efficiency of different writers

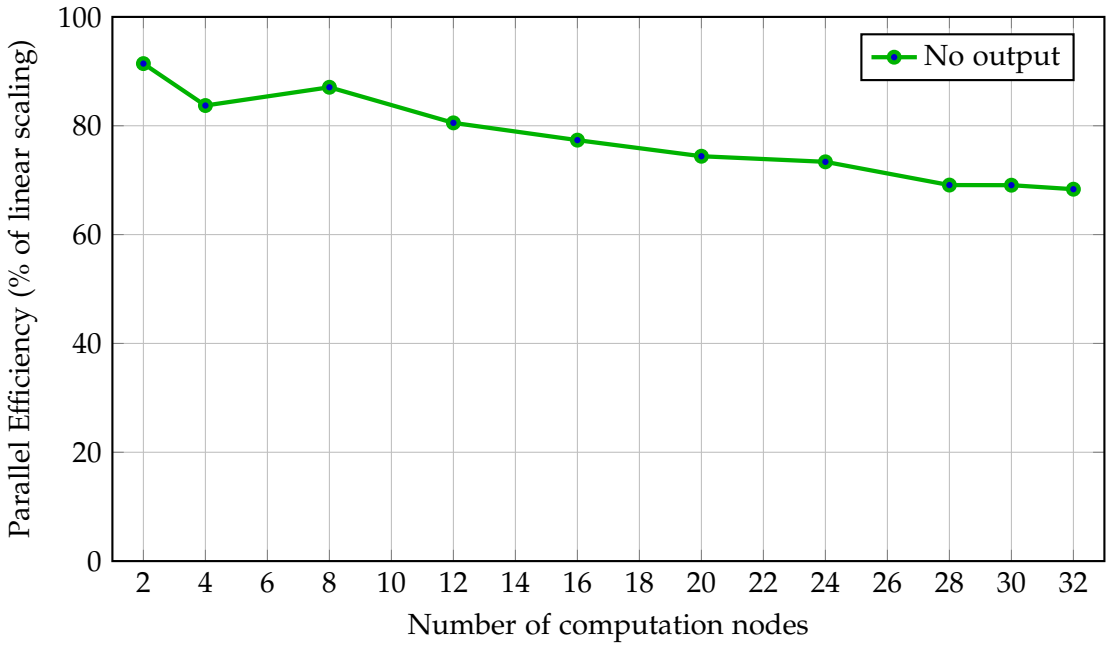


Figure 6.7: Strong scaling: Parallel efficiency without any writer

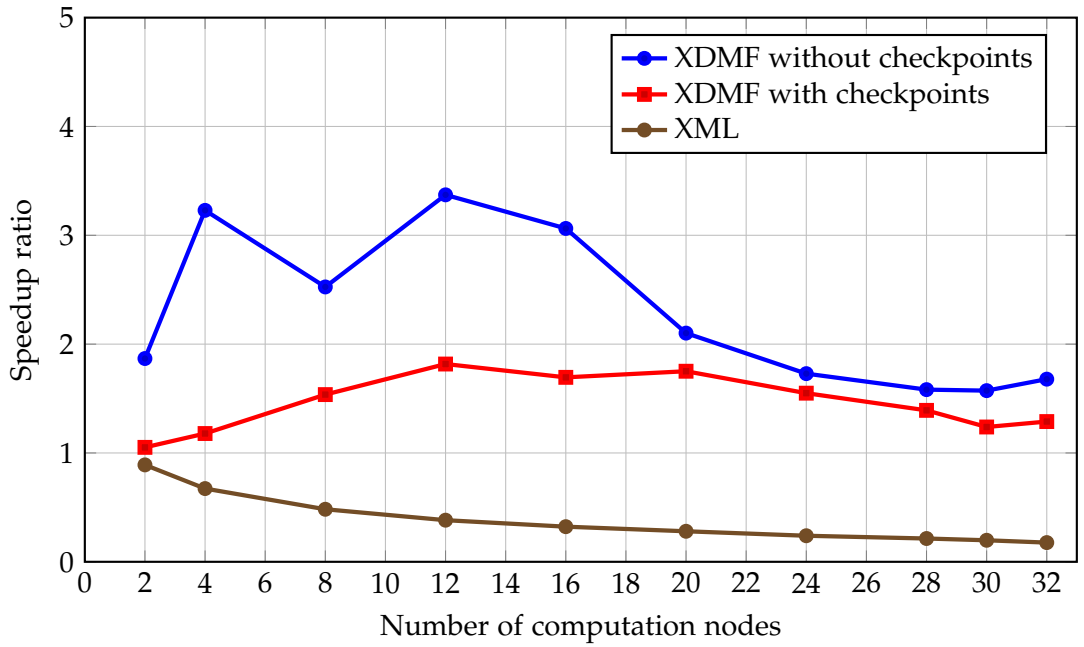


Figure 6.8: Strong scaling: Speedup ratio of different writers

