H2020 FETHPC-1-2014

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

Project Number 671603

D2.3 – AllScale System Architecture

WP2: Requirements and overall system architecture design

Version: 1.0

Author(s): Herbert Jordan (UIBK), Roman Iakymchuk (KTH), Thomas Fahringer (UIBK), Peter Thoman (UIBK), Thomas Heller (FAU), Xavi Aguilar (KTH), Khalid Hasanov (IBM), Kiril Dichev (QUB), Emanuele Ragnoli (IBM), Benoit Leonard (NUM)

Date: 03/01/17

Copyright © AllScale Consortium Partners 2017

Due date:	PM16
Submission date:	31/01/2017
Project start date:	01/10/2015
Project duration:	36 months
Deliverable lead organization	UIBK
Version:	1.0
Status	Final
Author(s):	Herbert Jordan (UIBK), Roman Iakymchuk (KTH), Thomas Fahringer (UIBK), Peter Thoman (UIBK), Thomas Heller (FAU), Xavi Aguilar (KTH), Khalid Hasanov (IBM), Kiril Dichev (QUB), Emanuele Ragnoli (IBM), Benoit Leonard (NUM)
Reviewer(s)	Person 1 (Org. short name), Person 2 (Org. short name)

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.

Copyright © AllScale Consortium Partners 2017

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 (NUMECA), 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	08/01/16	First draft	Herbert Jordan
0.2	18/02/16	Added compiler/runtime interface	Herbert Jordan
0.3	06/12/16	Updated to match developments	Herbert Jordan
0.4	07/12/16	Updated Intro, API and Compiler	Herbert Jordan
0.5	09/12/16	Updated Runtime & Subsystems	Herbert Jordan
0.6	19/12/16	Added Pilots, IO API, and Section 3	Herbert Jordan
0.7	20/12/16	Added Example Section 4	Herbert Jordan

Copyright © AllScale Consortium Partners 2017

0.8	21/12/16	Integrated Feedback	Herbert Jordan, Roman Iakymchuk, Thomas Fahringer, Kiril Dichev, Peter Thoman, Thomas Heller, Xavi Aguilar, Khalid Hasanov, Emanuele Ragnoli, Benoit Leonard
1.0	03/01/17	Final proof reading	Herbert Jordan

Copyright © AllScale Consortium Partners 2017

Table of Contents

1 Introduction 7
1.1 Formalism: 7
1.2 Terminology 8
2 AllScale System Architecture Design 10
2.1 Overview 10
2.2 AllScale API 12
2.2.1 Overview 12
2.2.2 Technological Base 13
2.2.3 AllScale Core API 14
2.2.4 AllScale User API 21
2.3 AllScale Pilot Applications 22
2.3.1 iPIC3D: Implicit Particle-in-Cell Code 23
2.3.2 FINE™/OPEN: Unstructured CFD solver 24
2.3.3 AMDADOS: Adaptive Meshing and Data Assimilation 26
2.4 AllScale Compiler 27
2.4.1 Overview 27
2.4.2 Compiler Infrastructure 28
2.4.3 Data Requirement Analysis 29
2.5 AllScale Runtime System 30
2.5.1 Overview 30
2.5.2 Runtime Hardware and Application Model 31
2.5.3 Technological Base 34
2.5.4 Failure Tolerance 35
2.5.5 Scheduler Interface 36
2.5.6 Resilience Manager Interface 36
2.5.7 Monitoring Service Interface 37
2.6 AllScale Scheduler 38
2.7 AllScale Monitoring Service 39
2.8 AllScale Resilience Manager 40
3 AllScale Component Interfaces 42
4 Example AllScale Application 44
4.1 The Example 44
4.2 API Support 45

Copyright © AllScale Consortium Partners 2017

4.2.1 Fine Grained Synchronization ... 46
4.2.2 Distributed Memory Support ... 47
4.3 Compilation ... 48
4.4 Execution ... 49
4.5 Monitoring ... 51
4.6 Scheduling ... 51
4.7 Resilience ... 52
5 Conclusions and Future Work ... 52

Copyright © AllScale Consortium Partners 2017

1 Introduction

This document provides a complete overview on the architecture of the AllScale Environment. To that end, Section 2 covers the internal design of the involved components. The interfaces among those components are covered in Section 3, while Section 4 provides an example demonstrating the dynamic interaction of all the involved components.

The architecture presented within this document is the object of ongoing development and can thus, necessarily, only provides a snapshot of the AllScale system design. Details of interfaces or internal details of the components' design may diverge from the content presented within this document as deemed necessary for fulfilling the project's objectives.

Before getting to the details of the AllScale components, this introduction section is providing a summary of the formalism and terminology utilized through the rest of this document.

1.1 Formalism:

To describe the design of concepts within this document the concept of abstract data types (ADT) is utilized. Thus, objects within the system specification are associated with abstract types on which (equally abstract) operations may be performed. While the details of the actual implementation of those types and operations are negligible for the overall system architecture, their signatures define the means for the interaction of components. Furthermore those constructs provide the level of abstraction needed by this document to focus on the high-level design concepts of the AllScale Environment. The actual implementation details of the various abstract types and operators are found within the corresponding deliverables and/or the actual source code realizing those concepts. Where necessary, those deliverables are referenced.

To specify abstract types and operators throughout this document, the following symbols and type constructors are utilized:

Symbol	Interpretation
$\boldsymbol{bool}$	abstract type for Boolean values
$\boldsymbol{int}$	abstract types for natural numbers
$\boldsymbol{unit}$	abstract type for the unit constant (void type in C/C++)
$\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\delta}, \dots$	type variables
$(\boldsymbol{T_1}, \boldsymbol{T_2}, \dots, \boldsymbol{T_n})$	tuple type with n components; component i is of type T_i
$\boldsymbol{T_1} \rightarrow \boldsymbol{T_2}$	type of a function accepting a value of type T_1 as an argument and returning a value of type T_2
$\boldsymbol{2^T}$	the power set of elements of type T

Copyright © AllScale Consortium Partners 2017

T^*	a list of elements of type T, may be empty
T^+	a list of elements of type T, not empty

1.2 Terminology

The following table provides a summary of the terms utilized throughout this document to address various components and aspects of the overall AllScale architecture:

Term	Description
AllScale	the entire project, comprising the AllScale Environment, pilots, and infrastructure
AllScale API	the AllScale component to be utilized by end users to develop applications, comprising the necessary parallel primitives
AllScale application	a program utilizing the AllScale Environment for implementing a solution for some particular, domain specific problem
AllScale Compiler	the compiler component of the AllScale Toolchain analyzing and transforming end user code to convert it to code processible by the AllScale Runtime System
AllScale Core API	a fix set of basic parallel primitives for the creation of AllScale applications; part of the AllScale API
AllScale Environment	the entire AllScale software stack, comprising the API, compiler, runtime system, and runtime system subsystems
AllScale Infrastructure	the hardware and software resources utilized for the development of AllScale, including source repositories and their configuration scripts, the AllScale website, and build, test, and benchmark servers
AllScale Monitoring Service	a runtime subsystem collecting information on a processed application as well as properties of the utilized hardware infrastructure
AllScale pilot	one of the three AllScale applications (iPIC3D, AMDADOS, or Fine/Open) created to demonstrate the abilities of the AllScale Environment
AllScale Resilience Manager	a runtime subsystem managing backup and recovery operations of an AllScale application during execution

Copyright © AllScale Consortium Partners 2017

AllScale Runtime	short form for AllScale Runtime System
AllScale Runtime System	the component managing the execution of an AllScale Application by controlling the distribution of computation and data among available hardware resources as well as hardware parameters
AllScale Scheduler	the runtime subsystem deciding on the workload and data distribution as well as the utilization and configuration of the available hardware resources
AllScale SDK	an empty AllScale application providing a template for building new AllScale applications
AllScale Toolchain	the AllScale Compiler and the AllScale Runtime System utilized to compile and run applications
AllScale User API	a freely extensible set of higher level parallel primitives built on top of the Core API providing convenience functionality to end users; a basic set of constructs is provided as part of the AllScale API
data item	the unit of data managed by the runtime system; data items can be uniquely addressed and actively distributed among nodes by the runtime system
dynamic optimizer	an alternative designation of the AllScale Scheduler component stressing its central obligation of steering an application execution towards a given optimization objective (e.g. low execution time or low energy)
end user	in the context of the API and compiler: a developer utilizing the AllScale Environment for developing an application; in the context of the pilots, the runtime system and its sub-components: the person executing an AllScale application
locality	a AllScale Runtime System process participating in the computation of an AllScale application, providing and managing an address space for data items
node	a physical entity of a target architecture, governed by a single OS image
runtime subsystem	components embedded within the AllScale Runtime System providing services to the processed application; this includes the scheduler, the monitoring system and the resilience manager
standard toolchain	a C++ build environment capable of compiling and

Copyright © AllScale Consortium Partners 2017

	running C++14 applications; e.g. setups utilizing the GCC or clang compiler
target architecture	a computer utilized for running an AllScale application
task	in the API layer: the unit of work; in the runtime layer: synonym to work item
work item	the unit of work managed by the runtime system; work items can be uniquely addressed, have associated data requirements, may be split into smaller work items and migrated among nodes

2 AllScale System Architecture Design

In this chapter the overall system architecture of the AllScale Environment and pilots is covered. Section 2.1 starts with a general overview, enumerating the main components and outlining their interaction. It is followed by Sections 2.2 - 2.8 elaborating on the architectural details of the individual system components.

2.1 Overview

Figure 1 outlines the overall architecture of the AllScale Environment and pilots. The architecture comprises seven components:

- The AllScale API (see Section 2.2)
- The AllScale Pilot Applications (see Section 2.3)
- The AllScale Compiler (see Section 2.4)
- The AllScale Runtime System (see Section 2.5)
- The AllScale Scheduler (see Section 2.6)
- The AllScale Monitoring Service (see Section 2.7)
- The AllScale Resilience Manager (see Section 2.8)

Copyright © AllScale Consortium Partners 2017

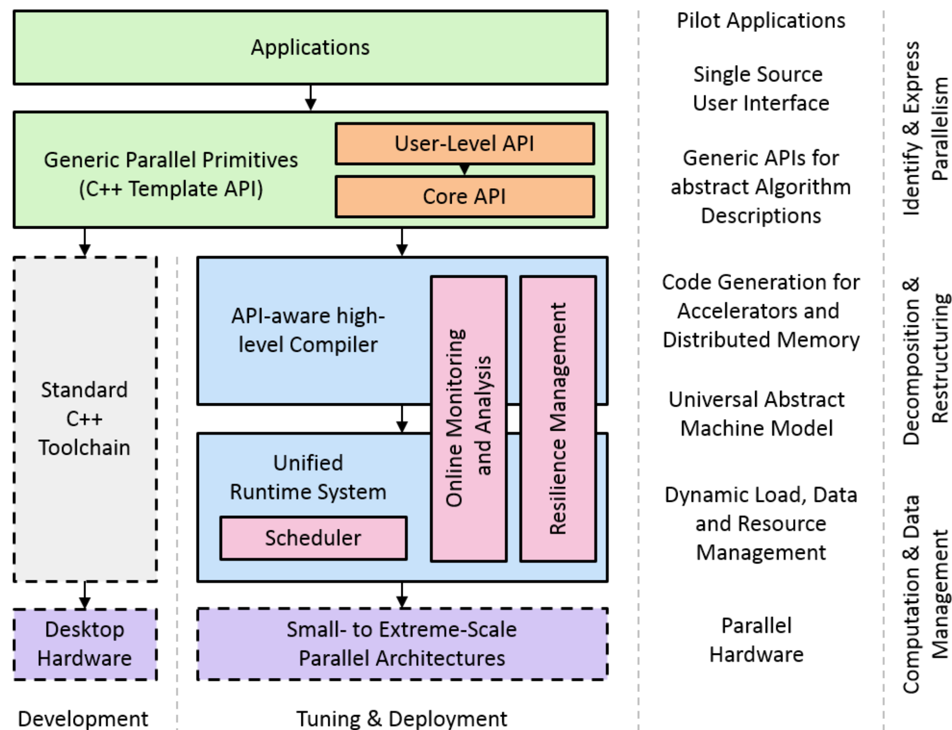


Figure 1. Overview on AllScale Architecture

The pilot applications are implemented based on a set of generic parallel APIs offered by the AllScale API. The API itself is subdivided into two layers: the User-Level API and the Core API. The Core API layer covers a small, concise set of essential primitives: constructs for expressing data structures, parallel control flows, and synchronization operations. While concise and expressive, the core API is not supposed to be directly utilized by application developers due the complex nature of included constructs. Their main purpose is to provide a compact interface for actual implementations, in particular the one realized by the AllScale compiler. To support application developers, the User API layer is introduced and maintained by expert developers. It maps common parallel patterns to the Core-API primitives. Thus, the User-Level API is a flexible, open layer that can be extended and customized without requiring any changes in the AllScale Environment.

