Computing and Informatics, Vol. 39, 2020, 808–837, doi: 10.31577/cai_2020_4_808

ALLSCALE API

Philipp Gschwandtner, Herbert Jordan
Peter Thoman, Thomas Fahringer

Department of Computer Science, University of Innsbruck
Technikerstrasse 21a, 6020 Innsbruck, Austria
e-mail: {philipp.gschwandtner, herbert.jordan, peter.thoman, thomas.fahringer}@uibk.ac.at

Abstract. Effectively implementing scientific algorithms in distributed memory parallel applications is a difficult task for domain scientists, as evident by the large number of domain-specific languages and libraries available today attempting to facilitate the process. However, they usually provide a closed set of parallel patterns and are not open for extension without vast modifications to the underlying system. In this work, we present the AllScale API, a programming interface for developing distributed memory parallel applications with the ease of shared memory programming models. The AllScale API is closed for a modification but open for an extension, allowing new user-defined parallel patterns and data structures to be implemented based on existing core primitives and therefore fully supported in the AllScale framework. Focusing on high-level functionality directly offered to application developers, we present the design advantages of such an API design, detail some of its specifications and evaluate it using three real-world use cases. Our results show that AllScale decreases the complexity of implementing scientific applications for distributed memory while attaining comparable or higher performance compared to MPI reference implementations.

Keywords: API, programming interface, parallel programming, shared memory, distributed memory, parallel operator, data structure

Mathematics Subject Classification 2010: 68-W10

1 INTRODUCTION

Even with the recent trend of many-core processors providing users with dozens of cores per chip in a single memory address space, distributed memory systems pose an essential aspect of HPC in order to achieve large-scale performance for scientific applications. Although there are certain system architectures that overcome the issue of distinct memory address spaces by hardware means (e.g. SGI's UV [21] series using the NumaLink protocol), the conventional approach is still to handle distinct memory address spaces in the software stack by providing a global address space in software or by explicit message exchange.

However, most of these ubiquitous software solutions entail several disadvantages that make them hard to use for domain scientists. Programming interfaces such as MPI are often too low-level for non-computer science experts and clutter up the application with a non-domain-relevant source code. On the other hand, there are high-level domain-specific languages or libraries that lack extensibility in order to support new scientific problems [4]. In addition, many of these solutions often lack the composability required for building libraries and integrating them seamlessly into larger applications, they deny an incremental approach that allows parallelizing an application step by step, or are limited to shared memory only. Therefore, users often resort to combining several of these solutions (e.g. MPI+OpenMP), which presupposes knowledge in at least two different programming models and entails a lack of resource management coordination that is left to the user.

In contrast, the AllScale API aims at providing the application developer with a single, extensible programming interface to express the parallel algorithms on a high level of abstraction, with automatic support for distributed memory.

The specific contributions of this work are:

- a shared-memory-style API for high-level specifications of algorithms and data structures with implicit distributed memory support,
- the capability of expressing new algorithms by extending the API with full compatibility to the rest of the software stack, and
- an evaluation of its programmability and performance using three real-life use cases.

While documentation and tutorials introducing the novice to the AllScale API are available online[1], the remainder of this work focuses on the API specification and important properties.

The rest of the paper is structured as follows. Section 2 discusses API design motivation while Section 3 and Section 4 detail API components. Implementation information is given in Section 5. Three real-world pilot applications and their respective API use are presented in Section 6 followed by an evaluation in Section 7. Related work is discussed in Section 8 and Section 9 provides the conclusion and future work.

[1] `https://github.com/allscale/allscale_api/wiki`

2 API DESIGN

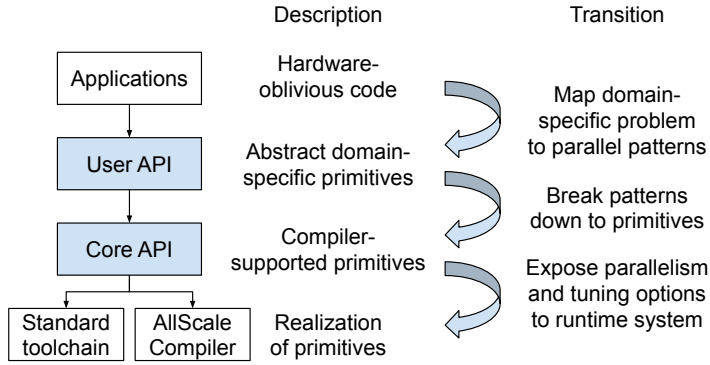

Figure 1. AllScale API design and usage overview

AllScale aims at providing domain scientists with the ability to write parallel applications for distributed memory using an API that is as easy to use as shared memory programming models such as OpenMP. While AllScale consists of many components including the API, a compiler, a distributed memory runtime system, and additional components for monitoring, resilience, etc., this work will present the API in detail. The overall architecture is discussed in detail by Jordan et al. [9].

The AllScale API is the façade of the AllScale Environment towards end-user applications. It provides the necessary primitives to express parallelism, data dependences, and needed synchronization steps within application code. The API is subdivided into two layers: the *AllScale Core API* and the *AllScale User API*. Their relationship is illustrated in Figure 1 and further discussed in the remainder of this section.

The Core API provides a concise set of basic generic primitives, comprising parallel control flow, synchronization, and communication constructs. It furthermore offers a generic data item interface that enables automatic data management of user-defined data structures. 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 or adaptive grids.

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. Currently there are two implementations available within AllScale:

- a shared memory, pure C++ implementation, also referred to as the *standard toolchain*, which can be compiled by any C++14-compliant compiler with no further third-party library dependences – this implementation serves as a de-

velopment platform for AllScale applications and also represents a reference implementation; and

- the implementation utilizing the AllScale Compiler and Runtime System, also called the *AllScale toolchain*, which comprises a combination of static program analysis (crucial for automatically deriving data dependences required for distributed memory execution), code generation, scheduling, and resilience techniques to provide a highly scalable and portable implementation of the Core API on distributed memory systems.

Hence, applications developed within AllScale can be ported from shared to distributed memory simply by switching the toolchain, without any modifications required in the application. Additional parallel constructs may be introduced in the 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, the User API shields application developers 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. Therefore, AllScale provides a separation of concerns – with the overall task of providing efficient parallel codes – distributed among three contributors:

- the *domain expert*, aiming at obtaining the most effective algorithmic solution for the problem of interest;
- the *HPC expert*, able to develop efficient domain specific primitives to be used by the domain expert, focusing on e.g. communication and synchronization overheads or cache efficiency; and
- the *system-level expert* focusing on providing the most flexible and portable implementation of the Core API, hence 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 (e.g. synchronization, communication, cache efficiency, scheduling, utilization of low-level parallel APIs), the resulting application code remains free of the otherwise necessary management code. This improves the maintainability of the resulting applications and thus the productivity of the domain expert.

