H2020 FETHPC-1-2014

An Exascale Programming, Multi-objective Optimisation and Resilience Management Environment Based on Nested Recursive Parallelism

Project Number 671603

D2.6 – AllScale API Specification (b)

WP2: Requirements and overall system architecture design

Version: 1.2
Author(s): Philipp Gschwandtner (UIBK), Herbert Jordan (UIBK)
Date: 27/10/17

Due date:	PM25
Submission date:	31/10/2017
Project start date:	01/10/2015
Project duration:	36 months
Deliverable lead organization	UIBK
Version:	1.2
Status	Final
Author(s):	Philipp Gschwandtner (UIBK) Herbert Jordan (UIBK)
Reviewer(s)	Kiril Dichev (QUB), Arne Hendricks (FAU)

Dissemination level	
PU	*PU - Public*

Disclaimer

This deliverable has been prepared by the responsible Work Package of the Project in accordance with the Consortium Agreement and the Grant Agreement Nr 671603. It solely reflects the opinion of the parties to such agreements on a collective basis in the context of the Project and to the extent foreseen in such agreements.

Acknowledgements

The work presented in this document has been conducted in the context of the EU Horizon 2020. AllScale is a 36-month project that started on October 1st, 2015 and is funded by the European Commission.

The partners in the project are UNIVERSITÄT INNSBRUCK (UBIK), FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN NÜRNBERG (FAU), THE QUEEN'S UNIVERSITY OF BELFAST (QUB), KUNGLIGA TEKNISKA HÖGSKOLAN (KTH), NUMERICAL MECHANICS APPLICATIONS INTERNATIONAL SA (NUMEXA), IBM IRELAND LIMITED (IBM).

The content of this document is the result of extensive discussions within the AllScale Consortium as a whole.

More information

Public AllScale reports and other information pertaining to the project are available through the AllScale public Web site under http://www.allscale.eu.

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	07/02/17	Final version (pre-review)	Philipp Gschwandtner, Herbert Jordan
1.0	27/02/17	Final version (post-review)	Philipp Gschwandtner, Herbert Jordan, Kiril Dichev, Arne Hendricks
1.1	01/10/17	(b) update for M25	Philipp Gschwandtner, Herbert Jordan
1.2	27/10/17	Final version of (b) update (post-review)	Philipp Gschwandtner, Herbert Jordan, Kiril Dichev, Arne Hendricks

Table of Contents

1 Contents

Executive Summary

This document provides an overview of the specifications of the AllScale Core API and the AllScale User API. It is intended to serve as a reference for developers working in the AllScale environment. All information given here represents the current state of work-in-progress within AllScale, and may change in the future.
The AllScale API provides a façade of the AllScale environment towards end-user applications. It aims at offering a minimal set of primitives to express parallelism, data dependencies, and synchronization required within application codes, while shielding the end-user from implementation details.
To achieve these goals, the API is split into two layers: The Core API and the User API. The former provides a concise set of basic generic primitives (parallel control flow, synchronization, communication). The User API builds on top of these primitives to offer specialized primitives for specific use cases (parallel loops, stencils, commonly used data structures such as grids, adaptive grids, and unstructured meshes).

Changes with respect to D2.5 AllScale API Specification (a)

Since the first version of this deliverable, D2.5 (a), the ongoing development of the system has caused a number of adaptations and extensions to the AllScale API. These changes, already incorporated in this version, D2.6 (b), are listed hereafter for clarity:

- In the Core API:
  - Clarification of the semantic of the treeture primitives
  - Extension of the data item concept by a shared data component
  - Additional operations in the IO interface:
    - Atomic transactions on input/output streams
  - The specification of the serialization infrastructure

- Additions to the User API:
  - Additional synchronization options for parallel loops
  - A parallel reduction operation (*preduce*)
  - a multiset implementation fulfilling the data item concept (*Bag*)

2 Introduction

This document provides a complete overview of the specifications of the AllScale Core API and the AllScale User API. It is intended to serve as a reference for developers working in the AllScale environment.
The specifications given here represent the current state of the work-in-progress within AllScale. It is not necessarily final or error-free, and might change in the future.
This document adheres to the terminology and formalism established in D2.3 “AllScale System Architecture” and hence employs abstract data types (ADTs) to gain the necessary level of abstraction required for conveying key information.

The presented signatures are to be adapted to C++'s idioms by deliverable D3.1 "API implementation for recursive parallelism".
Section 3 provides a short overview of both the Core API and the User API and their relationship. The Core API is covered in Section 4, while Section 5 details on the User API.

3 Overview

The AllScale API is the façade of the AllScale Environment towards end-user applications. It provides the necessary primitives to express parallelism, data dependencies, and synchronization steps required within application codes. The API is split into two layers:

- the AllScale Core API
- the AllScale User API

The Core API provides a concise set of basic generic primitives, comprising parallel control flow, synchronization, and communication constructs. The User API is harnessing the expressive power of the Core API to provide specialized primitives for particular use cases, including basic constructs like parallel loops as well as more sophisticated functionality offering efficient implementations of e.g. stencil operations.
The next two sections will describe the two APIs.