Codes implemented utilizing the AllScale API can be compiled by standard C++ tools and executed on parallel, shared memory machines. This modus is intended for use during the development and debugging phases of an application development project. However, for obtaining an extreme scale high-performance version that benefits from all the services offered by the AllScale Environment, the same code base has to be compiled by the API-aware AllScale Compiler. Unlike standard C++ compilers, the AllScale Compiler is aware of the interpretation of the parallel primitives offered by the Core API layer and restructures the application by introducing additional versions of encountered parallel code fragments. Each version targets a different architecture and/or represent a different trade-off between multiple optimization objectives (e.g. more parallelism vs. less overhead). Together with information describing the specific traits of the versions – such as the scenario they have been tuned for – as

Copyright © AllScale Consortium Partners 2017

well as details regarding their (data) dependencies, the resulting set of implementations are forwarded to the AllScale Runtime System.

The AllScale Runtime System provides the infrastructure for the AllScale Dynamic Optimizer (aka the scheduler) to dynamically auto-tune a running program for multiple objectives to achieve a desired trade-off among the considered tuning objectives. It does so, by distributing workload and data dynamically throughout the system, coordinating the exchange of information, and continuously adjusting performance relevant hardware parameters. The scheduler can, for this purpose, rely on dynamically updated knowledge provided by the AllScale Monitoring Service capturing the state of the target system as well as the managed program execution. Furthermore, the AllScale Resilience Manager aids the scheduler in preparing for and responding to errors occurring during the program execution. Therefore, it coordinates the utilization of self-healing and self-stabilizing mechanisms integrated in or revealed by the AllScale Environment.

Further details about these interactions and the internal design of the involved component are covered in the following sections.

2.2 AllScale API

2.2.1 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 needed synchronization steps within application code. The API is subdivided 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 purpose of the subdivision into a Core and User API is to enable the implementation of a variety of parallel primitives on top of a small, concise set of central constructs which can be utilized to provide portability among different implementations of the AllScale Core API. In particular, within the AllScale project, two implementations are developed:

- a shared memory, pure C++ implementation which can be compiled by any C++14 compatible compiler which furthermore serves as a reference implementation and development platform for the pilot applications, and
- the implementation utilizing the AllScale Compiler and Runtime System, utilizing a combination of static program analysis, code generation, scheduling, and resilience techniques to provide a highly scalable and portable implementation of the Core API

Copyright © AllScale Consortium Partners 2017

Additional parallel constructs may be introduced in the AllScale User API without the necessity of altering the underlying Core API implementation. Thus, the User API layer provides an effective way of extending the range of supported parallel patterns.

Furthermore, by introducing the User API, application developers are shielded from the complexity of the Core API constructs. Due to the introduction of the User API efficient implementations of primitives native to the domain of the applications can be provided by parallelization experts. Thus, the overall task of providing efficient parallel codes is distributed among three contributors:

- the *domain expert* aiming on obtaining the most effective algorithmic solution for the problem of interest
- the *high performance computing expert* able to develop efficient domain specific primitives to be utilized by the domain expert, focusing on e.g. communication and synchronization overheads and cache efficiency
- the *system level expert* focusing on providing the most flexible and portable implementation of the Core API, thereby handling load management, scheduling, resilience, and hardware management obligations

The separation of responsibilities also effects the code base. By shielding the domain expert from all the underlying details (synchronization, communication, cache efficiency, scheduling, utilization of low-level parallel APIs …), the resulting application code remains free of the otherwise necessary management code. This positively affects the maintainability of the resulting applications and thus the productivity of the domain expert.

2.2.2 Technological Base

The AllScale API makes heavy use of C++'s code template based meta-programming feature. This build-in language feature of C++ enables the scripted generation of code during the first stages of the compilation process. Widely utilized examples include the generation of data structures like vectors, sets, or maps specialized to specific type parameters. However, the capabilities of this features reach much further. Type parameters may be inspected, and cases may be distinguished. Thus, the meta-programming feature can be utilized to synthesize specialized code. It also enables the generic implementation of primitives, where a single primitive may cover a wide range of use cases, without the introduction of any abstraction overhead.

All primitives of the AllScale Core API are generic primitives making heavy use of C++ meta-programming features for the automated synthetization of program code. So may all of the AllScale User API constructs, to improve their usability, reusability, and flexibility.

The AllScale API utilizes C++ template features introduced by the C++14 standard revision, supported by recent versions of GCC, Clang, and Visual Studio. No additional libraries or system dependencies are required.

Copyright © AllScale Consortium Partners 2017

2.2.3 AllScale Core API

The AllScale Core API provides a concise set of generic primitives for expressing parallel control flows, communication, and synchronization operations.

2.2.3.1 Parallel Control Flow and Synchronization Primitives

The AllScale Core API provides a single primitive for running concurrent tasks. This primitive, the *prec* operator, is a higher order function combining three given functions into a new, recursive function[1]. The three combined input functions are:

- a function testing for the base case of a recursion
- a function processing the base case of a recursion
- a function processing the recursive step case

The *prec* operator combines those functions into a new recursive function which, for a given input parameter, conducts the specified computation accordingly. Thereby sub-tasks invoked by the step case function may be processed in parallel.

To support an arbitrary input type, the *prec* operator has the type

$$(\alpha \rightarrow bool, \alpha \rightarrow \beta, (\alpha, \alpha \rightarrow treeture\langle\beta\rangle) \rightarrow treeture\langle\beta\rangle) \rightarrow (\alpha \rightarrow treeture\langle\beta\rangle)$$

where α is the parameter type of the resulting recursive function and $treeture\langle\beta\rangle$ is a parameterized abstract data type (ADT) modeling a handle on parallel tasks (see below). The parameter of the *prec* operator are:

- a function of type $\alpha \rightarrow bool$ identifying base cases
- a function of type $\alpha \rightarrow \beta$ computing base cases
- a function of type $(\alpha, \alpha \rightarrow treeture\langle\beta\rangle) \rightarrow treeture\langle\beta\rangle$ computing a step cases where
    - the first parameter is the parameter for the recursive step and
    - the second parameter is a reference to the recursive function created by the *prec* call itself to compute sub-task

The resulting value of type $\alpha \rightarrow treeture\langle\beta\rangle$ is a function which, upon invocation, spawns a new task conducting the specified recursive operation in parallel. The resulting task handle can be utilized to orchestrate the parallel execution of additional tasks.

For the *treeture* ADT, the following operators are defined:

NAME	TYPE	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 finished and obtains the computed result

[1] For clarity we focus on the non-mutual recursive case in this document. The actual implementation provides support for the mutual recursive case as well.

Copyright © AllScale Consortium Partners 2017

done	$(\alpha) \rightarrow treeture\langle\alpha\rangle$	A function referencing a finished task which has produced the given value.
par	$\begin{pmatrix} treeture\langle\alpha\rangle \\ treeture\langle\beta\rangle, \\ (\alpha,\beta) \rightarrow \gamma \end{pmatrix} \rightarrow treeture\langle\gamma\rangle$	A function creating 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
seq	$\begin{pmatrix} treeture\langle\alpha\rangle \\ treeture\langle\beta\rangle, \\ (\alpha,\beta) \rightarrow \gamma \end{pmatrix} \rightarrow treeture\langle\gamma\rangle$	A function creating 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 function created by the *prec* operator is – beside the $done$, seq and par operators – the only constructor for tasks. The seq and par operators can be utilized for orchestrating the parallel control flow within the implementation of the step case of a parallel function, while the operators $wait$ and get can be used for synchronization and data transfers, similar to futures.

Within this document, we omit the third argument of the par and seq operator whenever it is a mere value consumption of the treeture results, not conducting any aggregation.

Due to the restriction of tasks being composed using the presented primitives, hierarchies of tasks can only be formed utilizing the par and seq operator recursively – both connecting two sub-tasks. Consequently, all task hierarchies are binary hierarchies, where each task has either no child task or two child tasks. Those child tasks are referred to as the left and right child task.

With the given primitives synchronization schemes equivalent to those of futures can be realized. However, for finer-grained dependencies, $treetures$ allow to obtain references to sub-tasks. Those references are modeled by the $tref$ (task reference) ADT and the following operators:

NAME	TYPE	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).

Copyright © AllScale Consortium Partners 2017

A logical sub-task is thereby the sub-task of a task that might in reality not yet or not ever exist. This may happen since it is left to the runtime to decide upon the granularity of the recursive tasks to be actually processed. Dependencies may address a finer grained granularity than the task which are actually processed. In those cases, task references are supposed to reference the task with the finest granularity comprising the workload of the intended task.

With those primitives more fine-grained synchronization between tasks within a single or across multiple task hierarchies can be realized. The following example outlines its application within the context of parallel loops.

Example:

To illustrate the interaction of the *prec* operator and the operators defined on tasks consider the following example. The objective is to initialize all elements of an array with 0. The sequential code is given by the following code fragment:

```
for( i = 0 … N ) {
        A[i] = 0;
}
```

For a nested recursive parallel implementation, the full range of N elements can be recursively sub-divided and recursive calls can be processed in parallel. This operation can be encoded utilizing the Core API primitives defined above as follows (in C++14 like syntax):

```
using range = pair<int,int>;
auto init = prec(
        [&](const range& r) { return r.second - r.first <= 1; },
        [&](const range& r) { for( i = r.first … r.second ) A[i] = 0; },
        [&](const range& r, const auto& rec) {
                int mid = r.first + (r.second – r. first) / 2;
                return par(
                        rec(range( r.first, mid )),
                        rec(range( mid, r.second )),
                );
        }
);
wait(init(0,N));
```

The *prec* operator call combines the base case test (at most one element), the base case computation step, and a function processing the step case into a function capable of initializing the entire array. The call to the *init* function (init(0,N)) triggers the parallel processing of the initialization and the concluding *wait* call awaits the completion of the task.

Notice, that the given code example demonstrates a general template for describing parallel loops based on the *prec* operator. This general pattern can be extracted into a generic function *pfor* of type

$$(\alpha, \alpha, \alpha \rightarrow \beta) \rightarrow task$$

defined by

Copyright © AllScale Consortium Partners 2017

```
task pfor(α l, α u, α → β body) {
    using range = pair< α, α >;
    auto loop = prec(
        [&](const range& r) { return r.second - r.first <= 1; },
        [&](const range& r) { for( i = r.first … r.second ) body(i); },
        [&](const range& r, const auto& pfor) {
            int mid = r.first + (r.second – r. first) / 2;
            return par(
                pfor(range( r.first, mid )),
                pfor(range( mid, r.second )),
            );
        }
    );
    return init(0,N);
}
```

which can be utilized for running parallel loops by invoking

```
wait(pfor(0,N,[&](int i) {
    A[i] = 0;
}));
```

which, when defining task handles as being implicitly synced upon destruction (C++ feature) can be further reduced to

```
pfor(0,N,[&](int i) {
    A[i] = 0;
});
```

The presented *pfor* function is one of the operators provided by the AllScale User API layer. It demonstrates the realization of a parallel pattern by utilizing the underlying Core API constructs.

Fine Grained Synchronization

In addition to enabling the awaiting of the completion of all the parallel loop iterations, the treeture returned by the pfor call can furthermore be utilized to synchronize on sub-sets of the iteration range. For instance, by utilizing the task reference operations *getLeft* and *getRight* references to the tasks processing the left and right half of the iteration space can be obtained. By applying this operation recursively, smaller subsets can be addressed. As a consequence, fine-grained synchronization between consecutive parallel loops can be realized.