3 CORE API

This section will detail the Core part of the AllScale API, specifically the primitives for parallel control flow and the concept of data items and their requirements. The User API, discussed in the section thereafter, builds on-top of these basic constructs to provide more high-level operations to domain experts. Note that while

the Core API also offers additional features such as a small performance profiling tool, discussing those exceeds the scope of this paper.

3.1 Parallel Control Flow

The AllScale Core API provides a single primitive for running concurrent tasks, resulting in feasible yet profound compiler and runtime system support for automatic distributed memory management of parallel applications. This single parallelism primitive forms the basis for all higher-level operators of the User API such as parallel loops, allowing the User API to be open for extension with new higher-level operators without any modifications required in the Core API or underlying compiler and runtime system [10].

This primitive, the *prec* [12] operator, is a higher order function combining three given functions into a new, recursive function. 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, and
- a function processing the recursive step case.

The result is a new recursive function which, for a given input parameter, conducts the specified computation accordingly. To support an arbitrary input type, the *prec* operator has the type

$$\left(\begin{array}{c} \alpha \to \mathit{bool}, \\ \alpha \to \beta, \\ (\alpha, \alpha \to \mathit{treeture}\langle\beta\rangle) \to \mathit{treeture}\langle\beta\rangle \end{array} \right) \to (\alpha \to \mathit{treeture}\langle\beta\rangle)$$

where α is the parameter type of the resulting recursive function and $\mathit{treeture}\langle\beta\rangle$ is a parameterized abstract data type (ADT) modeling a handle on parallel tasks. The three parameters of the *prec* operator are the input functions discussed above. The resulting value of type $\alpha \to \mathit{treeture}\langle\beta\rangle$ 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. A more in-depth discussion of ADTs can be found online [11].

3.2 Data Structure Primitives

While the parallel control flow primitive has been covered so far, it is not sufficient to compose parallel applications for distributed memory. In order to properly manage data dependences for parallel tasks executed in distinct memory address spaces, a specification for user-defined data structures needs to be defined as well. The purpose of this specification is to provide a single generic interface for HPC experts to implement new user-defined data structures while offering management access to the underlying runtime system for data distribution.

To this end, the data structure primitives offered by the Core API are a mere specification of any potential data type's interfaces and behaviors. Any data type T to be managed by an AllScale API implementation must provide a fragment type F for managing data storage and a range type R for addressing and managing subranges of the data structure. Table 1 lists the operators required to be defined by F and R. Proper implementation of these operators for any arbitrary data structure ensures its suitability for automatic distributed data management by the AllScale Compiler and Runtime System. Several examples that implement widely-used data structures such as grids are discussed in Section 4.2 while their implementation, among others, can be found online[2].

Name	Type	Description
		Fragment
create	$R \to F$	creates a fragment covering (at least) the specified range
delete	$F \to unit$	deletes the given fragment
resize	$(F, R) \to unit$	alters the capacity of given fragment F to cover at least the range R
mask	$F \to T$	provides access to the data stored in fragment F via the interface defined by type T
extract	$(F, R) \to Archive$	extracts the data addressed by R from fragment F and packs 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	$(F, R, Archive) \to unit$	imports the data stored in the given archive into fragment F at the specified range R
		Range
union	$(R, R) \to R$	computes the union of two ranges
intersect	$(R, R) \to R$	computes the intersection of two ranges
difference	$(R, R) \to R$	computes the set difference of two ranges
empty	$(R) \to bool$	determines whether the given range is empty, thus addressing no elements
pack	$(R) \to Archive$	serializes instances
unpack	$(Archive) \to R$	deserializes instances

Table 1. Operators to be defined by fragment F and range R types of an AllScale data structure

3.3 IO Primitives

All sensible applications require input/output (IO) for their operations. While high-performance IO is a research topic on its own, the Core API offers basic primitives

[2] `https://git.io/fj4Xj`

to facilitate high-performance IO while keeping actual implementations abstract. To that end, the Core API provides two means for storage interaction:

- stream-based, providing unordered input and output facilities, facilitating e.g. the writing of simulation results to output streams,
- memory-mapped, providing read-only facilities for efficient random access within large data sets.

Their individual discussion in Section 3.3.1 and Section 3.3.2 illustrates the need for two separate components that match the different requirements while still providing efficient operations.

3.3.1 Stream-Based

The underlying concept of the AllScale stream-based 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 mis-used for implementing synchronization operations among tasks (nor are they required for this purpose for applications that adhere to the AllScale programming model).

Within the API we utilize the abstract types *istream* and *ostream* as a representation of an input or output stream. Table 2 lists the operations provided by stream-based IO.

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 required to facilitate streaming IO.

3.3.2 Memory-Mapped

In some cases, quite complex input data structures need to be handled. For instance, indexed files providing efficient access to desired sub-fractions may be loaded by

Name	Type	Description
read	$(istream) \to \alpha$	atomically reads an element of type α from the given input stream
atomic	$\left(\begin{array}{c} istream, \\ (istream) \to unit \end{array} \right) \to unit$	An operator providing atomic access to an input stream, enabling the provided function to read a sequence of consecutive elements in order
write	$(ostream, \alpha) \to unit$	atomically writes the given element of type α to the given output stream, where it will be visible eventually
atomic	$\left(\begin{array}{c} ostream, \\ (ostream) \to unit \end{array} \right) \to unit$	An operator providing atomic access to an output stream, enabling the provided function to write a sequence of consecutive elements in order
create_in	$(string) \to istream$	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	$(string) \to ostream$	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	$(string) \to istream$	obtains an input stream to a previously opened input file which might be concurrently read
get_out	$(string) \to ostream$	obtains an output stream of a previously opened output file which might be concurrently written to

Table 2. Operations supported by stream-based IO

an application. Since the sequential access through streams would impose a major performance penalty for accessing such files, memory-mapped IO is offered for read only files. It provides the means for efficient random read-only access of sub-sets of larger data, with open files available in the address spaces of all AllScale runtime system processes.

The abstract type referencing a memory-mapped IO file is *mmfile*. Table 3 lists the operations provided by memory-mapped IO.

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

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

Table 3. Operations supported by memory-mapped IO

migration of the runtime system ensures 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 User API layer. An example is the static graph structure of a mesh (see Section 4.2.3). 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.

4 USER API

The generic nature of the Core API exceeds the complexity which could be effectively handled by domain experts for implementing parallel algorithms. For this reason, the AllScale User API aims at providing a set of more user-friendly, higher-level constructs for the composition of parallel applications by domain experts. The implementation of these constructs is carried out by HPC experts utilizing the primitives offered by the Core API.