4 Core API

The Core API is already presented in D2.3 "AllScale System Architecture" and will only be briefly summarized here for completeness.

4.1 Prec

The AllScale Core API provides a single primitive for running concurrent tasks, the *prec* operator. It is a higher order function combining three input functions.

$$prec(\alpha \to bool, \alpha \to \beta, (\alpha, \alpha \to treeture\langle\beta\rangle) \to treeture\langle\beta\rangle) \to (\alpha \to treeture\langle\beta\rangle)$$

Parameter Type	Description
$\alpha \to bool$	Function testing for base case. If it returns true for a given data of type α, this data will be processed with the base case.
$\alpha \to \beta$	Function processing the base case.
$(\alpha, \alpha \to treeture\langle\beta\rangle) \to treeture\langle\beta\rangle$	Function processing the recursive step case.

4.2 Treetures

A *treeture* is a parameterized abstract data type that represents a handle on parallel tasks.

Treetures provide the following operations:

Signature	Description
$wait(treeture\langle\alpha\rangle) \rightarrow unit$	Waits for the referenced task to be completed.
$get(treeture\langle\alpha\rangle) \rightarrow \alpha$	Waits for the referenced task to be completed and returns the result.
$done(\alpha) \rightarrow treeture\langle\alpha\rangle$	References a finished task that produced the given value, i.e. convert a value to a treeture holding that value.
$par\begin{pmatrix} treeture\langle\alpha\rangle, \\ treeture\langle\beta\rangle, \\ (\alpha,\beta) \rightarrow \gamma \end{pmatrix} \rightarrow treeture\langle\gamma\rangle$	Creates a new task waiting for the result of the two given treetures and computing a new result using the given combination function; the subtasks are processed in parallel. The task referenced by the first treeture becomes the left child of the resulting task, the task referenced by the second parameter the right child;
$seq\begin{pmatrix} treeture\langle\alpha\rangle, \\ treeture\langle\beta\rangle, \\ (\alpha,\beta) \rightarrow \gamma \end{pmatrix} \rightarrow treeture\langle\gamma\rangle$	Creates a new task waiting for the result of the two given treetures and computing a new result using the given combination function; the subtasks are processed in sequence. The task referenced by the first treeture becomes the left child of the resulting task, the task referenced by the second parameter the right child;

4.3 Task References

A *task reference* is an abstract data type that can reference a subtask of a treeture, thereby enabling the definition of fine-grained dependencies.

Task references provide the following operations:

Signature	Description
$toRef(treeture\langle\alpha\rangle) \rightarrow tref$	Converts a treeture to a task reference.
$getLeft(tref) \rightarrow tref$	Obtains a reference to the (logical) left sub-task of the referenced task.
$getRight(tref) \rightarrow tref$	Obtains a reference to the (logical) right sub-task of the referenced task.
$wait(tref) \rightarrow unit$	Waits for the referenced subtask to be completed (blocking).

4.4 Data Item

Each data structure managed by AllScale has to provide four components:

- A *façade*, implementing the interface to the data structure as seen by the domain expert when utilizing it within an application

- A *region* type R, providing means to addressing parts of the data structure
- A *fragment* type F, implementation, capable of maintaining regions of the overall data structure within a single address space of a device.
- A *shared data* segment type S, containing read-only information shared among all fragments of a data item instance;

Since the façade implements the interface to the data structure as seen by the domain expert, its specification is dependent on the data structure (see Section 5.2). The specification of regions, fragments, and shared data segments is detailed in the following three sections.

4.4.1 Region

For a region type R the following operations need to be provided:

Signature	Description
$union(R, R) \rightarrow R$	Computes the union of the specified regions.
$intersect(R, R) \rightarrow R$	Computes the intersection of the specified regions-
$empty(R) \rightarrow bool$	Tests whether the specified region is empty.
$pack(R) \rightarrow archive$	Packs the specified region to an archive for transfer to another address space.
$unpack(archive) \rightarrow R$	Unpacks the specified archive to a region.

4.4.2 Fragment

For a fragment type F, the following operations need to be provided:

Signature	Description
$create(S, R) \rightarrow F$	Creates a fragment using the given shared data information of type S covering at least the specified region.
$delete(F) \rightarrow unit$	Deletes the specified fragment.
$resize(F, R) \rightarrow unit$	Resizes the specified fragment to cover at least the specified region.
$mask(F) \rightarrow \alpha$	Provides access to the data stores within the specified fragment.
$extract(F, R) \rightarrow archive$	Extract the specified region of the specified fragment and pack it into an archive for transfer to another address space.
$insert(F, R, archive) \rightarrow unit$	Import the data within the specified archive into the specified region of the specified fragment.

4.4.3 Shared Data Segments

For a shared data type S, the following operations need to be provided:

Signature	Description
$extract(F) \rightarrow S$	Obtains the shared data associated to the given fragment.
$pack(S) \rightarrow archive$	Packs the specified shared data to an archive for transfer to another address space.
$unpack(archive) \rightarrow S$	Unpacks a shared data instance from a given archive.