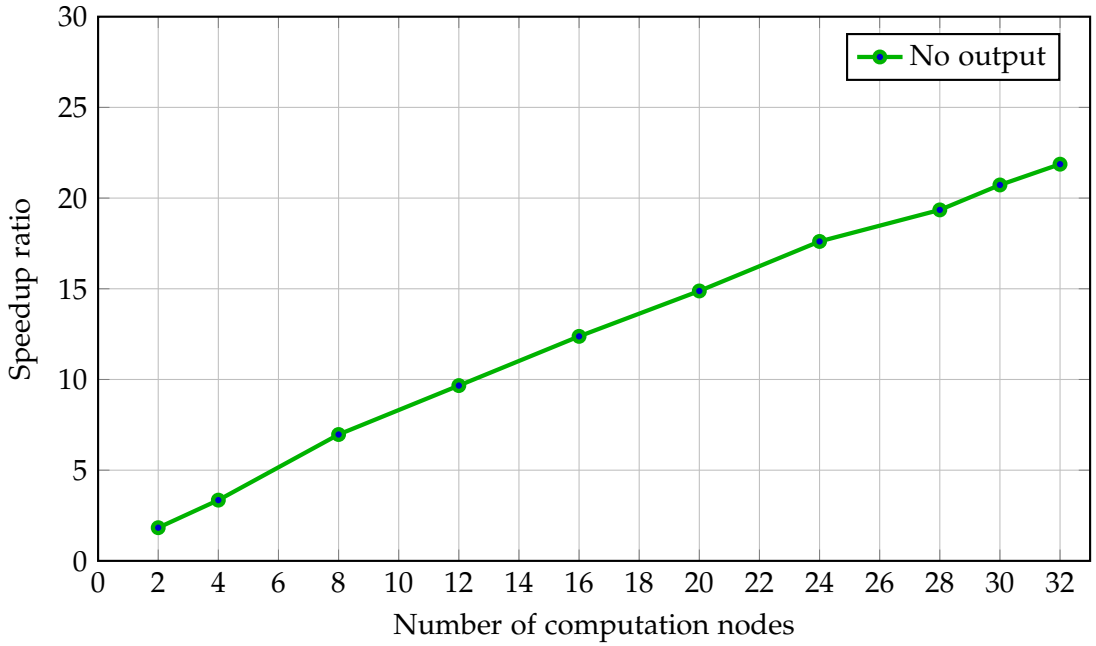


Figure 6.9: Strong scaling: Speedup ratio without any writer

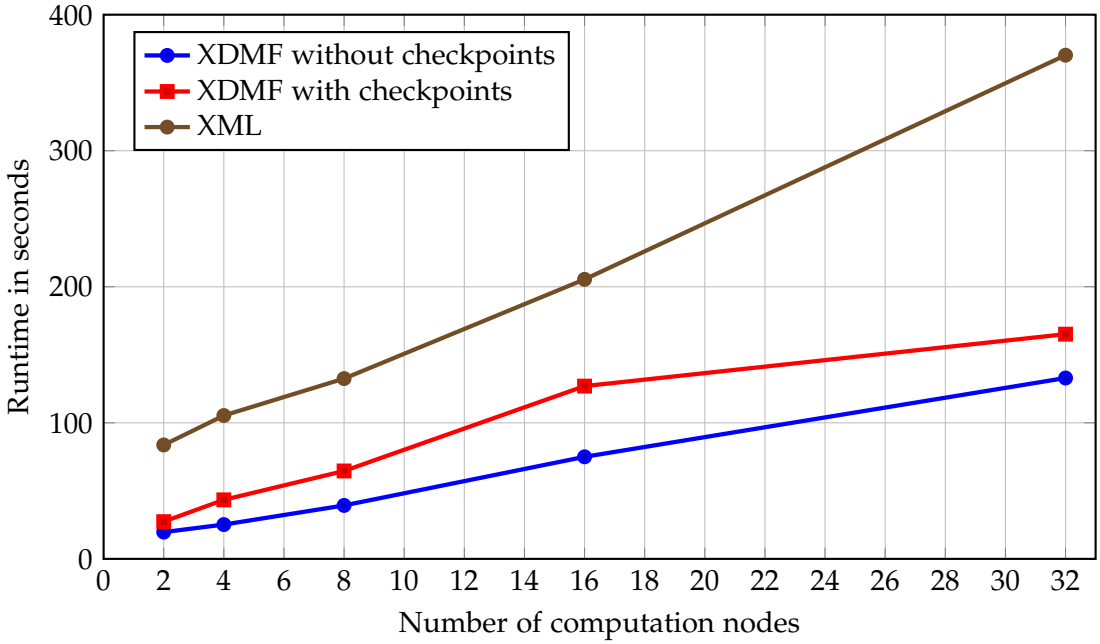


Figure 6.10: Weak scaling: Runtime of different writers