For instance, in the code fragment

```
auto As = pfor(0,N,[&](int i) {
    A[i] = f(..);
});
pfor(1,N-1,[&](int i) {
    A[i] = f(A[i-1], A[i], A[i+1]);
}, wait_for_neighbors(As));
```

Copyright © AllScale Consortium Partners 2017

the treeture obtained from the first parallel loop is utilized as an additional input for the second loop describing execution dependencies to be considered. In this particular case each iteration *i* of the second loop may only be processed once the iterations *i-1*, *i,* and *i+1* of the first loop have been completed. The operators supported on task references facilitate the implementation of such operations within the AllScale User API. The corresponding details as well as the C++ specification of the described operators and ADTs is covered in the AllScale API specification deliverables (D2.5 and D2.6).

2.2.3.2 Data Structure Primitives

As for the description of the control flow, data structures, to be managed by any underlying runtime system implementation, need to be equally specified utilizing a uniform set of primitives. As for the design of the control flow primitives, the objective for the data structure primitives is to provide a flexible generic interface such that expert developers of the User Level API have a maximum of flexibility to express data structures to be managed by the underlying system.

To this end, the data structure primitives offered by the core are a mere specification of any potential type's interfaces and behaviors – in C++ terms a concept. Any type T to be managed by an AllScale API implementation has to provide the following properties:

- type T has to specify the following types:
  - a type F for fragments of the data storage
  - a type R for addressing sub-ranges of the data structure

Each of those types has to provide the following operators:

- for the fragment type F:
  - *create* of type $R \rightarrow F$ creating a fragment covering (at least) the specified range
  - *delete* of type $F \rightarrow unit$ deleting the given fragment
  - *resize* of type $(F, R) \rightarrow unit$ altering the capacity of the given fragment to cover at least the range given by the second parameter
  - *mask* of type $F \rightarrow T$ providing access to the data stored in the fragment F via the interface defined by type T
  - *extract* of type $(F, R) \rightarrow Archive$ extracting the data addressed by the second parameters from the fragment given by the first parameter and packing it into an archive; Archive is a generic type of a utility provided by the API implementations to serialize data to be transferred between address spaces;
  - *insert* of type $(F, R, Archive) \rightarrow unit$ importing the data stored in the given archive into the given fragment at the specified range R.
- for the range type R:
  - *union* of type $(R, R) \rightarrow R$ computing (a super-set of) the union of the two ranges covered by the two parameters
  - *intersect* of type $(R, R) \rightarrow R$ computing (a super-set of) the intersection of the two ranges covered by the two parameters
  - *empty* of type $(R) \rightarrow bool$ determining whether the given range is empty, thus contains no elements to be stored

Copyright © AllScale Consortium Partners 2017

   - *pack* of type $(R) \rightarrow Archive$ to serialize instances
   - *unpack* of type $(Archive) \rightarrow R$ to deserialize instances

Those concepts and interfaces are covered by corresponding C++ type requirements checked during the compilation process.

Example:

To illustrate the design of data structures to be managed by AllScale API implementations, consider the example of storing a 2D grid of doubles. The corresponding types could be:

- T = Grid2D<double> offering operators for accessing elements within a 2D structure, indexed by coordinates of type R, where the storage is provided by an instance of type F;
- F = GridFragment2D<double> realizing the actual storage of fragments of the data stored in Grid2D instances; the implementation may hold a reference to allocated memory plus the coordinates of the covered ranges
- R = Range2D consisting of the conjunction of 2D-coordinate pairs describing axis-aligned boxes covering the range to be described

The implementations of the corresponding operations are then realizing according to the requirements specified above.

2.2.3.3 IO Primitives

All sensible applications require IO for their operations. While high-performance IO is a research topic on its own, the AllScale Core API provides basic primitives to facilitate high-performance IO while keeping actual implementations abstract.

There are two different kind of IO operations supported:

- Streaming, supported through an AllScale IO interface facilitating e.g. the writing of simulation results to output streams
- Memory mapped IO for the structured loading of static input data for which efficient random access operations are required

The following sections cover the corresponding interfaces.

Stream Based IO

The underlying concept of the AllScale streaming IO interface is an out-of-order stream. Data entries can be atomically read from or written to such a stream. However, the order in which entries show up in the stream is undefined. Although tasks may be restricted due to imposed synchronization constraints to write data in a certain order to a stream pointing e.g. to a file, the resulting file may contain the written data in an arbitrary order. Furthermore, the API only guarantees the eventual visibility of a written element within an output stream, before the application terminates – not any particular timing. Thus, in particular, stream IO primitives may not be utilized for realizing synchronization operations among tasks.

Within the API we utilize the abstract types *istream* and *ostream* as a representation of an input or output stream. Furthermore, the following operators are offered:

Copyright © AllScale Consortium Partners 2017

NAME	TYPE	DESCRIPTION
read	$(istream) \rightarrow \alpha$	Atomically reads an element of type α from the given input stream
write	$(ostream, \alpha) \rightarrow unit$	Atomically writes the given element of type α to the given output stream, where it will be visible eventually

Additionally, a few operations for the global management of streams and their association to files are offered:

NAME	TYPE	DESCRIPTION
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

Streams are designed to be the main facility to be utilized by application developers to produce output data without the artificial introduction of extra synchronization overhead. Furthermore, the abstraction to streams, their global addressing through names, and the lack of guarantees on the output order enables the flexible migration of tasks throughout the system. Tasks holding a stream to a file X on some node may be moved to another node, where they get assigned a new stream pointing to the logically same file. However, in reality the stream may point to a physically different output file maintained by the local runtime process. The concatenation of all the locally maintained output files controlled by the various AllScale Runtime System instances on a system are logically forming the actual output file. Thus, no synchronization beyond the boundaries of an AllScale node is every required to facilitate streaming IO.

Copyright © AllScale Consortium Partners 2017

Memory Mapped IO

However, in some cases more complex input data structures need to be loaded. For instance, indexed files providing efficient access to desired sub-fractions may be loaded by an application. Since the sequential access through streams would impose a mayor performance penalty for accessing such a file, memory mapped IO is offered for read only files.

The abstract type referencing a memory mapped IO file is *mmfile*. The following operations are supported on those:

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

Opening and closing memory mapped files is a global operation throughout the system. Once a file is opened, it is available within the address spaces of all runtime system processes, although not necessarily at the same address range. The task migration makes sure that references to such files are adapted accordingly whenever a task is migrated between nodes.

Memory mapped IO is mainly considered a facility for special use cases in the construction of efficient data structures within the AllScale User API layer. An example is the static graph structure of a mesh, as it is required by one of the pilot applications. While it might also be utilized by the end user, it will always be strictly limited to read-only use cases. Write operations are restricted to the steam based IO API.

2.2.4 AllScale User API

The generic nature of the Core API exceeds the complexity which could be effectively handled by domain experts for implementing parallel algorithms. Thus, it is the objective of the AllScale User API layer to provide a set of more user-friendly constructs for the composition of parallel applications. The implementation of those constructs are carried out by high-performance and C++ experts utilizing the primitives offered by the Core API. An example for this approach has been provided by Section 2.2.3.1 covering the implementation of a generic parallel loop.

The list of constructs covered by the AllScale User API comprises:

Copyright © AllScale Consortium Partners 2017

- parallel control flow primitives:
  - parallel loops with support for fine-grained dependency
    - over numerical ranges (e.g. 1 – 10)
    - over arbitrary ranges defined by C++ iterators
  - parallel reductions as an extension to parallel loops
  - a stencil API utilizing a recursive space-time decomposition schema
  - an adaptive grid refinement stencil as an extension to the standard stencil
- data structures:
  - multi-dimensional static and dynamically sized grids (utilized by iPIC3D and AMDADOS)
  - an adaptive refine-able grid (utilized by AMDADOS)
  - an unstructured multi-grid mesh (utilized by Fine/Open)

All of those are solely based on the constructs of the AllScale Core API and standard C++ features and are thus portable among different AllScale API implementations.

Since the User API layer is open for future extensions, the given list of example constructs comprises only those explicitly required in order to meet the projects objectives. Additional operators for other types of parallel applications, including e.g. branch and bound or MapReduce use cases may be implemented during the course of the development and tuning of the AllScale Environment.

2.3 AllScale Pilot Applications

In the context of the AllScale project, the AllScale User API is developed in particular to support three pilot applications. Those applications have been specifically chosen due to their high demand of computational resources, combined with dynamic load management requirements, reasonable scalability potential and scientific relevance. A general description of those application, as well as an assessment of their characteristics can be found in chapter 4 of Deliverable D2.1.

Within this section the architecture of the AllScale implementations of those prototypes are outlined. Thereby, the focus is placed in particular on the interaction between those pilot applications and the AllScale API.

As a general observation, all pilots are built around a central data structure on which properly orchestrated concurrent updates are iteratively applied. However, due to those updates, the underlying data structure may be gradually altered such that the necessary computational workload required for updating individual partitions of the data structure may significantly vary over time.

The following subsections summarize the underlying data structure as well as the necessary update operations for the individual pilots. Furthermore, the implementation of those constructs based on the AllScale API is outlined.

Copyright © AllScale Consortium Partners 2017

2.3.1 iPIC3D: Implicit Particle-in-Cell Code

The iPIC3D pilot application is a Particle-in-Cell code. As such, its underlying data structure is given by a three-dimensional, regular, equidistant grid where each cell maintains a dynamically sized list of particles. For each particle its species (e.g. electron or proton), and physical properties (location, velocity, charge and mass) is stored. The cells within the grid are partitioning the three-dimensional space in equally sized sub-regions. As an invariant, each particle stored within a cell has to be located within the sub-region represented by the cell.

In each iteration of the simulation, the physical effects of the simulated particles are aggregated to compute a set of induced force fields. Those fields are described by their strength on the corner nodes of the grid structure. In a next step, the forces affecting each particle due to the aggregated fields are computed and a resulting acceleration obtained. This acceleration is utilized to update the velocity and location of each individual particle. To preserve the particle location invariant, particles moving beyond the boundary of a cell need to be migrated. Once the migration of particles is completed, the next iteration can be computed.

The simulation is set up such that particles may never move fast enough to skip a full cell over the duration of a single time step (=iteration step). This property is effectively restricting communication patterns, such that e.g. regions that are n cells apart may differ in their simulation time by up to n time steps. It also localizes communication since particles may only be exchanged between adjacent cells.

<u>Technical Realization</u>

The iPIC3D prototype utilizes two main data structures: three-dimensional grids and particle lists. For both data structures it is necessary to provide the possibility of applying parallel operations upon, to exploit the inherent scalability of the application. Thus, for both, implementations fitting the data item concept imposed by the AllScale Core API have to be provided. Since regular n-dimensional grids and lists are general concepts, both of them are developed as part of the AllScale User API in the form of generic container-like data structures.

Those data structures are utilized to represent the grid, its particles, and the induced fields within the prototype. On top of those, parallel update operations are supported utilizing e.g. higher-dimensional variations of the pfor construct outlined within the API section above. Thus, the resulting simulation code is structured like a list of update loops, enclosed within a single time step loop (similar to the example covered in Section 4).

The algorithm utilized for solving the field equations after aggregating the effects of all particles within the simulation remains exchangeable within this pilot implementation. The alternative solvers represent different trade-offs between communication demands and the length of a simulation time step – and thus the number of iterations required for simulating a given physical process. For field solver algorithms exhibiting similar localized communication properties than the rest of the application, a recursive space-time division may be attempted. This

Copyright © AllScale Consortium Partners 2017

would, in theory, maximize the computational load per memory unit transferred through the memory hierarchy and thus lead to highest resource efficiency. This space-time decomposition will be supported by another User API primitive.

Main Challenge

The main challenge imposed by the iPIC3D pilot is its dynamic load. Large groups of particles are likely to dynamically form clusters. In fact, the resulting application is intended to study this kind of clustering phenomena. In those cases, the occupied cells exhibit orders of magnitudes higher computational load than relatively empty cells. The AllScale Environment is challenged to realize the necessary load balancing capability on an inter-node level to be able to distribute the load evenly among the available computational resources, while dynamically adapting upon the continuously mutating distribution of particles within the simulation.