Furthermore, for every constructor of the façade type, a constructor for the shared data segment accepting the same list of arguments has to be provided.

4.5 IO Primitives

AllScale provides two means for storage interaction:

- Stream-based, providing unordered input and output facilities
- Memory-mapped, providing efficient input-only facilities for random access within large data sets

4.5.1 Stream-based

Stream-based IO provides atomic means for sequentially reading and writing data entries from and to storage. The order of the entries in the stream is undefined. Furthermore, written entries are guaranteed to be visible in the file only when the application terminates.

Stream-based IO provides the following operations:

Signature	Description
$read(istream) \rightarrow \alpha$	Atomically reads an element of type α from the given input stream.
$atomic\left(\begin{matrix} istream \\ (istream) \rightarrow unit \end{matrix}\right) \rightarrow unit$	An operator providing atomic access to an input stream, enabling the provided function to read a sequence of consecutive elements in order;
$write(ostream, \alpha) \rightarrow unit$	Atomically writes the given element of type α to the given output stream, where it will be visible eventually.
$atomic\left(\begin{matrix} ostream \\ (ostream) \rightarrow unit \end{matrix}\right) \rightarrow unit$	An operator providing atomic access to an output stream, enabling the provided function to write a sequence of consecutive elements in order;
$create\_in(string) \rightarrow istream$	Opens an input file with the given name and provides a stream to read from it; the file format is implementation specific and data may only be read and written using the

	AllScale IO API.
$create\_out(string) \rightarrow ostream$	Creates a new empty file under the given name and provides an output stream to write information to the file; the file format is implementation specific and may only be read using AllScale IO primitives.
$get\_in(string) \rightarrow istream$	Obtains an input stream to a previously opened input file which might be concurrently read.
$get\_out(string) \rightarrow ostream$	Obtains an output stream of a previously opened output file which might be concurrently written to.

4.5.2 Memory-mapped

Memory-mapped IO provides the means for efficient random read-only access of sub-sets of larger data. Open files are available in the address spaces of all AllScale runtime system processes.

Memory-mapped IO provides the following operations:

Signature	Description
$open(string) \rightarrow mmfile$	Globally opens a memory mapped file with the given path.
$get(string) \rightarrow mmfile$	Obtains a reference to a previously opened memory mapped file.
$access(mmfile) \rightarrow \alpha$	Interprets the content of the memory mapped file as a value of type α.
$close(mmfile) \rightarrow unit$	Globally closes a memory mapped file such that it is no longer available for any process in the application.

4.6 Object Serialization

To facilitate the transfer of objects between address spaces, means for migrating instances are required, commonly referred to as a serialization infrastructure. The general idea is to have an intermediate (logical) archive format to which objects can be converted to as well as restored from.
Let *archive* be this archive type. An object of type T is serializable if the following two operations are provided:

Signature	Description
$pack(T) \rightarrow archive$	Packs the given object of type T to an archive for transfer to another address space.
$unpack(archive) \rightarrow T$	Unpacks an object of type T from a given archive.

Note that the same types and operators have been utilized in the context of data items. Consequently, regions and shared data segments are inherently serializable.

5 User API

5.1 Parallel Control Flow Constructs

5.1.1 Parallel Loops

A widely-used construct encountered in high performance applications are parallel loops. They provide the means to perform computational work in an iteration space in parallel at the cost of executing the individual iterations concurrently and in an arbitrary order. To that end, the User API offers a parallel loop construct for realizing data-parallel programming within the AllScale environment.

Requirements

The following features need to be covered:

- iterate over a given iterator range and apply a function (loop body) for elements in parallel
- support for fine-grained synchronization

Interface Specification

Let $iterator$ be of *RandomAccessIterator*[1] type of C++. As such, it needs to provide operators for moving them to point to any element in constant time. Then the *pfor* operator provides a parallel loop execution with the following parameters:

Name	Type	Description
begin	$iterator$	Points to the inclusive beginning of the range to be processed by the parallel loop
end	$iterator$	Points to the exclusive end of the range to be processed by the parallel loop
body	$(iterator) \rightarrow \beta$	The function to be applied to each iterator of the range $[begin, end)$
dependency	$dep\langle iterator \rangle$	Optional dependency to use for fine-grained synchronization

The *pfor* operator returns a loop reference $loop\_ref\langle iterator \rangle$ to be used for synchronization.

While offering full loop synchronization, the User API also includes two functions for establishing fine-grained dependencies to facilitate the removal of overly strict global synchronization:

[1] http://en.cppreference.com/w/cpp/concept/RandomAccessIterator

Name	Type	Description
toTreeture	$(loop\_ref\langle iterator\rangle) \rightarrow treeture\langle unit\rangle$	Provides a treeture for a given loop reference to synchronize on the entire loop
one_on_one	$(loop\_ref\langle iterator\rangle) \rightarrow dep\langle iterator\rangle$	Forms a one-on-one dependency where iteration i of a new parallel loop may be executed after iteration i of the given loop has been completed.
small_neighborhood_sync	$(loop\_ref\langle iterator\rangle) \rightarrow dep\langle iterator\rangle$	Forms a neighborhood dependency where iteration i of a new parallel loop may be executed after iterations $\{i + c \mid c \in \{-1,0,1\}^n \vee \lvert c\rvert \leq 1\}$ of the given loop have been completed; n is the number of dimensions of the iterator;
full_neighborhood_sync	$(loop\_ref\langle iterator\rangle) \rightarrow dep\langle iterator\rangle$	Forms a neighborhood dependency where iteration i of a new parallel loop may be executed after iterations $\{i + c \mid c \in \{-1,0,1\}^n\}$ of the given loop have been completed; n is the number of dimensions of the iterator;