4.1 Parallel Control Flow Constructs

While the User API is open for extension with new parallel patterns as required, several frequently-occurring patterns such as parallel loops are already provided and discussed below.

4.1.1 Parallel Loops

A vast majority of algorithms expressing data parallelism rely on parallel loops. They provide the means to perform computational work in an iteration space in parallel at the cost of executing the individual iterations concurrently and in an arbitrary order. To that end, the User API offers a parallel loop construct for realizing data-parallel programming within the AllScale environment.

Let *iterator* be a random access iterator. Then the *pfor* operator provides a parallel loop execution with the parameters defined in Table 4. Figure 2 shows a sample usage of the *pfor* operator with fine-grained synchronization. Several of these syn-

chronization patterns are available, such as *neighborhood_sync* or *one_on_one*. HPC experts are free to extend these by new patterns not yet covered.

Name	Type	Description
begin	*iterator*	inclusive beginning of the iterator range
end	*iterator*	exclusive end of the iterator range
body	$(iterator) \to \beta$	the function applied to each element
dependence	$dep\langle iterator\rangle$	optional dependence for fine-grained synchronization

Table 4. Parameters of the *pfor* operator

```
1   #include <array>
2   #include <allscale/api/user/algorithm/pfor.h>
3   namespace alg = allscale::api::user::algorithm;
4   using ArrayType = std::array<int,N>;
5   const int N = 200;
6   void initAndIncrement(const ArrayType& data, ArrayType& output) {
7   auto ref = alg::pfor(0,N,[&](int i) {
8           output[i] = ...; // initialization
9   });
10  alg::pfor(1,N−1,[&](int i) {
11          output[i] += data[i+1] + data[i] + data[i−1];
12  }, alg::neighborhood_sync(ref));
13  }
```

Figure 2. Two *pfor* operators initializing and incrementing data in a std::array with fine-grained synchronization. The second *pfor* will execute iteration i after the first has finished its iterations $i-1$, i, and $i+1$. Constructs specific to the AllScale API are shown in blue and underlined.

4.1.2 Recursive Space/Time Decomposition

A frequently utilized template for large-scale high-performance applications are stencils. In a stencil-based application, an update operation is iteratively applied to the elements of an n-dimensional array of cells. Thereby, for each update, the update operation is combining the previous values of cells within a locally confined area surrounding the targeted cell location to obtain the updated value for the targeted cell. Since these update operations within a single update step (also known as timestep) are independent, this application pattern provides a valuable source for parallelism within a correspondingly shaped application. The User API offers the *stencil* operator, the parameters of which are defined in Table 5.

Name	Type	Description
timesteps	int	number of time steps to be computed
size	int^n	spatial size of n-dimensional data to be processed
kernel function	$(int, int^n, \alpha^{s_1 \times \ldots \times s_n}) \rightarrow \alpha$	update function accepting current time, location, and grid, computing the resulting value
kernel shape	$(int^n)^*$	compile-time-constant list of offsets to cells accessed by kernel, determining its shape
boundary function	$(int, int^n, \alpha^{s_1 \times \ldots \times s_n}) \rightarrow \alpha$	update function for boundary cases, where some elements are outside the grid
initialization function	$(int^n) \rightarrow \alpha$	computes initial value for cell at given coordinate
finalization function	$(int^n, \alpha) \rightarrow unit$	function consuming value of cell at the end of a computation
observers	$\left(\begin{array}{c} (int, int^n) \rightarrow bool, \\ (int, int^n, \alpha) \rightarrow unit \end{array} \right)^*$	list of pairs, each describing an observer with time/location filtering function and actual trigger function to be applied

Table 5. Parameters of the *stencil* operator

4.1.3 Additional Operations

Beyond the *pfor* and *stencil* operators presented thus far, the User API offers additional parallel operations that are frequently encountered in parallel applications. These include e.g. the *map-reduce* operator for data aggregation, the *async* operator for single tasks, or the *vcycle* operator for multi-grid methods. However, a more detailed presentation is omitted for brevity.

4.2 Data Structures

4.2.1 Grid

A frequently-encountered data structure in high-performance codes is formed by n-dimensional arrays of values. While many programming languages support such structures for arbitrary dimensions, C/C++ only supports one-dimensional, dynamically sized arrays natively. However, this leaves creation and management of these structures to the user, forming a major obstacle for the usability of C++ on distributed memory systems.

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

- regular n-dimensional array of runtime-defined size,
- efficient read/write random access operators,
- efficient scan operation (processing all elements),
- type-parameterized in element type and no. of dimensions,
- enforces the serializability of its element types,
- implements data item concept for automated distribution.

Let $Grid\langle\alpha, n\rangle$ be the abstract data type family implemented by the AllScale User API to represent n-dimensional grids, where α is a type variable specifying the element type. Furthermore, let $type\langle\alpha\rangle$ be the meta type of type α. Then Table 6 lists the operators defined on *Grid* data structures.

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

Table 6. Operators defined on *Grid* data structures

Figure 3 illustrates the use of such a *Grid* data structure. In this case, `Grid<int,2>` (type T, as described in Section 3.2) offers operators for accessing elements within a two-dimensional structure, indexed by coordinates of type `GridRegion` (line 7, the type is not explicitly visible in this example code). `GridRegion` is the corresponding range type R of T and holds a conjunction of 2D-coordinate pairs describing axis-aligned boxes covering the range to be described – in this case a single point at position $\{7, 9\}$. An instance of type `GridFragment<double, 2>` (the corresponding fragment type F of T, generally not visible in user code) realizes the actual storage of fragments of the data stored in *Grid* instances; the implementation may hold a reference to allocated memory plus the coordinates of the covered ranges. For further reference, the *Grid* implementation of types T, R and F is available online[3].