More details on the AllScale implementation of this pilot application can be found in Deliverable D6.2.

2.3.2 FINE™/OPEN: Unstructured CFD solver

The FINE™/OPEN prototype is a computational fluid dynamics (CFD) solver. The underlying data structure is a static, unstructured mesh comprising objects like cells, faces, edges, nodes, or boundary faces. The geometric information is covered by a list of relations connecting those objects with each other (e.g. a relation relating a cell to its faces). Furthermore, for each object, a set of properties influencing the simulation is maintained. Those may comprise static information like e.g. the volume of a cell, the spatial location of a node or the conductivity of a face. However, it may also comprise dynamic information like the pressure within cells or the heat flow through a face. The latter is the state of the conducted simulation and the result end users are interested in. Finally, to aid the effective computation of the desired solution, multiple meshes describing the same objects in different resolutions are combined into a hierarchy of meshes to facilitate the application of a multigrid solver approach. Thus, the full data structure is a hierarchy of meshes, where each mesh comprises various sets of objects, each linked through geometric relations and associated with a set of static and dynamic properties.

In each simulation step, updates to the various properties associated to the mesh objects are conducted. Updates start in the mesh layer exhibiting the finest resolution. Thereby, physical effects are propagated through the connections between the various objects on this layer. After a fixed number of iterations, the current state of the simulated properties are aggregated and projected to the next coarser grained level of the hierarchical mesh. There, the same propagation and aggregation operations are repeated. After completing updates on the coarsest layer, modifications are projected recursively down towards the finer layers and the update starts over again for the next time step.

All updates are thereby local as defined by the relations formed over the objects on the various mesh layers. Since the meshes represent real world physical

Copyright © AllScale Consortium Partners 2017

structures, the resulting updates are representing spatially local interactions. The key for the realization of an efficient distributed implementation of this pilot application is to minimize data transfers. This is achieved through the partitioning of the meshes such that to a high degree updates can be conducted locally, minimizing the need of data exchanges.

Technical Realization

The underlying mesh data structure required by this prototype has to be implemented such that it meets the requirements imposed by the data item concept of the AllScale Core API. In particular, it has to provide support for partitioning. Meshes need to be decomposable into sub-meshes dynamically, to be distributed among the available resources. This decomposition has to be realized such that the necessary interaction between nodes is minimized.

To obtain an efficient, close to optimal decomposition we exploit the static nature of the simulated mesh. The partitioning of the mesh is computed offline, in the form of a pre-processing step. The resulting decomposed meshes, including their information regarding their boundary regions and closures, is incorporated into the data structure utilized during the actual simulation – and thus not required to be computed while running the time-critical part of the solver. The resulting available static information is those available during runtime for realizing data exchange and migration operations.

The provided mesh data structure is designed generically to facilitate an arbitrary list of objects, relations, hierarchy levels, and properties. Thus, future use cases demanding modified mesh variants are implicitly supported by the implementation developed for this pilot. While the generic mesh and basic operations are part of the User API as a general facility, the specific instantiation utilized by the pilot and IO operators are part of the pilot application.

Main Challenge

The main challenge imposed by this pilot is the complexity of the underlying data structure. The need for managing an unstructured mesh, required to be dynamically redistributed throughout a system during runtime imposes challenges to all layers of the software stack, in particular to the API which is required to implement all the necessary data item operations. Furthermore, the size of the handled meshes, comprising several billion objects, impose algorithmic challenges for the required pre-processing tools.

Another challenge of this prototype, as described in the requirements Deliverable D2.1 is the needed IO support. This problem is to be handled within the pilot implementation by customizing the output format of the simulation to facilitate high performance IO. In particular, the order of values in the output data stream as well as the actual file structure (one or multiple files, e.g. one per process) are addressed to that end. By relaxing those constraints, the IO performance bottleneck is addressed.

More details on the AllScale implementation of this pilot application can be found in Deliverable D6.4.

Copyright © AllScale Consortium Partners 2017

2.3.3 AMDADOS: Adaptive Meshing and Data Assimilation

The AMDADOS pilot application is a numerical simulation of an oil spill based on a structured, adaptively refined, regular grid, incorporating data assimilation events. Thus, the main data structure this pilot is based on is a regular, adaptive grid. The number of refinement levels is thereby known during development time and can be hard coded within the application. However, coarsening and refinement steps are applied dynamically during runtime based on the state of the simulation as well as data assimilation events.

The refinement of the resolution follows a hierarchical pattern. On the top level, a fixed size, regular 2D grid defines the domain of the overall simulated area. Each of those top-level cells (aka sub-domains) may then be itself recursively sub-divided into small regular grids, up to a statically fixed maximum resolution.

The simulation algorithm is updating each sub-domain independently for one time step at the currently active level of resolution. This update operation may take several iterations, yet does not necessitate the exchange of any information with any other sub-domain. Once completed, boundary information needs to be exchanged between adjacent sub-domains utilizing another iterative algorithm. This algorithm only requires the localized synchronization between neighboring sub-domains. Thus, sub-domains being n global cell-widths apart may be n time steps apart in their simulation time.

The assimilation of data is an optional step after the completion of an update of a sub-domain. In this case, the solution obtained for the processed sub-domain is combined with some externally obtained measurement before the simulation continues with the mutual exchange of information among adjacent cells and the next simulation time step.

An assimilation operation, however, is orders of magnitudes more complex than a mere simulation time step for the same sub-domain. Thus, assimilations are triggering load imbalance that has to be dealt with.

Technical Realization

The basic data structure is facilitated by an adaptive grid structure following the constraints imposed by the API's data item concept. This adaptive grid is developed as part of the AllScale User API and shares its implementation to a large extend with a regular grid, as it is utilized by other pilots. The update operations on the grid may be implemented utilizing pfor loop structures, yet this pilot is especially eligible for applying recursive space-time decomposition to enable the concurrent computation of multiple time steps on spatially sufficiently separated sub domains. The corresponding operator for this step is offered by the AllScale User API, in a similar generic way as the pfor operator.

Main Challenge

The main challenge of this pilot is the handling of the dynamic changes in the imposed load. Data assimilation steps are sporadic events that can cause unusual

Copyright © AllScale Consortium Partners 2017

high load for a single update operation. Furthermore, the gradual adaptation of the grid resolution is continuously altering the distribution of computational load throughout the simulated domain.

The design of the AllScale Environment, which is based on over-provisioning tasks for the available resources, mitigates the effects of load imbalance between individual tasks in the short term. Long term effects of shifting load can be handled due to the capability of dynamically reassigning data fragments – and thus ownership to domains – among the participating compute nodes. Finally, due to the native support of nested parallelism, the computation of an assimilation step can itself be recursively broken apart and distributed e.g. among the cores of a node to further mitigate the impact on the load balance of such events. In combination, those three techniques should provide the substrate for an efficient processing of this pilot.

More details on the AllScale implementation of this pilot application can be found in Deliverable D6.6.

2.4 AllScale Compiler

Codes implemented utilizing the AllScale API can be compiled by standard C++ tools and executed on parallel, shared memory machines. This mode is intended to be used during the development and debugging phases of an application development project. However, for obtaining an extreme scale high-performance version that benefits from all the novel services offered by the AllScale Environment, the same code base has to be compiled by the API-aware AllScale Compiler. Unlike standard C++ compilers, the AllScale Compiler is aware of the interpretation of the parallel primitives offered by the Core API layer and restructures the application by introducing additional versions of encountered parallel code fragments. Each version targets a different architecture and/or represent a different trade-off between multiple optimization objectives (e.g. more parallelism vs. less overhead). Together with information describing the specific traits of those versions as well as details regarding their (data) dependencies, the resulting set of implementations are forwarded to the AllScale runtime system (Section 2.5). The latter dynamically auto-tunes the program for adjustable objectives to achieve a desired trade-off among the considered tuning objectives.

The implementation of the AllScale Compiler is based on the Insieme source-to-source compiler infrastructure.

2.4.1 Overview

Figure 2 outlines the internal organization of the AllScale Compiler. In a first step a given input code (1) utilizing the AllScale API is parsed by a clang based C++ frontend. During this process, the C++ code templates utilized by the User-Level API for building user-friendly, domain-specific APIs is unfolded and instantiated. As a result their concrete instantiation becomes accessible to the second step, namely the semantic frontend. In this step, the instantiated, concrete program codes (2) are converted to a high-level, explicit parallel intermediate representation (3). Unlike conventional ASTs mirroring the original input language, the design of this intermediate representation (IR) is focus on a

Copyright © AllScale Consortium Partners 2017

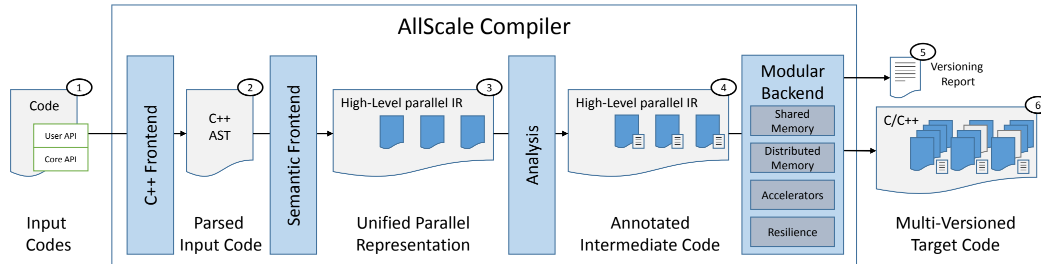


Figure 2. Overview AllScale Compiler Architecture

concise, minimal set of language constructs enabling the development of sophisticated analysis and transformations. Furthermore, unlike C++ or most other IRs, this IR provides explicit constructs to model parallel control flows and their synchronization. Also, the representation comprises the entire program, thus providing a global view exceeding the conventional limits of individual translation units. Based on this IR, in the third compilation step, parallel tasks are analyzed to determine their data requirements (4). These requirements describe e.g. the data that needs to be transferred – along with a task – to a different address space for the task to be successfully performed remotely, as well as the data to be communicated to any code fragment subsequently accessing the produced data. This abstract data requirement description is exposed to the runtime in the form of a function computing the data requirements of a concrete task. It enables the runtime to manage corresponding data migration operations between nodes of a cluster, or to move data to and from device memory associated to accelerators. Also, to improve the application's resilience, required data may be moved to and from persistent storage devices to facilitate checkpointing.

Furthermore, the compiler's modular backend generates code versions targeting different architectures (CPU, accelerators), as well as different performance trait-off objectives (e.g. degree of parallelism). However, the compiler may encounter limitations preventing it from being able to obtain accurate enough data requirements and/or generating desired code versions. To provide feedback regarding encountered obstacles, a report (5) summarizing failed, as well as successful code analysis and generation steps, is produced to aid debugging and code optimization to be undertaken by the software developer. Finally, all the available versions of the tasks are combined and encoded into a single C++ output code together with their meta-information (6). The runtime system may then flexibly choose among them.

2.4.2 Compiler Infrastructure

The internal representation (IR) of the Insieme compiler (Inspire) is an explicit parallel high-level IR for C/C++ enabling the analysis and transformation of C++ applications exhibiting high-level constructs, including generic types and operators.

Copyright © AllScale Consortium Partners 2017

For the AllScale project, Insieme's IR is extended by a module[2] capable of representing the primitives of the AllScale Core API, in particular the generic prec operator, the associated treeture and tref ADT and the manageable data structure concept. This modular IR extension provides, according to the design of the AllScale Compiler, the foundation of program analysis, transformation, and code generation steps.

The following modifications are required:

- a IR module modelling the AllScale Core operators
- a frontend extension identifying the corresponding constructs in the input code and converting them into the compiler's intermediate representation
- an extension to the analysis framework enabling the correct interpretation of the newly introduced language extensions
- a backend extension to synthesize code fitting the runtime interface's requirements in a first development step; in a second phase, the generation of customized code versions addressing various desired objectives (degree of parallelism, target architecture, resilience, instrumentation, …) are added incrementally

All those module extensions are integrated with the Insieme framework to produce a compiler executable facilitating the compilation of input applications into binaries utilizing the AllScale Runtime System for managing their execution on target systems. The combined AllScale toolchain should thereby serve as a complete drop-in replacement of a standard GCC or clang based toolchain utilized by a development or build system.