5.1.2 Recursive space/time decomposition

A frequently utilized template for large-scale high-performance applications are *stencils*. In a stencil-based application, an update operation is iteratively applied to the elements of an n-dimensional array of *cells*. Thereby, for each update, the update operation is combining the previous values of cells within a locally confined area surrounding the targeted cell location to obtain the updated value for the targeted cell. Since these update operations within a single update step (also known as timestep) are independent, this application pattern provides a valuable source for parallelism within a correspondingly shaped application. Furthermore, many HPC codes can be reduced to this kind of parallel pattern.

Due to its importance, the AllScale User API provides a generic framework for stencil-like applications. The specification, as it is covered in this document, does not put constraints on the internal implementation of this framework, and yet defines its abstract interface – as it is required to be provided to potential end-users.

Requirements

The following features need to be covered:

- the support for operating on n-dimensional, regular data structures; in particular, this must include grids as specified by Section 5.2.1.
- support for freely programmable update functions (also known as kernels); among others, no upper limit on the spatial confinements of the update operation may be imposed
- support for the specification of functions handling boundary elements, which, due to their proximity to the edge of the underlying grid, cannot be processed utilizing the main update function
- support for initialization code to be applied to individual cells before starting the computation
- support for finalization code to be applied to individual cells at the end of the computation
- support for introducing observer functions, which retrieve samples of data for selected time steps and locations

Interface Specification

The stencil framework is – like the pfor operator – realized through a single, generic *stencil* operator to be parameterized to customize its activities. In its most generic form it has the following parameters:

Name	Type	Description
timesteps	int	the number of time steps to be computed
size	int^n	the spatial size $s_1 \times \ldots \times s_n$ of the array to be processed, where n is the number of dimensions
kernel function	$(int, int^n, \alpha^{s_1 \times \ldots \times s_n}) \rightarrow \alpha$	the update (kernel) function, accepting the current time stamp, the current location, and the current grid state as an input, computing the resulting target value;
kernel shape	$(int^n)^*$	a static list of relative offsets to cells accessed by the kernel, determining its shape; this parameter must be a compile-time constant
boundary function	$(int, int^n, \alpha^{s_1 \times \ldots \times s_n}) \rightarrow \alpha$	the update function for boundary cases, where some elements are not within the grid; this one will be invoked whenever one of the input elements of the kernel is outside the special size of the internally maintained grid
initialization	$(int^n) \rightarrow \alpha$	a function computing the initial

function		value for a cell at a given coordinate
finalization function	$(int^n, \alpha) \rightarrow unit$	a function consuming the value of a cell at the end of a stencil based computation
observers	$\begin{pmatrix} (int, int^n) \rightarrow bool \\ (int, int^n, \alpha) \rightarrow unit \end{pmatrix}^*$	a list of pairs, where each element describes an observer of the computation; observers consist of two functions: one to describe interest in the result of a certain time step and location, and a second to trigger the observation; the length of the list of observers must be a compile-time constant

Thereby, the generic type α within those parameters is the element type to be stored within the grid (e.g. temperature, air pressure, or the combination of those).

The stencil implementation is responsible for

- creating internal data structures for maintaining the computation state
- initializing the state of each cell within this grid utilizing the initialization function
- conducting updates on the internal state for the given number of time steps utilizing the kernel and boundary functions, thereby ensuring that no data dependency among the update operations is violated
- triggering observer functions as requested over the course of the computation
- finishing the computation by applying the finalization function to each element of the resulting grid state
- cleaning up internally created grid data structures.

The general stencil operator, as described above, may be wrapped utilizing a set of simplified, more specialized operators providing default implementations for some of the parameters. For instance, observers may be omitted by utilizing an empty list of observers, finalization functions may be no-ops or initialization functions may be based on default-initialization logic build into the C++ language.

The internal implementation of the stencil framework remains implementation dependent.

5.1.3 Parallel Map-Reduce Operations

Parallel and distributed applications often require some sort of aggregation functionality that allows computing a single value from a greater amount of potentially distributed data. For this reason, the User API shall provide such functionality via a construct implementing a parallel map-reduce operation.

Requirements:

The following features need to be covered:

- iterate over a given range of data elements and extract their values of interest by means of a user-specified function (map step)
- aggregate these values into a single result value by means of a user-specified function (reduction step)

For efficiency reasons, blocks of the range being processed will be aggregated sequentially. The granularity of those blocks is thereby chosen by the runtime system. To enable users to gain advantage of this property, user defined operations shall be supported for initializing, aggregating, and extracting reduction results on a per-block level.