[3] `https://git.io/JUClF`

```
1  #include <allscale/api/user/data/grid.h>
2  // create a two−dimensional grid of integers of size 10x20
3  allscale::api::user::data::Grid<int, 2> grid({10,20});
4  // initialize all elements with 1.0
5  grid.pforEach([](int& element) { element = 1.0; });
6  // set element at position [7,9] to 5.0
7  grid[{7,9}] = 5.0;
```

Figure 3. Example usage of the *Grid* data structure

4.2.2 Adaptive Grid

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

Let $AGrid\langle\alpha, n, [r_1, \ldots, r_l]\rangle$ be the ADT family implemented by the AllScale User API to represent n-dimensional *Adaptive Grids*, where α is a type variable specifying the element type, $r_1, \ldots, r_l$ the refinement factors, and l the number of refinement levels. Thus, the size of the grid at level i is defined by

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

To address elements within an *Adaptive Grid* an extension of *Grid* coordinates is required. While elements within a *Grid* can be addressed using a single coordinate of type int^n, the *Adaptive Grid* requires information regarding the location of the addressed element in the nested grid structure. Thus, additional coordinates to navigate through these refinement layers are required. Hence, to address an element within an *Adaptive Grid*, a hierarchical coordinate of type $(int^n)^+$ is required. For instance, the coordinate $[[7, 3], [2, 4], [8, 2]]$ addresses the element located within the cell that can be reached by navigating first to the top-level cell $[7, 3]$, continuing to cell $[2, 4]$ of its first refinement layer, and ending up within cell $[8, 2]$ of the second refinement layer. Let $seq\langle r_1, \ldots, r_l\rangle$ be the static meta-type of a sequence of integers $r_1, \ldots, r_l$, then Table 7 lists the operators defined on *Adaptive Grid* data structures.

4.2.3 Unstructured Mesh

The *Mesh* data structure is designed to represent a graph structure of multiple node types that are connected through various types of edges. Furthermore, a *Mesh* may

Name	Type	Description
create	$\begin{pmatrix} type\langle\alpha\rangle, \\ int^n, \\ seq\langle r_1, \ldots, r_l\rangle \end{pmatrix} \rightarrow AGrid\langle\alpha, n, [r_1, \ldots, r_l]\rangle$	creates new n-dimensional adaptive grid with element type α of given size and grid cell structure
destroy	$(AGrid\langle\alpha, n, [r_1, \ldots, r_l]\rangle) \rightarrow unit$	deletes the given adaptive grid
read	$\begin{pmatrix} AGrid\langle\alpha, n, [r_1, \ldots, r_l]\rangle, \\ (int^n)^+ \end{pmatrix} \rightarrow \alpha$	reads element from given grid at specified hierarchical coordinates
write	$\begin{pmatrix} AGrid\langle\alpha, n, [r_1, \ldots, r_l]\rangle, \\ (int^n)^+, \\ \alpha \end{pmatrix} \rightarrow unit$	updates element within given grid at specified hierarchical coordinates
refine	$\begin{pmatrix} AGrid\langle\alpha, n, [r_1, \ldots, r_l]\rangle, \\ (int^n)^+, \\ Grid\langle\alpha, n\rangle \end{pmatrix} \rightarrow unit$	refines resolution of cell addressed by given hierarchical coordinate by inserting given grid data as refinement information
coarsen	$\begin{pmatrix} AGrid\langle\alpha, n, [r_1, \ldots, r_l]\rangle, \\ (int^n)^+, \\ \alpha \end{pmatrix} \rightarrow unit$	coarsens resolution of cell addressed by given hierarchical coordinate and inserting given value data as coarsened information
getLevel	$\begin{pmatrix} AGrid\langle\alpha, n, [r_1, \ldots, r_l]\rangle, \\ (int^n)^+ \end{pmatrix} \rightarrow int$	gets currently active resolution level at a specified hierarchical grid position
scan	$\begin{pmatrix} int^n, \\ int^n, \\ AGrid\langle\alpha, n, [r_1, \ldots, r_l]\rangle, \\ ((int^n)^+) \rightarrow \beta \end{pmatrix} \rightarrow treeture\langle unit\rangle$	applies given function (in parallel) to all active hierarchical coordinates of a given interval in an arbitrary order

Table 7. Operators defined on *Adaptive Grid* data structures

consist of several layers, which describe the same graph in different levels of detail. Hierarchical edges may connect the same nodes of different layers.

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

Let $n_1, \ldots, n_m$ be a list of node types, $e_1, \ldots, e_k \in \{n_1, \ldots, n_m\}^2$ a list of edge types, and $h_1, \ldots, h_o \in \{n_1, \ldots, n_m\}^2$ a list of hierarchical edge types. Then the type $\mathit{Mesh}\langle[n_1, \ldots, n_m], [e_1, \ldots, e_k], [h_1, \ldots, h_o], l\rangle$ represents the type of a *Mesh* structure including the given node, edge, and hierarchical edge types on l layers. Furthermore, let $\mathit{id}\langle\alpha, l\rangle$ be an identifier for an element of type α on layer l within a *Mesh* – thus the type of ID used for addressing nodes, edges, or hierarchical edges within meshes. Also, let $\mathit{MData}\langle n, l, \alpha\rangle$ be the type of an attribute collection associating values of type α to nodes of type n located on layer l of some *Mesh* instance. Finally, let $\mathit{MBuilder}\langle[n_1, \ldots, n_m], [e_1, \ldots, e_k], [h_1, \ldots, h_o], l\rangle$ be the type of construction utility for creating meshes. Then Table 8 lists the operations defined on these types.

5 IMPLEMENTATION

The AllScale API is based on C++, which allows the re-use of existing tools such as debuggers, and makes heavy use of template-based meta-programming. This built-in language feature of C++ enables the scripted generation of code during compilation. Widely utilized examples include the generation of data structures like vectors, sets, or maps specialized to specific type parameters. However, the capabilities of this feature reach much further. 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. The same applies for all AllScale User API constructs, to improve their (re-)usability and flexibility.

In addition, the standard toolchain implementation of the API only requires a C++14-compliant compiler and standard library (e.g. recent versions of GCC, Clang, Apple-Clang, and Visual Studio), and hence supports application development on at least three different operating systems (Linux, OS X, Windows). In order to mitigate the initial adoption barrier of porting applications to AllScale, an SDK comprising a build system infrastructure and setup scripts is provided[4].

6 USE CASES

This section presents our real-world pilot applications that build on the AllScale API. The first, iPIC3D [14], is a particle-in-cell simulation code developed together with KTH Stockholm and employs multiple *pfor* operators and 3-dimensional *Grid*