2.4.3 Data Requirement Analysis

Given a code fragment encoded within Insieme's parallel intermediate representation extended by the AllScale IR module, an analysis step symbolically determine the data requirements of the given fragment. For the code fragment computing

```
for(int i = a .. b ) {
        A[i] = B[i-1] + B[i] + B[i+1];
}
```

the analysis determines the following requirements:

- the array A is written for the range [a..b]
- the array B is read for the range [a-1…b+1]

This symbolic representation, parameterized by the free variables A, B, a, and b of the code fragment, is obtained by the data requirement analysis within the compiler. Within the backend of the compiler, this symbolic formula is converted into a function capable to compute the actual data dependencies based on the concrete values of the free variables. This function is then offered to the runtime system to manage the distribution of data and tasks throughout the system.

---

[2] A module in the Insieme infrastructure is essentially the definition of a set of abstract data types and their associated operations.

Copyright © AllScale Consortium Partners 2017

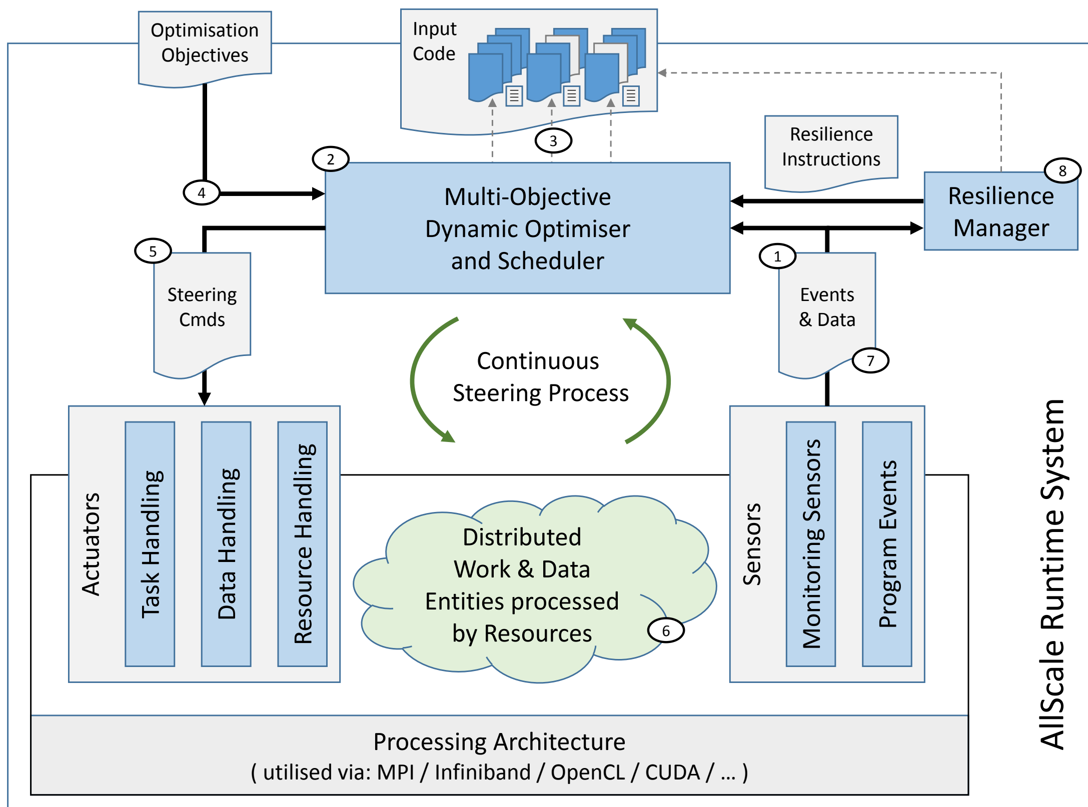


Figure 3. Overview AllScale Runtime System Architecture

The example above is a simple example to illustrate the general approach. A more realistic case is the recursive pfor implementation as covered in Section 2.2.3.1. In this case the data requirements of some sub-task has to be deduced from the captured values and the recursive range parameters. Details of this analysis, the internal IR modifications, and the backend code generation are covered in the corresponding Deliverables D3.3 and D3.4.

The more accurate those access patterns can be deduced, the more fine-granular data requirements can be forwarded to the runtime system. In cases where the analysis fails to obtain sufficiently accurate results, an issue is reported to the end-user utilizing the AllScale Compiler, indicating a problem a human user may be able to resolve e.g. by restructuring the code or adding hints through additional assertions.

2.5 AllScale Runtime System

2.5.1 Overview

The resulting C++ output code is compiled by platform specific C++ compilers and linked against the AllScale Runtime System. Unlike conventional programs orchestrating their own execution, in the AllScale Environment the runtime system steers the execution of the resulting program, while the input program offers execution options. Figure 3 outlines this process.

The central element of the runtime system is the dynamic optimizer and scheduler: it is the component that steers the execution. Triggered by program

Copyright © AllScale Consortium Partners 2017

events (1) like task spawning, suspension or completion, the scheduler conducts application managing decisions (2). The available options comprise those offered by the input program generated by the compiler (3), as well as the settings of any hardware-level actuators provided by the target platform, such as frequency scaling. Each task version provided by the compiler is annotated with requirements and cost models supporting the optimizer in its evaluation. Based on dynamically customizable optimization objectives (4) – like focusing on execution time or energy savings – the scheduler issues corresponding steering commands (5) to the actuators influencing the execution. These actuators cover task, data and resource management operations. Thus, the optimizer may alter the assignment of tasks to processors, the location of data elements in the system, and hardware parameters such as e.g. the frequency of cores. The actual task execution (6) is monitored and performance data is collected (7) which may be utilized subsequently as input for future iterations of the runtime system control loop. Furthermore, a resilience management component (8) monitors the state of the processing application to discover irregularities. Based on this monitoring, the manager may suggest corresponding precautionary measures (e.g. the creation of local checkpoints) as well as recovery operations to the dynamic optimizer. The scheduler is covered by Section 2.6, the monitoring system by Section 2.7, and the resilience management by Section 2.8. Due to their architectural position, those three components are referred to as runtime subsystems.

Note that, while for simplicity Figure 3 represents the runtime system and its control loop as a centralized system, the AllScale Environment utilizes distributed, scalable scheduling and management solutions. Specifically due to the scalability and minimal induced system performance noise requirements of the scheduler and the monitoring service, the design targets distributed implementations to offer scalable solutions for those components.

2.5.2 Runtime Hardware and Application Model

The foundation of the runtime system's design is based on two models abstracting its environment:

- the hardware model, describing the structure of the compute infrastructure an application is processed on, and
- the application model, describing the structure of the processed application

It is the runtime's job to utilize the information it maintains about the available hardware to map the computation and data described by the application model to the available resources according to some user-specified objectives.

2.5.2.1 The Hardware Model

The hardware model utilized by the AllScale Runtime System provides a common abstraction of the available hardware resources to be utilized and managed by the various runtime subsystems. It those provides a common way of addressing those resources throughout the system.

The hardware model comprises the following entities:

Copyright © AllScale Consortium Partners 2017

- *Node:* the unit of resource managed by a single AllScale process; each node has an associated locality, which can be addressed through the network of AllScale processes; also, every node has its own instances of runtime-subsystems (e.g. scheduler or resilience manager);

- *Compute Unit:* a hardware unit capable of processing work (see work item below); a CPU core or an accelerator are, for instance, represented as compute units; each compute unit belongs to a single node

- *Memory Unit:* a memory unit is the abstraction of an address space where each addressable memory location exhibits the same access behavior as any other location within the same space; for instance, NuMA nodes or GPU device memory are represented as memory units; each memory unit belongs to a single node

- *Links:* links connect compute and memory units; a link between a compute unit A and a memory unit B indicates that code running on A can access and manipulate data stored in memory location B.

Additionally we define a *machine* to be the term utilized for referencing a network of resources managed by a connected network of AllScale Runtime System process instances. Thus, the term machine references e.g. the part of a compute cluster provisioned for the execution of an AllScale application.

While application code running on a compute unit A may only access data located within directly linked memory locations, the runtime system is capable of moving data between any pair of memory locations. Thus, the runtime system is responsible for actively managing the workload and data distribution such that the data required by processed tasks is always located within accessible memory units.

An instance of the hardware model can be considered as a graph like structure, where nodes are given by compute and memory units and edges are formed through links. A concrete instance of the hardware model is referred to as a *hardware environment*. However, it is important to note that the hardware environment is not static over the course of the execution of an application. Nodes, links, compute or memory units may be added or removed dynamically – active, due to the management of resources by the runtime, or passively due to failures or external administrative operations.

For instance, to save energy the runtime may decide to shut down a number of cores during a particular phase of the program execution. This is effecting the hardware environment by removing the corresponding compute units. Failing nodes or nodes disconnected for maintenance as well as rejoining nodes after repair operations are as well altering the hardware environment.

Non-Functional Information

The hardware model only specifies the most basic information regarding the available hardware resources necessary for successfully processing applications. However, additional, non-functional information like the (current) clock rate of

Copyright © AllScale Consortium Partners 2017

compute units, lengths of work queues associated to compute units, the size and occupation of memory locations, or the capacity of links are included to aid the decision making processes within runtime subsystems. The runtime system provides the framework for representing, distributing and querying this kind of information. The actual kind of information to be maintained, however, is adapted as demanded over the course of the development of the project.

2.5.2.2 The Application Model

The information about the hardware environment is utilized by the runtime system to distribute and balance a running application among the available resources. To that end, similar to the hardware model, an abstraction of the managed application is required. This abstraction is given by the application model.

The application model describes applications governed by the AllScale Runtime System. The two central components are:

- *Work Item:* the entity representing work that is utilized by the scheduler to distribute computational load among available compute units
- *Data Item:* the entity representing data that is utilized by the runtime scheduler to distribute data among available memory units

Work items represent a given amount of work, e.g. the update of a certain set of elements or the inspection of a set of configurations in a search. Data items on the other hand represent data structures like arrays, trees or meshes.

In general, both, work and data items, are decomposable into partitions. Thus, a work item may be decomposed (up to a certain atomic granularity) into a set of smaller sub-items orchestrated to perform the same computation as the original item. Similar, data items may be partitioned into smaller structures to be distributed among multiple memory units.

The connection between work and data items is formed by data requirement functions associated to each work item instance. Through those, work items define the data required for their processing as well as the access mode. For instance, a work item copying a range [x,..,y] of elements from a vector A to a vector B states that it requires read access to the elements A[x] … A[y] as well as write access to the elements B[x] … B[y]. In this case the runtime is required to assign this work item to a compute unit exhibiting access to the corresponding fractions of the data items A and B. However, to that end, the runtime system may decide to move data or fractions of data as well as to split tasks to obtain a more efficient execution schedule. This scheduling is the main objective of the scheduler subsystem.

Furthermore, dependencies between work items may be formed to synchronize ongoing computation. Thus, work items may be scheduled only upon the completion of other work items. Combined with the data requirements, those dependencies are the only two means of synchronization offered to an AllScale application. Note that only the synchronization of tasks is exposed by the AllScale Core API to the user. Data requirements are extracted from the implementation code by the compiler through static code analysis.

Copyright © AllScale Consortium Partners 2017

Tunning Options

In addition to the data requirements, a function realizing the partitioning of a work item, and a function for processing the work item on some compute unit, each work item may be equipped with a set of alternative implementations. Each of those implementations is functionally equivalent to the function processing the work item. However, those implementations may be tuned for specific scenarios or optimization trade-offs. Some may be prepared for utilizing accelerator hardware, while others may produce varying degrees of nested parallelism, exploit specific hardware available on some systems, or even represent different trade-offs between execution speed and energy efficiency. The runtime is free to choose any of those variants for processing a given work item on a selected compute unit.

The set of alternatives offered to the runtime is not fixed a priory. Future development iterations on the compiler may introduce new code variants to be exploited by the runtime. To facilitate the utilization of those variants, additional meta-information describing the properties of the various code variants are added by the compiler. An example may be versions marked for being utilized on certain GPU types. As for the non-functional information maintained for the hardware environment, the AllScale toolchain components provide support for a generic set of code variants and meta-information properties. The ongoing development of the system will show which code variants and properties are beneficial for enforcing the objectives defined for the execution of a program.