Interface Specification:
Let $iterator$ be of *RandomAccessIterator*[2] type of C++. Then the *preduce* operator provides map and reduce semantics with the following parameters:

Name	Type	Description
begin	$iterator$	Points to the inclusive beginning of the range to be processed by the reduction
end	$iterator$	Points to the exclusive end of the range to be processed by the reduction
initialization function	$() \rightarrow \alpha$	A function initializing a local temporary value at the start of the reduction of a block of data.
aggregation function	$(iterator, \alpha) \rightarrow \alpha$	A function used to aggregate (fold) elements of a block of data into an aggregated value;
finalization function	$(\alpha) \rightarrow \beta$	A function extracting the reduction result of obtained from a block of data;
reduction function	$(\beta, \beta) \rightarrow \beta$	A function combining the result of two (concurrently processed) reduction steps on two sub-ranges of the overall processed block;

The *preduce* operator returns a $treeture\langle\beta\rangle$ instance that will hold the aggregated result.

[2] http://en.cppreference.com/w/cpp/concept/RandomAccessIterator

5.2 Parallel Data Structures

To increase productivity when working with the User API, it provides a variety of common parallel data structure types – used by the pilot applications – that match the data item specification in Section 4.4.

5.2.1 Grid

A data structure that is frequently encountered within high-performance codes are n-dimensional arrays of values. While many programming languages support such structures for arbitrary dimensions, C/C++ only supports one-dimensional, dynamically sized arrays natively. Frequently, these arrays form the foundation for the distribution of applications within distributed memory systems. Thus, due to the lack of language support, the creation and management of these structures is left to the user, forming a major obstacle for the usability of C++ on distributed memory systems.

Requirements:

To ease the use of C++ for use cases depending on such structures, the AllScale User API shall provide a uniform Grid data structure providing the following features:

- regular n-dimensional array whose size is defined dynamically at creation
- efficient read/write random access operators
- efficient scan operation (visiting all elements)
- type-parameterized in its element type and number of dimensions
- enforcing the serializability of its element types
- implementing AllScale's data item concept to facilitate the distribution and automated management of grid instances

Interface Specification:

Let $Grid\langle\alpha, n\rangle$ be the abstract data type family implemented by the AllScale User API to represent n-dimensional grids, where α is a type variable specifying the element type and n the number of dimensions. Furthermore, let $type\langle\alpha\rangle$ be the meta type of type α. The following operations need to be defined on grids:

Name	Type	Description
create	$(type\langle\alpha\rangle, int^n) \rightarrow Grid\langle\alpha, n\rangle$	creates a new n-dimensional grid with element type α of the given size;
destroy	$(Grid\langle\alpha, n\rangle) \rightarrow unit$	deletes the given grid
read	$(\text{Grid}\langle\alpha, n\rangle, int^n) \rightarrow \alpha$	reads an element from the given grid, at the specified coordinates
write	$(\text{Grid}\langle\alpha, n\rangle, int^n, \alpha) \rightarrow unit$	updates the element within the given grid, at the given coordinates to the given value
scan	$(\text{int}^n, int^n, (int^n) \rightarrow \beta) \rightarrow treeture\langle unit\rangle$	applies the given function (in parallel) to all elements of the given

		interval in an arbitrary order; given parameters a and b, the interval includes the points $\{p \in int^n \mid \forall i \in [1..n] a_i \leq p_i < b_i\}$; the operations is performed asynchronously and may be synchronized up on through the resulting treeture

Due to the constraints of the data item concept, the Grid data structure is required to provide façade, region and fragment implementations. The Grid's shared data segment contains the overall grid size. The table above is outlining the interface of the façade. The mathematical structure selected for describing grid regions (explicit enumeration, bounding boxes, octagons, polyhedrals) is implementation dependent. However, static parameters for selecting among the available region implementations should be provided to facilitate algorithmic optimizations to be conducted by end users.

5.2.2 Adaptive Grid

An advanced variant of the Grid structure covered above also frequently encountered within simulation code is the *Adaptive Grid*. In addition to the properties of the Grid, the Adaptive Grid provides means to nest grids within grid cells. For a given Adaptive Grid instance, each top-level grid cell contains an identically structured fixed-length sequence of grids. The first of those contains a single cell. Every consecutive grid contains a multiple number of cells per dimension of its predecessor. Each top-level grid cell comprising the sequence of its nested grids is referred to as an *adaptive grid cell*.

To facilitate the utilization of this advanced concept, an additional data structure providing an implementation of it shall be offered by the AllScale User API.