[4] `https://github.com/allscale/allscale_sdk`

Name	Type	Description
	Mesh Builder	
create	$(type\langle Mesh\langle n,e,h,l\rangle\rangle) \to MBuilder\langle n,e,h,l\rangle$	creates builder for a given mesh type; intially, mesh is empty
destroy	$(MBuilder\langle n,e,h,l\rangle) \to unit$	destroys builder instance
addNode	$\left(\begin{array}{c} MBuilder\langle [n_1,\ldots,n_m],e,h,l\rangle, \\ type\langle n\rangle, \\ type\langle i\rangle \end{array}\right) \to id\langle n,i\rangle$	creates a new node of a given type n on a given level i within mesh under construction
link	$\left(\begin{array}{c} MBuilder\langle n,[(n_{1a},n_{1b}),\ldots,(n_{ka},n_{kb})],h,l\rangle, \\ id\langle n_{ia},j\rangle, \\ id\langle n_{ib},j\rangle \end{array}\right) \to unit$	ads edge to mesh under construction
link	$\left(\begin{array}{c} MBuilder\langle n,e,[(n_{1a},n_{1b}),\ldots,(n_{oa},n_{ob})],l\rangle, \\ id\langle n_{ia},j+1\rangle, \\ id\langle n_{ib},j\rangle \end{array}\right) \to unit$	adds hierarchical edge to mesh under construction
toMesh	$(MBuilder\langle n,e,h,l\rangle) \to Mesh\langle n,e,h,l\rangle$	obtains a copy of mesh under construction
	Mesh Structure	
store	$(Mesh\langle n,e,h,l\rangle) \to byte*$	serializes mesh
load	$\left(\begin{array}{c} byte*, \\ type\langle Mesh\langle n,e,h,l\rangle\rangle \end{array}\right) \to Mesh\langle n,e,h,l\rangle$	deserializes given byte array to mesh
destroy	$(Mesh\langle n,e,h,l\rangle) \to unit$	destroys given mesh
getNeighbors	$\left(\begin{array}{c} Mesh\langle n,[(n_{1a},n_{1b}),\ldots,(n_{ka},n_{kb})],h,l\rangle, \\ type\langle n_{ia},n_{ib}\rangle, \\ id\langle n_{ia},j\rangle \end{array}\right) \to id\langle n_{ib},j\rangle*$	obtains a list of neighbors of the given node following given kind of edge
getParents	$\left(\begin{array}{c} Mesh\langle n,e,[(h_{1a},h_{1b}),\ldots,(h_{oa},h_{ob})],l\rangle, \\ type\langle h_{ia},h_{ib}\rangle, \\ id\langle h_{ia},j\rangle \end{array}\right) \to id\langle h_{ib},j+1\rangle*$	obtains a list of parents of the given node following given kind of hierarchical edge
getChildren	$\left(\begin{array}{c} Mesh\langle n,e,[(h_{1a},h_{1b}),\ldots,(h_{oa},h_{ob})],l\rangle, \\ type\langle h_{ia},h_{ib}\rangle, \\ id\langle h_{ib},j\rangle \end{array}\right) \to id\langle h_{ia},j-1\rangle*$	obtains a list of children of the given node following given kind of hierarchical edge
scan	$\left(\begin{array}{c} Mesh\langle [n_1,\ldots,n_m],e,h,l\rangle, \\ type\langle n_i\rangle, \\ type\langle j\rangle, \\ (id\langle n_i,j\rangle) \to \beta \end{array}\right) \to treeture\langle unit\rangle$	applies given operation to every instance of selected node type on selected level within given mesh
scan	$\left(\begin{array}{c} Mesh\langle n,[(n_{1a},n_{1b}),\ldots,(n_{ka},n_{kb})],h,l\rangle, \\ type\langle (n_{ia},n_{ib})\rangle, \\ type\langle j\rangle, \\ (id\langle n_{ia},j\rangle, id\langle n_{ib},j\rangle) \to \beta \end{array}\right) \to treeture\langle unit\rangle$	applies given operation to every instance of selected edge type on selected level within given mesh
scan	$\left(\begin{array}{c} Mesh\langle n,e,[(n_{1a},n_{1b}),\ldots,(n_{oa},n_{ob})],l\rangle, \\ type\langle (n_{ia},n_{ib})\rangle, \\ type\langle j\rangle, \\ (id\langle n_{ia},j+1\rangle, id\langle n_{ib},j\rangle) \to \beta \end{array}\right) \to treeture\langle unit\rangle$	applies given operation to every instance of selected hierarchical edge type on selected pair of adjacent levels within given mesh

Mesh Data

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

Table 8. Operators defined on *Mesh* builder, *Mesh* structure and *Mesh* data

data structures. The second, AMDADOS [2], is an advection-diffusion code developed together with IBM Research Ireland and uses a 2-dimensional *stencil* operator and *adaptive grid* structure. While the full implementation of these applications is available online with an in-depth discussion available in literature [17], we only present code excerpts of the main computation here for brevity.

6.1 iPIC3D

The iPIC3D pilot application is an iterative particle-in-cell space weather simulation code and its main computation loop is shown in Figure 4. Its underlying data structure is a 3-dimensional regular equidistant *Grid* (line 13) where each element is a cell representing a cuboid and maintaining a dynamically-sized list of particles (line 8) located in this cuboid. Furthermore, each particle stores physical properties such as location, velocity, charge, and mass.

In each iteration of the simulation, the physical effects of the particles are aggregated to compute a set of induced force fields (lines 21–26). These force fields are also represented by 3-dimensional *Grid* structures (lines 9 and 14). In a next step, electromagnetic field equations are solved (lines 27–29), the forces affecting each particle's position and velocity are computed and the particles are updated accordingly (lines 35–37). Particles moving beyond the boundary of a cell need to be migrated (lines 33–35) to the respective target cell, which can be any of 26 neighbor cells. 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.

```
1   unsigned numSteps = ...; // number of time steps
2   auto zero = utils::Coordinate<3>(0); // point of origin
3   auto size = ...; // size of domain
4
5   namespace alg = allscale::api::user::algorithm;
6   namespace data = allscale::api::user::data;
7
8   struct Cell { std::vector<Particle> particles; };
9   struct FieldNode { ... // electric and magnetic field components };
10  struct DensityNode { Vector3<double> J; // current density };
11
12  // 3D grids for cells, electromagnetic field and current density
13  data::Grid<Cell, 3> cells = ...;
14  data::Grid<FieldNode,3> field = ...;
15  data::Grid<DensityNode,3> density = ...;
16  // create a grid of buffers for density projection from particles to grid nodes
17  data::Grid<DensityNode> densityContributions(size * 2);
18
19  // run time loop for the simulation
20  for(unsigned i = 0; i < numSteps; ++i) {
21  alg::pfor(zero, size, [&](const utils::Coordinate<3>& pos) {
22          //STEP 1a: collect particle density contributions and store in
                buffers
23          particleToFieldProjector(cells[pos], pos, densityContributions); });
24  alg::pfor(zero, density.size(), [&](const utils::Coordinate<3>& pos) {
25          // STEP 1b: aggregate density in buffers to density nodes
26          aggregateDensityContributions(densityContributions, pos, density[
                pos]); });
27  alg::pfor(fieldStart, fieldEnd, [&](const utils::Coordinate<3>& pos){
28          // STEP 2: solve electromagnetic field equations
29          fieldSolver(pos, density, field); });
30  alg::pfor(zero, size, [&](const utils::Coordinate<3>& pos){
31          // STEP 3: project forces to particles and move particles
32          particleMover(cells[pos], pos, field, particleTransfers); });
33  alg::pfor(zero, size, [&](const utils::Coordinate<3>& pos){
34          // STEP 4: transfer particles into destination cells
35          transferParticles(cells[pos], pos, particleTransfers); });
36  }
```

Figure 4. Code excerpt of main data structures and simulation loop of iPIC3D. The full code is available online at `https://github.com/allscale/allscale_ipic3d`.