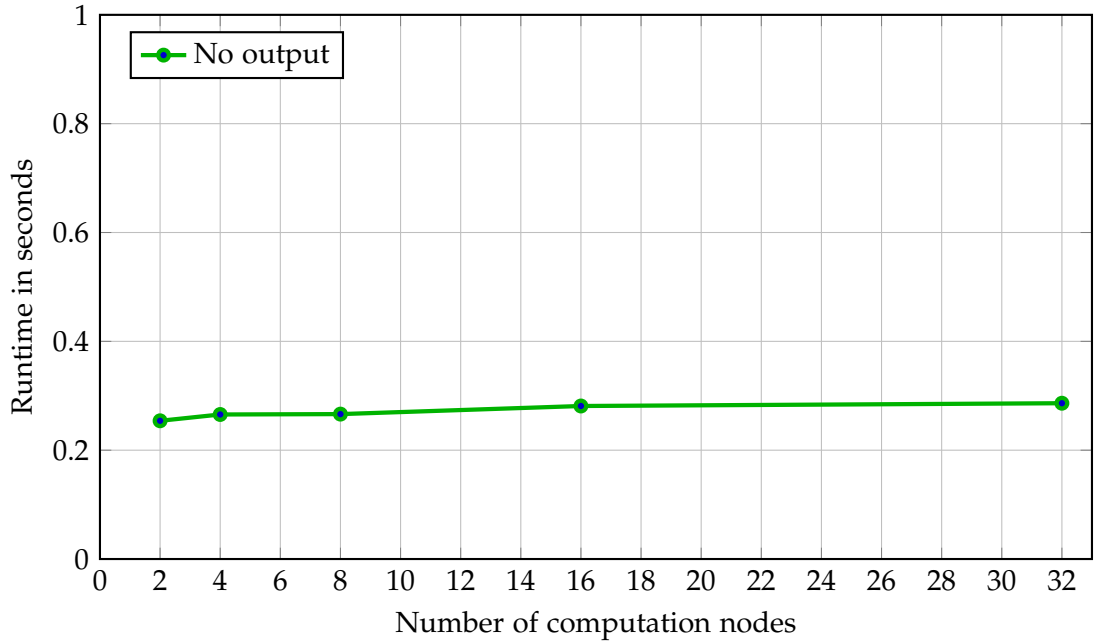


Figure 6.11: Weak scaling: Runtime without any writer

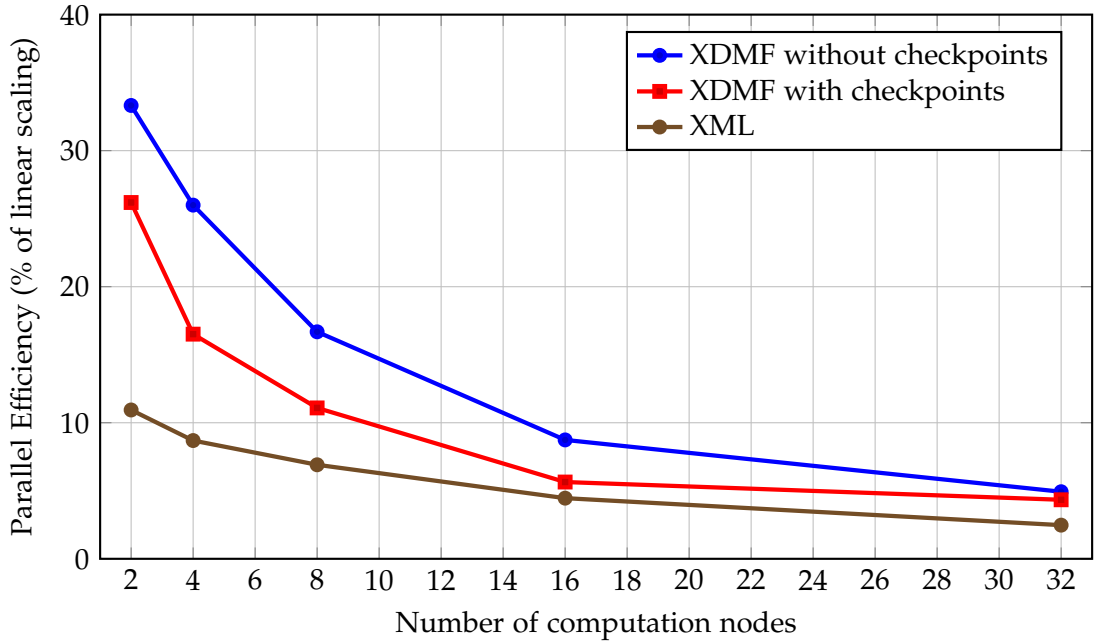


Figure 6.12: Weak scaling: Parallel efficiency of different writers