Requirements:

The implementation of the Adaptive Grid as provided by the User API is required to provide the following features:

- regular n-dimensional adaptive grid whose size is defined dynamically at creation, comprising statically structured adaptive grid cells
- efficient read/write random access operators
- refinement and coarsening operations
- efficient scan operation
- type-parameterized in its element type, number of dimensions, and structure of adaptive grid cells
- enforcing the serializability of its element types
- implementing AllScale's data item concept to facilitate the distribution and automated management of grid instances

Interface Specification:
Let $AGrid\langle\alpha, n, [r_1, .., r_l]\rangle$ be the abstract data type family implemented by the AllScale User API to represent n-dimensional adaptive grids, where α is a type variable specifying the element type, n the number of dimensions, $r_1, \dots, r_l$ the refinement factors, and l the number of refinement levels. Thus, the size of the grid at level i is defined by

$$s(i) = \begin{cases} [1, \dots, 1] \in int^n & if\ i = 0 \\ s(i-1) * r_i & otherwise \end{cases}$$

To address elements within an Adaptive Grid an extension of the Grid coordinates is required. While elements within a Grid can be addressed using a single coordinate of type int^n, within the adaptive grid addition information regarding the location of the addressed element in the nested grid structure is required. Thus, additional coordinates to navigate through those refinement layers are required. Hence, to address an element within and adaptive grid, a *hierarchical coordinate* of type

$$(int^n)^+$$

is required. For instance, the coordinate

[[7,3], [2,4], [8,2]]

addresses the element located within the cell that can be reached by navigating first to the top-level cell [7,3], continuing to cell [2,4] of its first refinement layer, and ending up within cell [8,2] of the second refinement layer.

Let $seq\langle r_1, \dots, r_l\rangle$ be the static meta-type of a sequence of integers $r_1, \dots, r_l$. The following operations need to be defined on adaptive grids:

Name	Type	Description
create	$(type\langle\alpha\rangle, int^n, seq\langle r_1, \dots, r_l\rangle) \rightarrow AGrid\langle\alpha, n, [r_1, .., r_l]\rangle$	creates a new adaptive grid containing the given element type, of the given size, and grid cell structure;
destroy	$(AGrid\langle\alpha, n, [r_1, .., r_l]\rangle) \rightarrow unit$	deletes the given adaptive grid
read	$(AGrid\langle\alpha, n, [r_1, .., r_l]\rangle, (int^n)^+) \rightarrow \alpha$	retrieves the elementary value stored within the given adaptive grid at the specified hierarchical coordinates
write	$(AGrid\langle\alpha, n, [r_1, .., r_l]\rangle, (int^n)^+, \alpha) \rightarrow unit$	updates the elementary value stored within the

		given adaptive grid at the specified hierarchical coordinates
refine	$(AGrid\langle\alpha, n, [r_1, .., r_l]\rangle, (int^n)^+, Grid\langle\alpha, n\rangle) \rightarrow unit$	refines the resolution of a cell within the given adaptive grid addressed by the given hierarchical coordinate by refining the resolution and inserting the given grid data as the refined information
coarsen	$(AGrid\langle\alpha, n, [r_1, .., r_l]\rangle, (int^n)^+, \alpha) \rightarrow unit$	coarsens the resolution of a cell within the given adaptive grid addressed by the given hierarchical coordinate by coarsening the resolution and inserting the given value as the coarsened information
getLevel	$(AGrid\langle\alpha, n, [r_1, .., r_l]\rangle, (int^n)^+) \rightarrow int$	obtains the currently active resolution level at the specified hierarchical grid position;
scan	$(int^n, int^n, AGrid\langle\alpha, n, [r_1, .., r_l]\rangle, ((int^n)^+) \rightarrow \beta) \rightarrow treeture\langle unit\rangle$	applies the given function (in parallel) to all active hierarchical coordinates within the given interval in an arbitrary order; given the first two parameters a and b, and the set of active hierarchical addresses $A \subset (int^n)^+$ in the given adaptive gird, the interval includes the points

		$\{[p, \dots] \in A \mid \forall i \in [1..n] a_i \leq p_i < b_i\}$; the operations is performed asynchronously and may be synchronized up on through the resulting treeture

Due to the constraints of the data item concept, the Adaptive Grid data structure is required to provide façade, region and fragment implementations. The Adaptive Grid's shared data segment contains the size of the top-level grid. The table above is outlining the interface of the façade. The mathematical structure selected for describing adaptive grid regions (explicit enumeration, bounding boxes, octagons, polyhedrals) is implementation dependent. However, static parameters for selecting among the available region implementations should be provided to facilitate algorithmic optimizations to be conducted by end users.

5.2.3 Mesh

The Mesh data structure is designed to represent a graph structure of multiple node types that are connected through various types of edges. Furthermore, a Mesh may consist of several layers, which describe the same graph in different levels of detail. Hierarchical edges may connect the same nodes of different layers.

Beside the topological information maintained by Mesh instances, means to maintain attributes associated to nodes, edges, and hierarchical edges within a Mesh need to be included. For instance, node IDs, coordinates, volumes, temperatures, and other domain space specific properties may be incorporated through this facility.