These major simulation steps are all parallel update operations using higher-dimensional variations of the *pfor* operator. Thus, the resulting simulation code is structured like a list of update loops, enclosed within a single time step loop.

A crucial step for distributed memory implementations is the exchange of particles and field strength values among adjacent cells. The original iPIC3D implementation requires three synchronization steps for each time step based on explicit halo cell updates. The AllScale Toolchain integrates those exchanges implicitly, transparent to the application developer, and overlaps computation and communication to reduce the impact of necessary synchronization steps.

6.2 AMDADOS

AMDADOS is a numerical simulation of an oil spill, with an excerpt of the main computation code shown in Figure 5. It is based on a 2-dimensional stencil (lines 20–44) and incorporates data assimilation events (line 28) using external sensor data (line 16) in order to mitigate simulation errors. The basic data structure of this application is a regular, *Adaptive Grid* (line 18). The number of refinement levels is a compile-time constant and can be hard coded within the application (lines 4–8). However, coarsening and refinement steps are applied dynamically during execution based on the state of the simulation as well as data assimilation events (inside the functions called in lines 26 and 28, not shown in detail here).

The resolution refinement follows a hierarchical pattern. On the top level, a fixed size, regular 2D grid defines the domain of the overall simulated area. Each of these top-level cells (also called sub-domains) may then be itself recursively subdivided into small regular grids, up to a statically fixed maximum resolution. The simulation algorithm updates each sub-domain independently for a single 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 neighboring sub-domains. Once complete, boundary information is exchanged between adjacent sub-domains. Thus, sub-domains being n top-level cell-widths apart may be n time steps apart in their simulation time.

While this application could be implemented using the *pfor* operator, this would lead to a flat parallelism structure with synchronization enforced at the end of each time step. For this reason, it utilizes the *stencil* operator instead, which exposes recursive space-time decomposition and allows multiple time steps on spatially sufficiently separated sub-domains to be computed in parallel – sub-domains being n global cell-widths apart may be n time steps apart in their simulation time. In addition, it shows the observer functionality of the *stencil* operator, which allows for time- and space-controlled output of the simulation state.

The assimilation of data (line 28) 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.

```
1   namespace alg = algorithm;
2   namespace alg = data;
3
4   typedef data::CellConfig<2, data::layers<
5           data::layer<1,1>, // layer 2, size 1 x 1
6           data::layer<8,8>, // layer 1, size 8 x 8
7           data::layer<2,2> // layer 0, size 16 x 16
8   >> subdomain_config_t;
9
10  using subdomain_t = data::AdaptiveGridCell<double, subdomain_config_t
        >;
11  using domain_t = data::Grid<subdomain_t, 2>;
12
13  struct Sensor { ... };
14
15  // 2D grid of sensor data
16  const data::Grid<Sensor, 2> sensors = ...;
17  const size_t Nt = ...; // number of time steps
18  domain_t state_field = ...; // 2D grid of sub-domains constitutes entire
        domain
19
20  alg::stencil(state_field, Nt,
21  [&,Nt](time_t t, const point2d_t& idx, const domain_t& state)
22  -> const subdomain_t
23  { // Computation of subdomains
24          subdomain_t temp_field;
25          if(contexts[idx].sensors.empty()) // subdomain without sensors
26          SubdomNoSensors(t, state, temp_field, contexts[idx], idx);
27          else // subdomain with sensors, assimilation occurs
28          SubdomKalman(sensors[idx], t, state, temp_field, contexts[idx], idx);
29          return temp_field;
30  },
31  alg::observer( // Monitoring: periodically write full subdomain to file
32          [=](time_t t) { return (t % output_interval == 0); // time filter },
33          [](const point2d_t&) { return true; // Space filter: no constraints
                },
34          // Append a full field to the file of simulation results.
35          [&](time_t t, const point2d_t& idx, const subdomain_t& cell) {
36          cell.forAllActiveNodes([&](const point2d_t& loc, double val) {
37                  point2d_t glo = computeGlobalIndex(loc, idx);
38                  out_stream.atomic([=](auto& file) {
39                  file << ... << "\n"; // write output data
40                  });
```

```
41                  });
42                  }
43          )
44          );
```

Figure 5. Code excerpt of the main stencil computation of AMDADOS. The full code is available online at `https://github.com/allscale/allscale_amdados`.

An assimilation operation, however, is orders of magnitudes more complex than a mere simulation time step for the same sub-domain. These trigger load imbalance that has to be dealt with. Thus the application consists of fixed-size subdomains with two levels of varying computational expenses:

- coarsely resolved sub-domains incur less computational workload than finer resolved sub-domains,
- optional data assimilation steps on sub-domains incur significantly additional, sporadic computational costs.

6.3 FINE/Open

The FINE/Open application is a computational fluid dynamics (CFD) solver developed by NUMECA [16]. The underlying data structure is a static, unstructured *Mesh* comprising objects such as cells, faces, edges, nodes, or boundary faces. The computation is based on a *vcycle*, which is further detailed below.

While it is not possible to show the actual implementation of FINE/Open due to non-disclosure concerns, Figure 6 shows a basic code example which is built on-top of the same *Mesh* data structure and also performs computations using the *vcycle* operator. The geometric information is covered by a list of relations connecting cells, faces and vertices (line 3 and lines 5–9) with each other (e.g. a relation of a cell to its faces). These relations can be easily navigated by templated member functions (lines 45–46, 51 and 53). Note that the type system of C++ ensures proper object navigation during compile-time (e.g., attempting to illegally access a vertex from a face would result in a compiler error). Furthermore, for each object, a set of properties influencing the simulation is maintained (lines 13–16). These may comprise static information like e.g. the volume of a cell or dynamic information such as 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 enable the use of multi-grid solvers (lines 39–60). The hierarchy of meshes can be navigated via hierarchical edges (line 11).