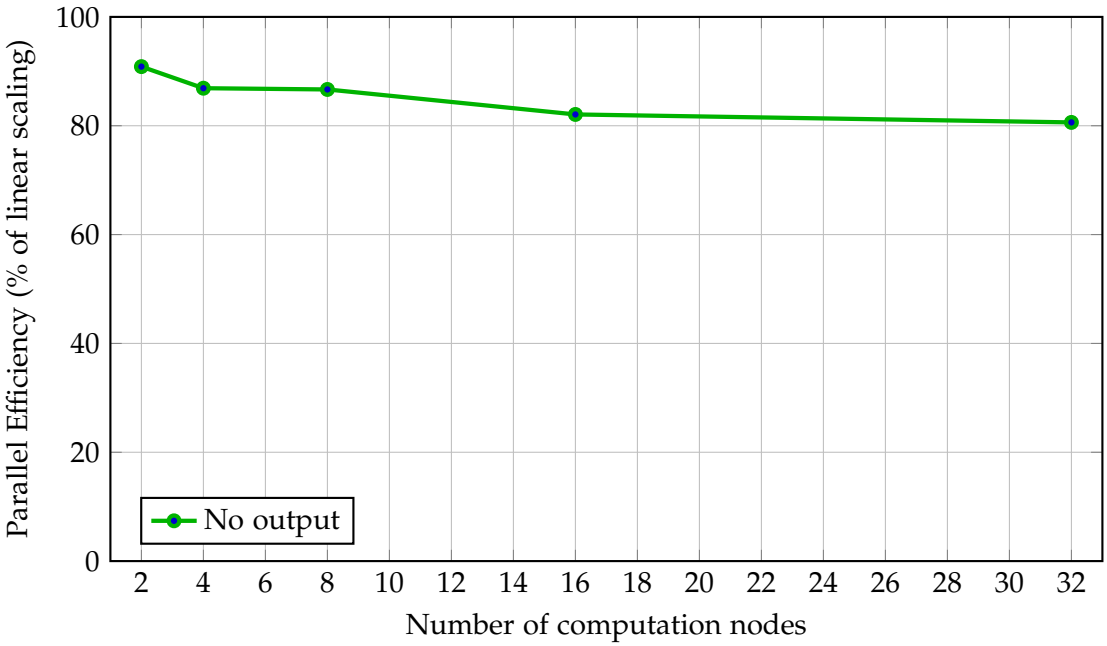


Figure 6.13: Weak scaling: Parallel efficiency without any writers

7 Implementation Notes

The complete source code can be retrieved from https://gitlab.lrz.de/samoa/samoa.