Requirements:

The implementation of the Mesh as provided by the User API is required to provide the following features:

- a representation of a mesh structure parameterized by its nodes, edges, hierarchical edges and number of layers
- an associated data structure to attach attributes to the nodes of meshes
- means to build meshes by incrementally adding nodes, edges, and hierarchical edges
- efficient operators to navigate the mesh structure
- efficient operators to retrieve and update attributes associated to nodes
- efficient operators for iterating over all nodes or edges of a given type on a given level
- enforcing the serializability of its attribute types
- implementing AllScale's data item concept to facilitate the distribution and automated management of Mesh instances

Interface Specification:

Let $n_1, \dots, n_m$ be a list of node types, $e_1, \dots, e_k \in \{n_1, \dots, n_m\}^2$ be a list of edge types, and $h_1, \dots, h_o \in \{n_1, \dots, n_m\}^2$ be a list of hierarchical edge types. Then the type

$$Mesh\langle [n_1, \dots, n_m], [e_1, \dots, e_k], [h_1, \dots, h_o], l \rangle$$

represents the type of a Mesh structure including the given node, edge, and hierarchical edge types on l layers. Furthermore, let

$$id\langle \alpha, l \rangle$$

be an identifier for an element of type α on layer l within a Mesh – thus the type of ID utilized for addressing nodes, edges, or hierarchical edges within meshes. Also, let

$$MData\langle n, l, \alpha \rangle$$

be the type of an attribute collection associating values of type α to nodes of type n located on layer l of some mesh instance. Finally, let

$$MBuilder\langle [n_1, \dots, n_m], [e_1, \dots, e_k], [h_1, \dots, h_o], l \rangle$$

be the type of construction utility for creating meshes.

The following operations need to be defined on the types introduced above. For the mesh builder those comprise:

Name	Type	Description
create	$(type\langle Mesh\langle n, e, h, l \rangle \rangle) \rightarrow MBuilder\langle n, e, h, l \rangle$	creates a builder for the given mesh type; initially, the mesh under construction is left empty
destroy	$(MBuilder\langle n, e, h, l \rangle) \rightarrow unit$	destroys a builder instance
addNode	$(MBuilder\langle [n_1, \dots, n_m], e, h, l \rangle, type\langle n \rangle, type\langle i \rangle) \rightarrow id\langle n, i \rangle$	creates a new node of the given type n on the given level i within the mesh under construction
link	$(MBuilder\langle n, [(n_{1a}, n_{1b}), \dots, (n_{ka}, n_{kb})], h, l \rangle, id\langle n_{ia}, j \rangle, id\langle n_{ib}, j \rangle) \rightarrow unit$	adds an edge to the mesh under construction
link	$(MBuilder\langle n, e, [(n_{1a}, n_{1b}), \dots, (n_{oa}, n_{ob})], l \rangle, id\langle n_{ia}, j+1 \rangle, id\langle n_{ib}, j \rangle) \rightarrow unit$	adds a hierarchical edge to the mesh

		under construction
toMesh	$(MBuilder\langle n,e,h,l\rangle) \rightarrow Mesh\langle n,e,h,l\rangle$	obtains a copy of the mesh under construction

For the Mesh data structure itself, covering the topological information of the processed mesh, the following operations have to be provided:

Name	Type	Description
store	$(Mesh\langle n,e,h,l\rangle) \rightarrow byte^*$	converts a given mesh into a byte array (serialization)
load	$\begin{pmatrix} byte^*, \\ type\langle Mesh\langle n,e,h,l\rangle\rangle \end{pmatrix} \rightarrow Mesh\langle n,e,h,l\rangle$	converts a given byte array into a mesh (deserialization)
destroy	$(Mesh\langle n,e,h,l\rangle) \rightarrow unit$	destroys a mesh
getNeighbors	$\begin{pmatrix} Mesh\langle n,[(n_{1a},n_{1b}),\dots,(n_{ka},n_{kb})],h,l\rangle \\ type\langle(n_{ia},n_{ib})\rangle \\ id\langle n_{ia},j\rangle \\ \rightarrow id\langle n_{ib},j\rangle^* \end{pmatrix}$	obtains a list of neighbors of a given node following a given kind of edge
getParents	$\begin{pmatrix} Mesh\langle n,e,[(h_{1a},h_{1b}),\dots,(h_{oa},h_{ob})],l\rangle \\ type\langle(h_{ia},h_{ib})\rangle \\ id\langle h_{ia},j\rangle \\ \rightarrow id\langle h_{ib},j+1\rangle^* \end{pmatrix}$	obtains a list of parents of a given node following a given kind of hierarchical edge
getChildren	$\begin{pmatrix} Mesh\langle n,e,[(h_{1a},h_{1b}),\dots,(h_{oa},h_{ob})],l\rangle \\ type\langle(h_{ia},h_{ib})\rangle \\ id\langle h_{ib},j\rangle \\ \rightarrow id\langle h_{ia},j-1\rangle^* \end{pmatrix}$	obtains a list of children of a given node following a given kind of hierarchical edge
scan	$\begin{pmatrix} Mesh\langle[n_1,..,n_m],e,h,l\rangle \\ type\langle n_i\rangle \\ type\langle j\rangle \\ (id\langle n_i,j\rangle) \rightarrow \beta \\ \rightarrow treeture\langle unit\rangle \end{pmatrix}$	applies a given operation to every instance of a selected node type on a selected level within the given mesh
scan	$\begin{pmatrix} Mesh\langle n,[(n_{1a},n_{1b}),\dots,(n_{ka},n_{kb})],h,l\rangle \\ type\langle(n_{ia},n_{ib})\rangle \\ type\langle j\rangle \\ (id\langle n_{ia},j\rangle,id\langle n_{ib},j\rangle) \rightarrow \beta \\ \rightarrow treeture\langle unit\rangle \end{pmatrix}$	applies a given operation to every instance of a selected edge type on a selected level within the given mesh