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

```
1   using namespace allscale::api::user;
2   // - elements -
3   struct Cell {}; struct Face {}; struct Vertex {};
4   // - edges -
5   struct Cell2Vertex : public data::edge<Cell, Vertex> {};
6   struct Cell2Face_In : public data::edge<Cell, Face> {};
7   struct Cell2Face_Out : public data::edge<Cell, Face> {};
8   struct Face2Cell_In : public data::edge<Face, Cell> {};
9   struct Face2Cell_Out : public data::edge<Face, Cell> {};
10  // - hierarchical edges -
11  struct Parent2Child : public data::hierarchy<Cell,Cell> {};
12  // -- property data --
13  struct CellTemperature : public data::mesh_property<Cell,double> {};
14  struct FaceArea : public data::mesh_property<Face,value_t> {};
15  struct FaceVolRatio : public data::mesh_property<Face,value_t> {};
16  struct FaceConductivity : public data::mesh_property<Face,double> {};
17  // - define the mesh and builder -
18  template<unsigned levels = 1>
19  using MeshBuilder = data::MeshBuilder<
20  data::nodes<Cell, Face, Vertex>,
21  data::edges<Cell2Vertex, Cell2Face_In, Cell2Face_Out, Face2Cell_In,
        Face2Cell_Out>,
22  data::hierarchies<Parent2Child>,
23  levels>;
24  // -- type of a mesh --
25  template<unsigned levels = 1, unsigned PDepth = P_DEPTH>
26  using Mesh = typename MeshBuilder<levels>::template mesh_type<
        PDepth>;
27  // -- type of the properties of a mesh --
28  template<typename Mesh>
29  using MeshProperties = data::MeshProperties<Mesh::levels,
30  typename Mesh::partition_tree_type, CellTemperature, FaceConductivity,
        VertexPosition>;
31  // V-Cycle stage
32  template<typename Mesh, unsigned Lvl>
33  struct TemperatureStage {
34  const Mesh& mesh; // mesh structure, properties and cell data
35  MeshProperties<Mesh>& properties;
36  attribute<Cell, value_t> temperature;
37  attribute<Face, value_t> fluxes;
38
39  void jacobiSolver() {
40          auto& fConductivity = properties.template get<FaceConductivity
                , Lvl>();
41          auto& fArea = properties.template get<FaceArea, Lvl>();
```

```
42                  auto& fVolumeRatio = properties.template get<FaceVolRatio,
                        Lvl>();
43
44                  mesh.template pforAll<Face, Lvl>([&](auto f) { // compute per-
                        face flux
45                  auto in = mesh.template getNeighbor<Face2Cell_In>(f);
46                  auto out = mesh.template getNeighbor<Face2Cell_Out>(f);
47                  value_t gradTemp = temperature[in] - temperature[out];
48                  fluxes[f] = fVolumeRatio[f] * fConductivity[f] * fArea[f] * gradTemp;
49                  });
50                  mesh.template pforAll<Cell, Lvl>([&](auto c) { // update per-
                        cell solution
51                  auto subtractingFaces = mesh.template getNeighbors<
                        Face2Cell_In>(c);
52                  for(auto sf : subtractingFaces) { temperature[c] -= fluxes[sf]; }
53                  auto addingFaces = mesh.template getNeighbors<Face2Cell_Out
                        >(c);
54                  for(auto af : addingFaces) { temperature[c] += fluxes[af]; }
55                  }); }
56
57          void computeFineToCoarse() { ... }
58          void computeCoarseToFine() { ... }
59          void restrictFrom(TemperatureStage<Mesh,Lvl-1>& childStage) { ... }
60          void prolongateTo(TemperatureStage<Mesh,Lvl-1>& childStage) { ... }
61          };
```

Figure 6. Code excerpt of a sample application using the *vcycle* operator on a multi-grid mesh. The full version is available online at `https://git.io/fjBTq`.

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 simulation continues with the next time step.

7 EVALUATION

7.1 Productivity

Table 9 lists absolute values for code metrics collected for the AllScale and MPI versions of iPIC3D and AMDADOS, in order to get a grasp on the productivity of working with the AllScale API compared to MPI. *P.SLOC* denotes the lines of code spent on parallelizing the application, counting only the minimal set of lines

containing explicit interface calls. Additional code required for e.g. preparing arguments is not accounted for, and hence these results present a best-case perspective for MPI. Nevertheless, as the numbers show, AllScale clearly outperforms MPI. Furthermore, we have measured the sum of the cyclomatic complexity [15] ($TOT\ CY$) as well as the total count of non-comment lines of code ($SLOC$) over all translation units. Compared to $P.SLOC$, these give an indication of how much of the overall pilot application complexity is related to their manual MPI parallelization rather than actual domain science content.

	AMDADOS		iPIC3D	
	AllScale	MPI	AllScale	MPI
P.SLOC	25	70	23	56
SLOC	1 136	1 420	1 443	1 717
TOT CY	154	181	204	264

Table 9. Pilot application code metrics

Note that, in addition to greatly reducing the user-facing complexity of implementing distributed memory parallel programs, the AllScale versions inherently provide the possibility of integrating advanced features such as hybrid distributed/shared memory parallelism, inter-node load balancing, overlapping of communication and computation, or high-level monitoring facilities. All these features are not present in the MPI versions, and thus not accounted for in the comparisons presented here.

7.2 Performance

In order to ascertain the performance of the AllScale API and the underlying AllScale toolchain, we conducted weak scaling experiments for AMDADOS and iPIC3D on the Vienna Scientific Cluster (VSC-3) and the Meggie cluster of the University of Erlangen-Nuremberg. Table 10 lists their hardware characteristics. The initial problem size for a single node was chosen such that application throughput did not noticeably improve when further increasing the problem size.

Name	Nodes	CPU (Intel Xeon)	RAM	Interconnect	Compiler	MPI
VSC-3	512	2x E5-2650 v2	64 GB	IB QDR-80	GCC 7.2	OpenMPI 3.0.0
Meggie	256	2x E5-2630 v4		OPA 100 GBit	GCC 7.3	Intel MPI 2018.2

Table 10. Experimental platform description. The number of nodes refers to the maximum used in this work.

Figure 7 compares the performance results of the AllScale implementations against MPI reference implementations, with performance measured as application throughput.

For AMDADOS the AllScale variant achieves higher performance on both the VSC-3 and Meggie cluster. On both systems up to 160 % higher performance is

obtained, in particular for executions involving a larger number of nodes. In case of the 512 node run on VSC-3, the degrading scalability of the MPI reference implementation lead to exceeding the available time limit for our jobs. We defined this limit as 30 times the execution time of a single node run.

For iPIC3D, on the other hand, the MPI implementation demonstrates nearly-optimal scalability on both the VSC-3 and Meggie cluster. The throughput per node remains almost constant throughout the range of evaluated system setups. The AllScale version, however, shows varying performance characteristics. Generally, good performance is observed for a small number of nodes. However, while on Meggie throughput on larger scale systems remains comparable to the MPI reference version, on VSC-3 a considerably lower throughput is observed, which remains constant between 4 and 128 nodes. Beyond 128 nodes, performance degrades considerably again.