7.1 Command Line Parameters

The XDMF implementation contributes the following new command line parameters:

- `-xdmfoutput` Enables the XDMF output writer. *Disabled* by default.
  - `-xdmfspf 100` Sets the amount of steps per HDF5 file. `100` by default.
  - `-xdmfcpint 1` Sets the checkpoint per output step interval. `1` by default. Set to `0` to disable checkpointing.
- `-xdmfinput filename.xmf` Restores the simulation from the file `filename.xmf`. *Disabled* by default.

7.2 Code Structure

Although the community supporting the *XDMF* format provides an official API for dealing with the format, a custom implementation was established due to the inflexibility of the official API in regards to the underlying HDF5 structures.

The provided code is split into two main parts, the base library functions and a reference implementation for the *SWE* (see section 1.1.2) module. For both the input and output traversal subroutines the base library provides a generic implementation, which is wrapped and augmented by a module-specific implementation. Correspondingly, the traversal data type is also wrapped and extended.

The base library consists of the following files residing in `src/XDMF`:

`XDMF_compilation_control.f90`	Compiler definitions and macros
`XDMF_config.f90`	Extensions to the configuration and CLI reader
`XDMF_data_types.f90`	Common data types used by the library
`XDMF_hdf5.f90`	Helper functions for handling HDF5 files
`XDMF_math.f90`	Handles prime numbers and hashing

Modules handling the output are found in `src/XDMF/output`:

`XDMF_output_base_data_types.f90`	Base type of the output traversal data structure
`XDMF_output_base.f90`	Base implementation of the output traversal
`XDMF_xmf.f90`	Helper functions for generating XMF files

Similarly, modules handling the input are found in `src/XDMF/input`:

`XDMF_fox.f90`	Namespace for the FoX library
`XDMF_initialize_base_data_types.f90`	Base type of the input traversal data structure
`XDMF_initialize_base.f90`	Base implementation of the input traversal

The reference implementation of the XDMF writer and reader in the *SWE* module consists of these additional files in `src/SWE`:

`SWE_XDMF_adapt.f90`	Empty traversal needed by the framework
`SWE_XDMF_initialize.f90`	Specific implementation of the input traversal
`SWE_XDMF_output.f90`	Specific implementation of the output traversal

Multiple pre-existing files have been modified to integrate the new system. Notably, `Config.f90` and `SWE/SWE.f90`.

7.3 Compilation

For successful compilation, the HDF5 libraries, FoX libraries and compiler scripts have to be properly configured and installed.

7.3.1 SCons

SCons, the build system used by the sam(oa)^2 software, was extended by a few command line options regarding the *optional* compilation of the XDMF system:

`xdmf`	Boolean	Enables XDMF support, *disabled* by default
`xdmf_fox_dir`	Path string	FoX library directory
`xdmf_hdf5_dir`	Path string	HDF5 library directory

7.3.2 HDF5

The HDF5 library is required to be installed at version `1.10.4` or later. All code using the HDF5 libraries has to be compiled using the `h5fc` and `h5pfc` wrapper scripts.

It is recommended to compile the library from source, because some nonstandard configuration parameters are required:

`-enable-parallel`	Enables MPI support
`-enable-threadsafe`	Enables HDF5 to run in a multithreaded process
`-enable-fortran`	Creates Fortran bindings
`-enable-unsupported`	Allows nonstandard configuration
`-enable-optimization=high`	Enables compiler optimization

7.3.3 FoX

The *Fortran Library for XML* [1] (*FoX* for short) library has to be linked either statically or dynamically to the program. Support for the `DOM` module has to be enabled at compile-time.

`-DFoX_ENABLE_DOM=1`	Enables DOM support

7.3.4 Installation scripts

Several shell-scripts are provided in `scripts/XDMF`, simplifying the process of downloading, configuring and installing the required libraries. The scripts `install_all_dev.sh` and `install_all_hpc.sh` may be invoked with an optional destination directory as the first command line argument, otherwise they will install the required libraries into `/opt/samoa_xdmf_libs` and `~/local`, respectively.