scan	$\begin{pmatrix} Mesh\langle n, e, [(n_{1a}, n_{1b}), \dots, (n_{oa}, n_{ob})], l\rangle \\ type\langle (n_{ia}, n_{ib})\rangle \\ type\langle j\rangle \\ (id\langle n_{ia}, j+1\rangle, id\langle n_{ib}, j\rangle) \to \beta \\ \to treeture\langle unit\rangle \end{pmatrix}$	applies a given operation to every instance of a selected hierarchical edge type on a selected pair of adjacent levels within the given mesh

Finally, for attribute stores (MData) the following list of operators have to be offered:

Name	Type	Description
create	$\begin{pmatrix} Mesh\langle [n_1, .., n_m], e, h, l\rangle \\ type\langle n_i\rangle \\ type\langle j\rangle \\ type\langle \alpha\rangle \\ \to MData\langle n_i, j, \alpha\rangle \end{pmatrix}$	creates an attribute storage associating a value of type α to each node instance of type n_i on layer j present in the given mesh
destroy	$(MData\langle n, l, \alpha\rangle) \to unit$	deletes the given attribute storage
read	$(MData\langle n, l, \alpha\rangle, id\langle n, l\rangle) \to \alpha$	retrieves the value of an attribute associated to the given node form the given attribute store
write	$(MData\langle n, l, \alpha\rangle, id\langle n, l\rangle, \alpha) \to unit$	updates the value of an attribute associated to the given node in the given attribute store

Due to the constraints of the data item concept, the Mesh data structure and the MData type is required to provide façade, region and fragment implementations. The table above is outlining the interface of their façades. The mathematical structure selected for describing regions – thus subgraphs of the mesh and parts of the data store – are implementation dependent. However, static parameters for selecting among the available region implementations shall be provided to facilitate algorithmic optimizations to be conducted by end users.

5.2.4 Bag

Many scientific simulations deal with physical entities that do not need to be selectively identifiable or addressable, but whose presence is required due to their aggregated effect on the simulated environment. Examples for such entities are charged particles that travel through space, forming and influencing electromagnetic fields in the process. Thereby the mere presents of those elements is important. They neither need to be unique nor in a particular order. However, concurrently adding and removing elements should be efficiently supported.

Classical tree or hash based set implementations provide additional constraints on the contained elements (uniqueness) that make them unsuitable for the described use case scenario. Dynamic arrays, on the other hand, do not support efficient manipulation operations, while linked list based approaches exhibit generally low efficiency in parallel contexts due to the gradual defragmentation of the elements memory locations and frequent system calls for memory allocation and de-allocation.
Thus, to tackle the described use case, a bag data structure (aka multiset) is to be implemented, satisfying the following requirements.

Requirements:
To ease the use of C++ for use cases depending on such structures, the AllScale User API shall provide a data structure that offers the following features:

- support for concurrent insertion of (potentially duplicated) elements
- support for user-specified concurrent updates of the elements' properties
- support for the concurrent filtering (extraction) of elements that meet certain user-specified criteria

Interface Specification:
Let $Bag\langle\alpha\rangle$ be the abstract data type family implemented by the AllScale User API to represent this data structure holding an unordered collection of potentially duplicated elements, where α is a type variable specifying the element type. Furthermore, let $type\langle\alpha\rangle$ be the meta type representing type α. Moreover, let $Collection\langle\alpha\rangle$ be a container type that holds elements of type α. Then, the following operations need to be defined on a Bag:

Name	Type	Description
create	$(type\langle\alpha\rangle) \rightarrow Bag\langle\alpha\rangle$	creates a new, empty bag with element type α
destroy	$(Bag\langle\alpha\rangle) \rightarrow unit$	deletes the given bag
filter	$(Bag\langle\alpha\rangle, (\alpha) \rightarrow bool) \rightarrow treeture\langle Collection\langle\alpha\rangle\rangle$	filters the given Bag for elements for which the given user-specified predicate holds, removes these elements from the Bag and returns them in a collection; the operation is performed asynchronously and may be synchronized upon through the resulting treeture
insert	$(Bag\langle\alpha\rangle, Collection\langle\alpha\rangle) \rightarrow treeture\langle unit\rangle$	inserts the given elements into the given Bag; the operation is performed asynchronously and may be synchronized upon through the resulting treeture
update	$(Bag\langle\alpha\rangle, (\alpha) \rightarrow \alpha) \rightarrow treeture\langle unit\rangle$	applies the given update function (in parallel) to all elements of the given Bag; the operation is performed asynchronously and may be synchronized upon through the resulting treeture

Due to the constraints of the data item concept, the Bag data structure is required to provide façade, region, fragment, and shared data segment implementations. The table above is outlining the interface of the façade. The mathematical structure selected for describing bag regions is implementation dependent.