2.5.3 Technological Base

The AllScale Runtime System is based on HPX, extended by the necessary infrastructure for fulfilling the requirements imposed on it. The necessary modifications and adaptations comprise:

- the implementation of the work item infrastructure, comprising means for the specification, creation, addressing, and synchronization of work items; this mainly consisted of mapping those AllScale concepts to equivalent constructs offered by the underlying HPX runtime system

- the implementation of a data item manager enabling the creation, addressing, manipulation, and localization of data items and partitions of those; this service is a new service built on top of the preexisting HPX infrastructure

- the implementation of the hardware model, in particular means for addressing compute and memory units as well as to query their interconnection and non-functional properties

- the logical concentration of management decisions to unite all performance impacting decision-making processes into to a single scheduler implementation; this requires an adaptation of the HPX scheduler to provide access to intra-node scheduling policies, in addition

Copyright © AllScale Consortium Partners 2017

to the inter-node scheduling options to be realized as part of the work item infrastructure

- a framework for the monitoring system supporting the collection, processing, aggregation, subscription and querying of information associated to the various entities in the hardware and application model; this sub-system is an extension of a preexisting HPX internal monitoring service

- an interface for the resilience manager enabling this component to observe the execution state, suggest backup operations and recommend recovery operations upon failures; this component is built on top of the monitoring framework and in cooperation with the scheduler, which it may instruct to perform backup or recovery operations

A particular challenge for the AllScale Runtime System is imposed by its high scalability requirements. To that end, none of aforementioned services and infrastructures may rely on centralized knowledge. Distributed, localized means for realizing the required work and data item operations, monitoring and scheduling services are required.

For work items, the locality of the recursively decomposed tasks themselves are aiding in the design of localized management structures. The likelihood of task imposing synchronization constraints collocated on the same node is much higher than inter-node dependencies due to the structure of the AllScale program model. Those, synchronization on tasks is mostly to be handled within nodes and only in rare cases through information exchange between nodes.

The data items, on the other hand, require a directory maintaining their current distribution state that can be accessed by all nodes throughout the system. However, the support for a hierarchical partitioning of data items, the favoring of localized operations due to the AllScale programming model, and the relatively low number of times updates on the data distribution are performed, provide the foundation for a scalable implementation of such a global directory. Techniques including the distribution of responsibility and local caching are employed to provide a scalable implementation of the data item localization and manipulation operations.

2.5.4 Failure Tolerance

An application can only be as resilient as the components it is based on. Thus, the projects objective to provide a failure resilient execution environment of large scale applications requires the underlying runtime system to be equally resilient. Fundamentally, the communication facilities utilized by the runtime must be capable of dealing with failing nodes. Additionally, all services built on top of those communication primitives must be design such that leaving or joining nodes can be handled.

HPX, the foundation the AllScale Runtime System is based on, provides communication layer implementations resilient against node failures. However, functionality dealing with failing, leaving and joining nodes needs to be

Copyright © AllScale Consortium Partners 2017

reevaluated and adapted. Furthermore, the work and data item management services need to be equipped with means for recovering failures. The corresponding features are closely linked to the resilience manager subsystem and are thus discussed in detail in the corresponding section (Section 2.8). Furthermore, the scheduler and monitoring infrastructure must be design such that individual node failures can be recovered. Details on the techniques employed for ensuring this property are beyond the scope of this architecture deliverable and can be found in the deliverables covering the individual contributions.

2.5.5 Scheduler Interface

One of the key design goals of the AllScale Runtime System is to concentrate all the performance impacting decision making processes into a single component. This component shall be capable of manipulating the work and data distribution of an application throughout the available hardware. Furthermore, it should adjusting the hardware environment and hardware parameters as well as internal runtime settings as, for instance, the granularity of collected monitoring data. The fundamental idea is to have a (logically) central component to conduct research on how to most effectively manage the execution of AllScale applications.

The purpose of the other runtime subsystems, as well as the surrounding runtime system implementation itself, is to provide the necessary level of abstraction to the scheduler component. Scalable, efficient operations to inspect and influence work and data items, hardware parameters, and general runtime system behavior shall be offered through corresponding interfaces.

Those operations include:

- callback operations whenever a work item needs to be scheduled
- means to run a selected work item variant on a selected compute unit
- means to examine the data requirements of a work item
- means to query and manipulate the distribution of data items
- means to query the system state, e.g. queue lengths
- means to communicate between scheduler instances of different nodes
- means to adapt hardware parameters
- callbacks upon the discovery of leaving and joining nodes
- means to obtain information regarding possible recovery options

Those extensive capabilities of the scheduler component make this element the central system responsible for the dynamic optimization and tuning of running applications.

2.5.6 Resilience Manager Interface

The resilience manager within the runtime system architecture is designed to observe the ongoing computation and produce suggestions on backup and recovery operations to be forwarded to the scheduler. For the observation of the system it requires access to the monitoring system. This access enables it to

Copyright © AllScale Consortium Partners 2017

observe the state of the application and the hardware components throughout the system. Based on those observations suggestions on when to conduct backup operations are fed to the scheduler. Furthermore, in case of a failure state it is the responsibility of the resilience manager to provide suggestions on recovery options to the scheduler.

Thus, the interface of the resilience manager involves:

- access to the monitoring system to continuously observe the state of the system
- the possibility to register callbacks to certain events throughout the system, e.g. the start / end of a task or an event triggered upon a node failure
- means to communicate with resilience manager instances on other nodes throughout the system

The resilience manager has to provide:

- an interface to the scheduler enabling the scheduler to request recovery operations to compensate for failures

While the AllScale Runtime System component is establishing those interfaces, it is the obligation of the work package associated to the corresponding components to provide the actual implementations.

2.5.7 Monitoring Service Interface

The monitoring subsystem is responsible for collecting, aggregating, distributing and maintaining information about the various entities throughout the system. It is based on an HPX internal information service, capable of forwarding data. However, in addition it has to provide options to selectively enable and disable the collection of performance data to support the active minimization of its performance impact. Furthermore, the monitoring service has to provide the possibility to register call back events triggered upon the occurrence of certain conditions.

The monitoring service therefore requires:

- access to the HPX monitoring service
- means to schedule data aggregation and maintenance tasks

The monitoring service has to provide:

- an interface for querying (aggregated) information about the various entities observed throughout the system
- means to subscribe for events
- means to customize the kind and granularity of data to be collected

The main task of the monitoring component is to provide implementations of those interfaces and to populate the list of observed properties with information relevant for the subsystems depending on those.

Copyright © AllScale Consortium Partners 2017

2.6 AllScale Scheduler

As has been outlined within Section 2.5.5, the scheduler component is the central component when it comes to decision making processes in the runtime system. Among others, it decides:

- where to place data
- where to process work
- how to set up the underlying hardware infrastructure
- what data should be collected and aggregated by the monitoring system

All of those decisions have to be made towards a user given object. This objective is given through a policy defining the tradeoff among objectives including the total execution time, the resource usage, and the energy consumption.

Objective Function

While in theory a multi-objective optimization process yields a set of optimal configurations (aka Pareto frontier), within the context of the runtime system a single configuration out of those optimal configurations has to be selected since the application is only processed once. To select among those options, a tie breaking weighting function is employed. The system seeks to minimize the objective function

$$t^n e^m r^k$$

where t is the total execution time, e the total energy requirement, and r the total resource usage[3]. The constants $0 \leq n, m, k \leq 2$ are chosen by the user to weight the various objectives. For instance, if $n = 1, m = k = 0$ the objective function is reduced to the execution time only. Thus, the runtime system would seek to process the application as fast as possible. On the other hand, by choosing $m = 1, n = k = 0$ the system would make its decisions such that as little energy as possible is consumed. Settings where more than one parameter is greater than zero result in trade-off solutions among multiple objectives.

To simplify the specification of the objective function, predefined parameter settings are offered to the end user (e.g. as_fast_as_possible or balanced_time_and_energy). The evaluation and investigation of practical use cases will provide the necessary default values for the three parameters. Furthermore, the runtime provides means to alter the user defined objective during runtime. So changed conditions resulting in a change of the objectives can be forwarded to the runtime system.

Scheduler Design

Beside the efficiency of the execution controlled by the scheduler, its own efficiency in deriving those scheduling decisions is an important element to be considered in the design of the scheduler. A frequent, high delay in scheduling decisions due to inter-node communication or complex computation steps is

[3] Technically this makes the optimization problem a mono-objective optimization problem; however, since dealing with multiple objectives and conducting the tie break during runtime, this is known as a multi-objective optimization problem in the context of code optimization

Copyright © AllScale Consortium Partners 2017

negatively effecting the overall execution performance of the application. Thus, a two-layer scheduler architecture has been designed:

- a top layer, ***strategic scheduler*** is periodically evaluating the utilization of the available hardware and deciding upon the adding, removal, or reconfiguration of the available hardware; while each node has its own strategic scheduler instance, decisions are made globally due to the employment of distributed algorithms; this layer works asynchronously, independent of the events within the processed application; it is also the layer interpreting the objective function given by the user

- a bottom layer, ***tactical scheduler*** is utilizing the resources provisioned by the strategic scheduler as efficiently as possible for processing work items; it is the layer determining whether to split or process work items, selecting which work item implementation variant is to be processed by which compute unit under its control, and moving data item fragments as needed; each node has its own tactical scheduler, mostly working independently of the scheduler of other nodes with the exception of e.g. localized work stealing events to facilitate automated inter-node load balancing;

Both layers of the scheduler constitute the implementation of the scheduler component. The separation of concerns serves the goal of providing fast scheduling decisions while considering global, long-term scheduling objectives.

Naturally, due to the required resilience towards node failures, the communication protocol between scheduler instances of different nodes has to be designed stateless. The details of those protocols, as well as the details of the inner structure of the scheduling layers is covered in the corresponding Deliverable D4.6 and D4.7.

2.7 AllScale Monitoring Service

The AllScale Monitoring Service is designed to be an extension of the monitoring service implementation included within HPX, the system the AllScale Runtime System is based on. However, the following modifications and extensions are necessary:

- means to collect hardware information, including the hardware configuration (e.g. current DFVS setup) or performance counters need to be added; the foundation is provided by the PAPI library, however attributing collected data to the corresponding resources remains to be the responsibility of the AllScale Monitoring Service;
- means to collect information on various runtime entities need to be integrated; e.g. execution time of work items, work queue lengths, energy usage of a work item, allocated memory within a memory unit; corresponding hooks and access points have to be integrated into HPX and the AllScale Runtime System to enable those observations
- means to subscribe and unsubscribe to events need to be provided
- means to query for information need to be provided

Copyright © AllScale Consortium Partners 2017

- means to control the kind and granularity of collected data
- functionality filtering, aggregating, compressing, and storing monitoring data need to be integrated

While most of this functionality serves the purpose of online-monitoring to support decision making processes, the monitoring system is furthermore required to provide the foundation of post mortem analysis. Thus, the necessary data formats, recording, extraction, and analysis utilities for investigating application executions are required. While of interest for the end user for investigating the characteristic of the processed application, this toolbox serves as well as an important aid for the development process of the AllScale toolchain itself. Additional details regarding the design and implementation of the required features are covered within Deliverable D5.2 and D5.3.

2.8 AllScale Resilience Manager

The resilience manager is responsible for providing recovery options to the scheduler upon failures. Such failures may range from bit flips in memory, over faulty computations in the cores, to whole node or even network failures. However, the requirement analysis conducted during the initial phase of the project (see Deliverable D5.1) revealed the lack of potential fine-grained recovery strategies for soft failures like bit flips. Furthermore the unavailability of hardware capable of identifying such problems has been stated as another major obstacle for the realization of such failure recovery schemas. Thus, the focus in the design of the resilience manager was placed on the much more probable case of node failures in extreme scale systems – which may be caused by actual hardware failures or failures leading to AllScale processes to terminate their execution unexpectedly.