The development variant will attempt to compile the libraries using the *GNU Compiler Collection* without any MPI support, while the HPC variant will attempt to use the *Intel Compiler* with support for MPI.

However, the scripts are easily adaptable to use other compilers or MPI variants.

8 Future Work

8.1 Extension to 64 Bit

In order to process more than 2^{31} cells and approximately 29 levels of refinement, The XDMF output module would have to use 64 Bit counters and offsets internally, as well as in the handling of HDF5 files. Although the hash computation itself (see section 4.2.1) already operates purely on 64 Bit integers, other variables would need to be expanded to provide full 64 bit support.

The HDF5 library already supports data offsets in 64 bit, but the record data types would have to be upgraded. This of course would entail double memory and disk space usage.

8.2 Adaption to Other Cell Types

The original SWE module is not the only use case for the *sam(oa)*^{2} framework, other modules include *ADER-DG* and *FLASH*, which both take a different approach to approximating inner-cell geometry.

Adaptation of the XDMF writer and reader should be possible without many issues, as the substantial core part is independent of any specific module (see section 7.2).

8.3 Coarse Output

Coarse output would condense multiple neighboring cells into one, but just for the output. This would entail a reduced disk space usage and higher output performance, albeit less precision.

This would allow to quickly and visualize very large or dense domains.

List of Figures

List of Tables

Listings

Bibliography

[1] Andrew Walker. *The FoX Library*. 2019. URL: http://homepages.see.leeds.ac.uk/~earawa/FoX/DoX/ (visited on 01/29/2019).

[2] Kitware, Inc. *ParaView*. 2019. URL: https://www.paraview.org/ (visited on 01/24/2019).

[3] Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften. *The Linux Cluster CoolMUC-2*. 2019. URL: https://www.lrz.de/services/compute/linux-cluster/ (visited on 01/21/2019).

[4] Randall J. LeVeque, David L. George, and Marsha J. Berger. "Tsunami modelling with adaptively refined finite volume methods." In: *Acta Numerica* 20 (2016), 211–289.

[5] Oliver Meister. "Sierpinski Curves for Parallel Adaptive Mesh Refinement in Finite Element and Finite Volume Methods." In: *Technical University of Munich, Department of Informatics* (2016).

[6] MPI Forum. *MPI - Message Passing Interface*. 2019. URL: https://www.mpi-forum.org/ (visited on 01/21/2019).

[7] MPICH Community. *MPICH API Documentation*. 2019. URL: http://www.mpich.org/static/docs/latest/ (visited on 01/21/2019).

[8] OpenMP. *Open Multi-Processing*. 2019. URL: https://www.openmp.org/ (visited on 01/30/2019).

[9] David A. Randall. "The Shallow Water Equations." In: *Colorado State University, Department of Atmospheric Science* (2006).

[10] The HDF Group. *About the HDF Group*. 2019. URL: https://www.hdfgroup.org/about-us/ (visited on 01/29/2019).

[11] The HDF Group. *The HDF5 API*. 2019. URL: https://support.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html (visited on 01/29/2019).

[12] The HDF Group. *The HDF5 Library and File Format*. 2019. URL: https://www.hdfgroup.org/solutions/hdf5/ (visited on 01/21/2019).

[13] University of Illinois Urbana-Champaign. *National Center for Supercomputing Applications*. 2019. URL: https://www.ncsa.illinois.edu/ (visited on 01/29/2019).

[14] University of Illinois Urbana-Champaign. *University of Illinois Urbana-Champaign*. 2019. URL: https://illinois.edu/ (visited on 01/29/2019).

[15] Victor Eijkhout with Robert van de Geijn and Edmond Chow. *Introduction to High Performance Scientific Computing, MPI Topic*. 2011. URL: http://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-onesided.html (visited on 01/21/2019).

[16] XDMF Community. *XDMF - eXtensible Data Model and Format*. 2019. URL: http://xdmf.org/index.php/Main_Page (visited on 01/21/2019).

[17] XDMF Community. *XDMF API*. 2019. URL: http://xdmf.org/index.php/XDMF_API (visited on 01/21/2019).

[18] XDMF Community. *XDMF Model and Format*. 2019. URL: http://xdmf.org/index.php/XDMF_Model_and_Format (visited on 01/21/2019).