Integrating Resilience

The design of the AllScale Resilience Manager aims at the handling of node failures transparently to the end user. It utilizes the runtime system's application model as a foundation and aims on guaranteeing the proper execution of every work item according to the involved work and data item dependencies.

The basic idea is to have for every node A a protector node B who maintains a backup of the work items currently processed by node A. Whenever node A gets a work item assigned for processing, it informs node B about this newly gained responsibility. Likewise, whenever A finishes a work item, it informs B about this fulfilled obligation. Thus, in case node A crashes, node B knows about the tasks being processed on A and restarts those tasks. By arranging all nodes in long enough cycles (not necessarily in a single one), failures of individual nodes can be recovered through this approach. Furthermore, if deemed necessary, the protocol can be extended to guarantee the survival of even higher rates of failures by e.g. utilizing two protectors per node.

One of the major challenges of this approach is the amount of necessary communication between a node and its protector. However, a property of the AllScale Runtime System application model can help to solve this problem. To avoid updating a node's protector node upon every single processed work item, the hierarchical relation between work items can be exploited to significantly reduce the number of necessary updates.

Copyright © AllScale Consortium Partners 2017

Whenever a new work item is assigned to a node, it can locally check whether its protector knows already about this responsibility due to a responsibility towards one of the parent work items already previously reported. In such a case the new responsibility is not required to be reported to the protector since the previously reported responsibility towards the larger work item includes the responsibility of the new work item. Since most work items processed within a node are likely to be descendants of a rather small set of work items, this significantly reduces the number of necessary updates.

Thus, the recursive partitioning of workload inherent in the AllScale application model provides means to effectively keep track of the obligations of the various nodes throughout the system. By utilizing a distributed scheme for keeping backups information exchanges are furthermore localized.

<u>Resilience Protocol Details:</u>

The overview description provided above is a high-level description of the employed algorithm. Various additional problems need to be solved:

- Before restarting tasks known to be processed by a failed node, the protector node has to reset the initial state that has been in place before a potential first execution of those work items. Since the progress of the work item evaluation is unknown, globally accessible data may have been partially updated. Data may also have been lost since it has been located on the failed node. Thus, the backup of a task on the protector node is not only comprising the parameters describing the task but also a snapshot of the data items read by the backed up task. In case of a failure event, this information is used to restore the initial state of the task and its input data before restarting the task.
  The backup and restoration of input data can be derived from the data requirements associated to the work item. This information, originally included for the scheduling process, can be used to back up and restore all the necessary information. In the case of data items lost due to node failures, the restoration of the task operating on those data fragments implicitly restores them as well.

- The naïve restart of tasks can lead to race conditions and/or duplicated updates due to the same task being processed twice. This may happen if a sub-task of a backed up task gets stolen to another node. This node would not notice that the node the task got stolen from crashed. Thus, it also is not aware that the task it is processing is going to be processed again. To eliminate this problem, before a work item gets reissued it has to be ensured that no child-task of the recovered work item is running anywhere else in the system. This might be realized simply through a purge broadcast or, more sophisticated, by tracking steal operations and keeping temporary records on nodes who have stolen sub tasks. Independently of the actual realization, this additional step in the recovery protocol ensures that each task is effectively only processed once.

Like the scheduler and monitoring subsystem, the resilience subsystem has to be designed keeping its own state resilient to node failures. Thus, the creation of links connecting nodes to their protectors has to be failure resistant. For instance, upon node failures, the node protected by the failed node has to obtain a new protector.

Copyright © AllScale Consortium Partners 2017

Backup Storage

The designed resilience system does not specify the media utilized for backing up information – nor does the protected application need to know about this details (in fact, the entire application resilience remains transparent to the application developer). Consequently, the backup of work items and their data environment may be conducted within the memory of the protector node, one of its local discs, any special, non-volatile memory hardware resource, or any other kind of suitable storage device. The backup and recovery protocol can be adapted to the target architecture without changing the original application code. By design, it may even be adjusted at runtime.

Failure Detection

A final piece for the design of the resilience manager is the need for a method to identify node failures. This method touches the responsibility of three different components: naturally the resilience manager, the monitoring service due to its responsibility of collecting data about the system state, and the runtime system itself since it provides the underlying communication infrastructure and is probably the first to notice.

Since for the overall design of the system the actual implementation is less important than the mere fact that such a method is present, the realization remains exchangeable. The current design foresees a simple heartbeat signal between a protector and its protected node. If this signal detects a failed node, an event in the monitoring system signaling a node failure is triggered and – through a corresponding subscription – forward to the resident resilience manager.

3 AllScale Component Interfaces

This section briefly summarizes the interfaces present within the various AllScale components. Since many of the referenced interfaces have already been discussed in Section 2, this section is a mere summary of the interfaces between components.

Application to API Interface

The API itself constitutes the interface between AllScale applications and the underlying AllScale Environment. However, the boundary between what belongs to an application and what is part of the user-level API is an open boundary. Similar to the design decision of what functionality should be placed in a general library like the STL or boost library, and what should remain application code specific, this is mostly a semantic and/or technical issue. Within this project we tend to move generic utilities that may be utilized in more than a single specific use case into the user level API, while all other codes shall remain with the application code. Furthermore, the user API is not closed. It may be extended and additional third-party libraries may be built on top of it or the AllScale Core API. However, the one crucial constraint in all of those implementations is to not violate the constraints imposed by the AllScale API regarding the utilization of

Copyright © AllScale Consortium Partners 2017

external resources (e.g. IO) or parallel constructs (e.g. locks, pthreads, OpenMP or MPI constructs).

API to Compiler Interface

The contract between the API and the compiler is defined by the AllScale Core API, for which in principle the compiler provides just another implementation. This set of constructs comprises the prec operator and the associated treetures (Section 2.2.3.1), the data item concepts (2.2.3.2), and the IO primitives (2.2.3.3).

Compiler to Runtime Interface

The interface between the compiler and the runtime is defined through the AllScale Runtime System application model as outlined by Section 2.5.2.2, mainly comprising work and data items. It is the compilers objective to generate code satisfying the requirements specified by those two concepts to realize parallel operations as intended by the code expressed by the application developer.

Monitoring Service Interface

As a runtime sub-system the monitoring service provides efficient means for other subsystems to collect, aggregate, relay and query information regarding the state of a processed application and the hardware it is executed on. Thus, interfaces for adding sensors and aggregators are required, as well as a query and subscription API. Details have been covered in Section 2.5.6. Additionally, the monitoring service is required to provide means for the end user (from the runtime systems perspective) to enable the inspection of the execution of an AllScale application for performance debugging. The corresponding tool and visualization support are covered in Deliverable D5.2.

For its operation the monitoring service requires access to the HPX monitoring service, as covered in Section 2.5.7.

Scheduler Interface

The dynamic optimizer (aka scheduler) is utilizing the interfaces provided by the various other runtime subsystems for realizing its task. It thus requires access to the work item and data management system (see Section 2.5.5), the monitoring interfaces (2.5.7 and 2.7) and the services provided by the resilience manager for preparing for and reacting to failures (2.8). Additional interfaces to components managing hardware resources may be utilized depending on the implementation of the scheduler. However, different implementations may depend on different interfaces for manipulating hardware parameters.

Additionally, the scheduler has to provide a user interface for the specification of tuning objectives.

Copyright © AllScale Consortium Partners 2017

Resilience Manager Interfaces

The resilience manager utilizes the monitoring service for observing the state and progress of a managed application. Furthermore, it uses the interface provided by work item implementations to backup or restore the state of parts of an AllScale application.

To support the scheduler in managing resilience specific objectives, it is required to provide an interface for obtaining suggestions regarding backup and restore operations (see Section 2.5.6 and 2.8).

4 Example AllScale Application

While the previous sections have been detailing the overall structure of the AllScale Environment from its architectural point of view, this section addresses the behavioral aspects of the various component and their interfaces. For its illustration, the process of an example application is described.

4.1 The Example

To illustrate the contribution and dynamic behavior of all the components of the AllScale Environment the processing of a small program is covered step by step within this section.

A typical example for a highly scalable code of great scientific interest are codes computing a solution for the field $\rho(\vec{x}, t)$ for some space domain $\vec{x} \in \mathbb{R}^n$ and time domain $t \in \mathbb{R}$ satisfying the differential equation

$$\frac{d\rho(\vec{x}, t)}{dt} = k\frac{d^2\rho(\vec{x}, t)}{d^2\vec{x}}$$

where we have an initial state for time $t = 0$. Equations of this shape may, for instance, be utilized to model the density of some liquid over time or the heat propagation within a solid.

Typically, to solve the equation numerically, the space/time domains are discretized using constant intervals. Let Δs and Δt be those chosen space and time steps. Furthermore, let $u(\vec{x}, t)$ be the discretized approximate solution computed for $\rho(\vec{x}, t)$. For the 1-dimensional case, the derivation results into the following equation:

$$u(x, t + \Delta t) := u(x, t) + k\frac{u(x - \Delta s, t) + u(x + \Delta s, t) - 2u(x, t)}{\Delta s}\Delta t$$

By defining $c = k\frac{\Delta t}{\Delta s}$ we obtain

$$u(x, t + \Delta t) := u(x, t) + c(u(x - \Delta s, t) + u(x + \Delta s, t) - 2u(x, t))$$

Thus, the following code[4] computes the solution for $u(x, T)$:

[4] for clarity we omit the handling of boundary conditions

Copyright © AllScale Consortium Partners 2017

```
double fieldA[N];
double fieldB[N];
double* A = fieldA;
double* B = fieldB;

// initialize the field with the initial state
for( int x = 0 ; x < N ; ++x) {
   A[x] = … ;  // some value
}

// compute the solution over the discrete time steps
for( int t = 0; t< T; ++t) {
   for ( int x = 1; x < N – 1 ; ++x) {
      B[x] = A[x] + c * (A[x-1] + A[x+1] – 2*A[x]);
   }
   swap(A,B);
}

// the final solution for u(x,T) is now stored in B[x]
```

The following sections illustrate how this code is processed/executed by the AllScale Environment.

4.2 API Support

By analyzing the initial (sequential) code, it is easy to see that both space loops can be processed in parallel. Thus, for the parallelization of the given code the pfor construct of the AllScale User API can be employed as follows:

```
double fieldA[N];
double fieldB[N];
double* A = fieldA;
double* B = fieldB;

// initialize the field with the initial state
pfor(0,N,[&](int x) {
   A[x] = … ;  // some value
});

// compute the solution over the discrete time steps
for( int t = 0; t< T; ++t) {
   pfor(1, N – 1, [&](int x) {
      B[x] = A[x] + c * (A[x-1] + A[x+1] – 2*A[x]);
   });
   swap(A,B);
}

// the final solution for u(x,T) is now stored in B[x]
```

Copyright © AllScale Consortium Partners 2017

This parallelizes the given code fragment similar to the way developers would handle it utilizing OpenMP or cilk. However, similar to the OpenMP or cilk implementation, the given code fragment includes an implicit barrier at the end of each parallel loop. For instance, after every call of the pfor loop in the update step the main program is waiting until the processing of the pfor is completed before swapping the two pointers A and B and starting over with the next time step. As a consequence, at each time step, all task queues in the system need to be drained before the program may continue. This incurs performance overhead we seek to omit.

The AllScale Monitoring Service provides tools enabling the identification of this kind of performance bottlenecks.

4.2.1 Fine Grained Synchronization

The fine grained synchronization constructs of the AllScale API provide the necessary means to eliminate this bottleneck. The following code fragment demonstrates its utilization:

```
double fieldA[N];
double fieldB[N];
double* A = fieldA;
double* B = fieldB;

// initialize the field with the initial state
auto ref = pfor(0,N,[&](int x) {
  A[x] = ... ;  // some value
});

// compute the solution over the discrete time steps
for( int t = 0; t< T; ++t) {
  ref = pfor(1, N – 1, [&](int x) {
    B[x] = A[x] + c * (A[x-1] + A[x+1] – 2*A[x]);
  }, wait_for_neighbors(ref));
  swap(A,B);
}
ref.wait();
// the final solution for u(x,T) is now stored in B[x]
```

The pfor provides a reference to its parallel loop iterations, which can be utilized for orchestrating the fine-grained synchronization of the created tasks.

Also note that by utilizing the loop references the implicit barrier at the end of a loop is gone and the program will not block after every invocation of the pfor. Instead, the main program quickly creates a chain of T+1 tasks, each representing one pfor invocation and depending on its predecessor. After that, the main program does nothing more than waiting for the last task, addressed by the final reference, to finish.

Copyright © AllScale Consortium Partners 2017

4.2.2 Distributed Memory Support

The presented program now runs efficiently on a shared memory system. The AllScale Scheduler takes care of a proper load balance, to not waste resources. However, it cannot yet be processed on a distributed memory system.

By default, the system is not capable of automatically distributing arrays[5]. Only data items can be distributed throughout the nodes of a distributed memory system. Thus, one more modification to the code enabling the distribution of the field data is required.

The AllScale User API provides a Grid container representing multi-dimensional arrays, satisfying the concept of a data item – which thus can be distributed. The modified code looks as follows:

```
Grid<double> A(N);
Grid<double> B(N);

// initialize the field with the initial state
auto ref = pfor(0,N,[&](int x) {
   A[x] = … ;  // some value
});

// compute the solution over the discrete time steps
for( int t = 0; t< T; ++t) {
   ref = pfor(1, N – 1, [&](int x) {
      B[x] = A[x] + c * (A[x-1] + A[x+1] – 2*A[x]);
   }, wait_for_neighbors(ref));
   swap(A,B);
}
ref.wait();
// the final solution for u(x,T) is now stored in B[x]
```

Note that by default the Grid type represents a 1-dimensional array. Also, the swap is exchanging references, not the actual data (O(1)).

All the presented codes can be compiled by any C++14 compliant standard tool chain during the development phase and is processed effectively on a shared memory architecture. However, all of them can as well be compiled utilizing the AllScale toolchain benefiting from all its additional services. Those include the advanced scheduler and monitoring support (also on shared memory) and in its final version the support of the automated distribution of the execution on a distributed memory system and/or GPUs. Furthermore, the resilience manager adds transparent resilience towards node failures to the resulting application.

[5] A special treatment for scalars and arrays may be added to the compiler, however, this is beyond the initial design of the AllScale architecture;

Copyright © AllScale Consortium Partners 2017

All those additional features are integrated by the various AllScale components covered next.

4.3 Compilation

When compiling an AllScale Application utilizing the AllScale Compiler it locates the usage of AllScale Core API primitives. While within the given user code those are not visible, the implementations of the pfor operator and the Grid container are based on those, and the compiler conducts the necessary resolution.

After locating those, each invocation of the prec operator, which is the unified central parallel construct of the Core API, is analyzed and rewritten to fit the work item interface required by the runtime system.

In the given example, the compiler locates the prec calls providing the foundation for the implementation of the pfor function utilized in the code. For each call site of the pfor another work item is created.

A work item essentially consists of the following elements:

- a sequential variant (=implementation) of the represented task, conducting the actual computation (the progress variant); for the present pfors, those implementations are implementations of simple, sequential loops processing the given body statements;
- a parallel variant of the task splitting a given task into smaller tasks and orchestrating their execution (the split variant); for the present pfors, the split steps divide the range to be covered in half, followed by spawning the two fractions in parallel
- additional (optional) code variants processing the given task by targeting specific hardware (e.g. accelerators) or optimization criteria (e.g. restricted instruction set)

For each of those work item variants a function determining the data requirements depending on the closure parameters captured by the task have to be provided in the form of an extra function callable by the runtime system.

In the given case, the captured closures are as follows:

- The initialization loop captures the start and end values of the iterator x as well as a reference to the Grid instance A
- The update loop captures the start and end values of the iterator x as well as a reference to the Grid instance A and B; note that the actual objects referenced by A and B switch at each time step;

The data requirement function obtained through static compiler analysis looks as follows:

- For the initialization loop: given a range (a,b) and grid reference A in the closure, the following data requirements are obtained
    - For the progress variant: { (WRITE_FIRST, A, [a,b)) }, stating that this variant requires write access to subrange $[a, b)$ of the Grid A. It does not care for the initial content of this fraction of A;
    - For the split variant: {}; no data beyond the parameters in the closure are required

Copyright © AllScale Consortium Partners 2017

- For the update loop: given a range (a,b) and grid references A and B in the closure, the following data requirements are obtained
    - For the progress variant: { (READ, A, [a-1,b+1)), (WRITE_FIRST, B, [a,b)) }, stating that this variant requires read access to the subrange [a-1,b+1) of grid A and write access to subrange [a,b) of Grid B. It does not care for the initial content of B;
    - For the split variant: {}; no data beyond the parameters in the closure are required

The information regarding the data dependencies is obtained through static code analysis in the compiler. If this analysis determines the capture of values that cannot be migrated – or it fails to obtain data dependencies automatically – this circumstance is reported to the developer. In those cases, the developer may eliminate the identified dependencies or try to restructure the code to aid the compiler in conducting its analysis.

As a final step, the compiler converts the entry point of an application – the main function – into an additional work item capturing the (optional) command line parameters. This main work item is exposed to the runtime as the initial task to be processed on startup. It does no support a split option.

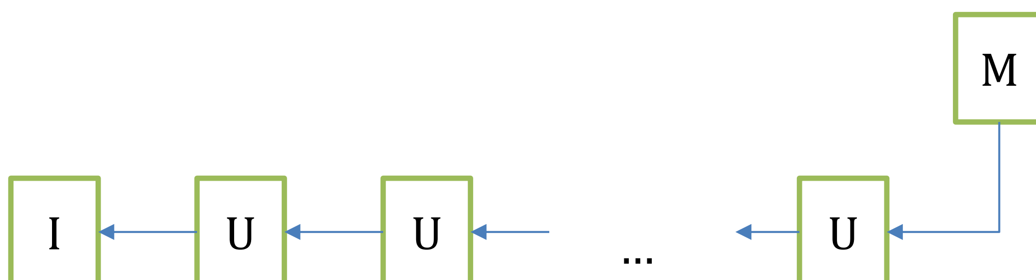
4.4 Execution

The execution of an application is steered by the AllScale Runtime System. After an initial startup phase, where the AllScale processed on the various involved nodes are initialized and a communication infrastructure is established, the processing of the main-task of the processed application is initiated.

In our running example, the main task creates the two grid instances A and B, followed by the processing of the initialization loop and the iterative update steps.

During the initialization of the grid instances A and B, an identifier for the newly created data items is generated by the runtime system and associated with its total size (the range [0,N)). Furthermore, management information regarding the distribution of those grids is initialized. Initially no fraction of the data is available throughout the system.

The execution of main task continues by creating the string of initialization and update work items, as described in Section 4.2. Thus, after a short time we might have the following task dependencies in the system:



Copyright © AllScale Consortium Partners 2017

Here, each box represents a work item, arrows represent task dependencies. The box labeled by I represents the initializing work item, the box labeled M is the main work item and each box labeled U is an update work item.

In the given situation no parallel tasks are present. However, all but the main work item can be split, thereby generating more parallelism and finer grained dependencies. So the scheduler of the runtime may decide to start splitting tasks as follows:

Each work item annotated with a prime (') has been split, and thus replaced by fractions. Consequently also their dependencies have been refined. By refining tasks to a sufficient level, enough parallelism for the available parallel resources will be eventually reached (assuming the application provides sufficient parallelism in the first place, hence N is large enough). Simultaneously, task dependencies as well as the data dependencies of tasks are reduced consistently.

The scheduler passes through a phase of splitting and distributing tasks throughout the system, thereby gradually refining those. Once a granularity deemed suitable by the scheduler is reached, the initialization tasks are assigned to some compute unit for being processed.

Before being able to processed, the data requirements of work items have to be satisfied. In our running example, the initialization tasks require WRITE_FIRST access to fractions of the grid A. Thus, the runtime may allocate arbitrary buffers local to the compute units intended for processing the selected work item, register this new fragment within the data item manager keeping track of the distribution state of the data item A, and trigger the execution.

Once the first update tasks get ready for execution due to satisfied dependencies, their data dependencies are checked. Since they do have a dependency on the A grid, they are either moved to the same location where the corresponding fragment is located or, if the scheduler decides so, the data is moved to the location of the tasks. The fraction of B required as a target buffer is allocated by the runtime wherever necessary (since it is a WRITE_FIRST dependency) and registered as a new fragment within the data item manager.

Starting with the second generation of the update steps, the data dependencies begin to reference data item fragments which are potentially not collocated on the same node. In those cases the runtime has to collect and transfer the

Copyright © AllScale Consortium Partners 2017

necessary information before being able to processes the depending work items. This process is known as the exchange of ghost cells in a conventional MPI based implementation. In the AllScale Environment, the data item manager handles this kind of information exchange automatically.

This way, the AllScale Runtime System and its scheduler gradually work through the application, having the possibility to overlap time steps, without the inherent need of global communication or phases where task queues need to be drained.

4.5 Monitoring

Throughout the execution of the application, the monitoring service collects data on the performance of the application (e.g. execution times of tasks or delays due to dependencies), the runtime system (e.g. queue lengths, idle times), and the hardware (e.g. cache misses, energy usage). This information is available to the scheduler during execution, or, later on, for a post-mortem analysis of the processed algorithm.

To control the behavior of the monitoring service the executable produced by the AllScale Compiler exhibits a set of command line options. With those, the collection of additional profiling data may, for instance, be triggered. Tools for analyzing and visualizing those results are provided as part of the AllScale Environment.

In our running example, the monitoring system could have been utilized to identify the bottleneck imposed by the barrier present at the end of each pfor invocation, before upgrading the code to utilize fine grained dependencies.

4.6 Scheduling

The scheduler is an integral part of the runtime system. It steers the work item decomposition, relocation and execution as well as the data item distribution. Furthermore it manages the configuration and utilization of the underlying hardware. It may thus, for instance, adapt the clock frequency of compute units.

As covered in the previous sections, the scheduler is flexible enough to optimize toward different objectives. Trade-offs between execution time, resource usage, power dissipation, or energy requirements may be stated. The objective to aim for may thereby be passed as a command line parameter to the executable. Thus, it is not necessary to recompile an application when adapting optimization objectives. There might even be support for altering the objective during runtime.

Since our running example is likely to be a memory bound problem the scheduler should be able to steer towards a configuration saturating all available memory controllers, while not consuming extra power for any additional cores.

Copyright © AllScale Consortium Partners 2017

4.7 Resilience

As covered within Section 2.8, AllScale applications processed by the AllScale Runtime System are implicitly hardened against node failures. This is achieved through backing up the initial state of tasks, such that upon a failure, those task may be restarted on a different locality.

In our running example, the resilience manager within the runtime system is selecting a subset of task to be backed up before being processed. It thereby ensures that the union of the transitive child-relation closures of the backed up tasks are a complete coverage of all active tasks within the system. This way, recovery from individual node failures can be guaranteed.

Whenever a task is backed up, its input values are recorded. This comprises the values within the closure of a work item as well as the data within read data item fragments. For instance, within the running example

- for initialization tasks, only the closure has to be backed up
- for update tasks, the closure and the content of the read fraction of the A grid have to be backed up; thus, for a closure covering the range [a,b) and the grids A and B, the values for a and b and the grid references A and B are backed up; furthermore, the content of the range [a,b) of grid A is backed up
- for the main task, the closure containing the initial command line arguments is backed up

The backup data is kept until the task is completed. In case a node crashes, the initial state of all the tasks currently processed by the failed node is recovered from the backup data (closure and data item state), before restarting the recovered tasks.

The entire recovery procedure is transparent to the application code develop by the end use.

5 Conclusions and Future Work

Within this document the architecture of the AllScale system has been covered. Details regarding the internal organization of the AllScale pilots, API, Compiler, Runtime System, Scheduler, Monitoring Service, and Resilience Manager have been elaborated by Section 2. Section 3 summarized the interfaces between the various components, before Section 4 demonstrated the contributions of the various components based on a concrete example.

The presented architecture design fulfills the requirements as stated in Deliverable D2.1 and provides a guideline for the development of the various AllScale components. However, new insights gained over the course of the project lead to gradual refinements or modifications of the presented architecture, to be reflected in future revisions of this document.

Copyright © AllScale Consortium Partners 2017