These results demonstrate the feasibility of using automatically managed user-defined data structures in large-scale high performance applications.

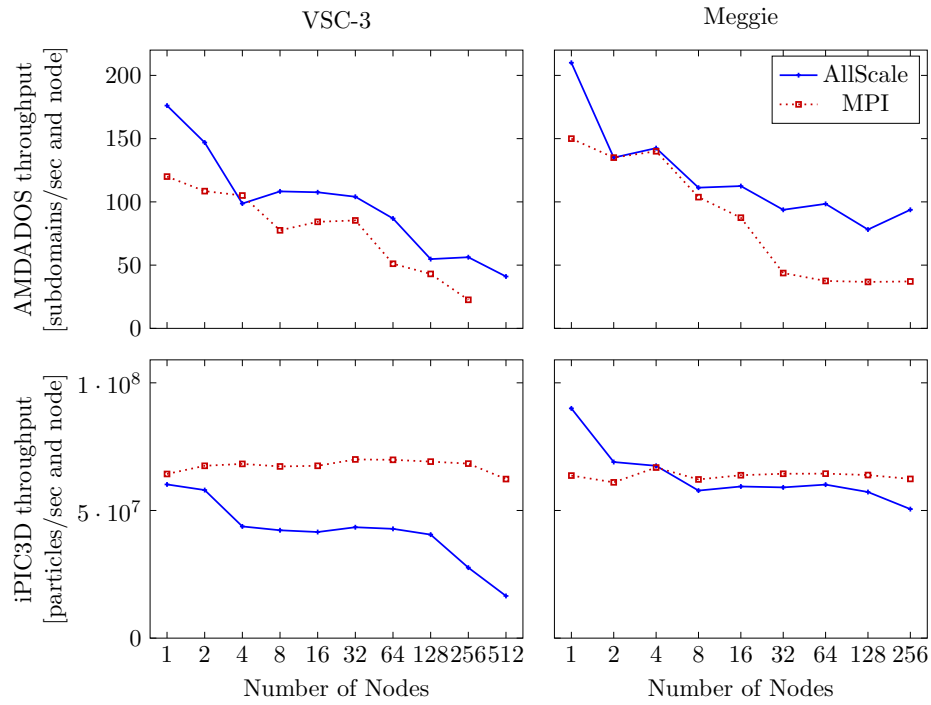

Figure 7. Comparison of throughput per node for AMDADOS and iPiC3D on VSC-3 and Meggie for MPI and AllScale

8 RELATED WORK

Conventional, low-level HPC infrastructures comprising combinations of MPI with some per-node parallelism APIs are still the default platforms for building HPC applications, but require programmers manually implement workload decomposition. Systems providing a higher level of abstraction, such as the AllScale API, can be grouped into three broad categories: new general purpose languages, domain-specific frameworks, and general purpose libraries. Note that there is a large number of parallelism approaches constrained to single-node shared memory hardware. We omit these from our overview provided here as they do not address the same problem space as the AllScale API.

In terms of languages, X10 [7] and Chapel [6] have targeted (recursive) parallelism on large scale, distributed systems, but left locality and data management to the user. Charm++ [13], on the other hand, is a C++ extension aiming at isolating the user from low-level mapping activities, thus facilitating portability. Its design is based on message-exchanging entities exposed to the user and lacks automated data distribution management. Recently, the ANTAREX research project [19] proposed a DSL-based approach, facilitating the separation of concerns between functional and non-functional aspects of HPC applications. However, due to its DSL-focused design, users require additional tools and may not rely on the experience of an established developer community.

Several new frameworks such as Lift [20], Delite [5], or AnyDSL [18] provide environments for implementing DSLs. Internally, DSL constructs are encoded using functional IR constructs like `map`, `reduce`, or `zip`. However, the resulting programming interface for the domain experts remains a DSL, targeting very specific application domains and inheriting the difficulties of DSLs noted above.

Domain-specific, C++-based libraries such as PETSc [4] or TensorFlow [1] handle several of the challenges addressed by our framework successfully for their respective domains. However, they are tailored towards specific domains instead of supporting a wider range of applications.

More general purpose parallel C++ library based frameworks like STAPL [3] and Kokkos [8] are exercising control over parallel algorithms and data structures similar to our architecture. STAPL envisions a separation of concerns strategy similar to ours. Kokkos, on the other hand, has a strong focus on multidimensional arrays and parallel loops, unlike the wider range of data structures and operations supported by our architecture. Due to a lack of compiler integration, these approaches require data dependences of code regions to be expressed explicitly as part of the API, while this is covered implicitly in our approach.

9 CONCLUSION

This work presented the AllScale API, a novel interface for implementing distributed memory parallel applications with the programmability of a shared memory API. We illustrated how the distinction into the User and Core components provides

a separation of concerns for domain experts, HPC experts and system-level experts, and discussed several constructs of the AllScale API in detail. In addition, the three use cases presented show the suitability of our approach to real-world scientific problems, evaluated in both productiveness and parallel performance.

Future work includes better user feedback for programming errors, additional pre-provided operators in the User API along with new applications.

Acknowledgements

This project has received funding from the European Unions Horizon 2020 research and innovation programme as part of the FETHPC AllScale project under grant agreement No. 671603 and from the FFG (Austrian Research Promotion Agency) project INPACT, project No. 868018. The computational results presented have been achieved in part using the Vienna Scientific Cluster and the Regionales Rechen-Zentrum Erlangen.

REFERENCES

[1] Abadi, M.—Agarwal, A.—Barham, P.—Brevdo, E.—Chen, Z.—Citro, C.—Corrado, G. S.—Davis, A.—Dean, J.—Devin, M. et al.: Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2016, arXiv preprint arXiv:1603.04467.

[2] Akhriev, A.—O'Donncha, F.—Gschwandtner, P.—Jordan, H.: A Localised Data Assimilation Framework within the 'AllScale' Parallel Development Environment. OCEANS 2018 MTS/IEEE Charleston, 2018, pp. 1–7, doi: 10.1109/OCEANS.2018.8604556.

[3] An, P.—Jula, A.—Rus, S.—Saunders, S.—Smith, T.—Tanase, G.—Thomas, N.—Amato, N.—Rauchwerger, L.: STAPL: An Adaptive, Generic Parallel C++ Library. International Workshop on Languages and Compilers for Parallel Computing, Springer, 2001, pp. 193–208, doi: 10.1007/3-540-35767-X_13.

[4] Balay, S.—Abhyankar, S.—Adams, M.—Brune, P.—Buschelman, K.—Dalcin, L.—Gropp, W.—Smith, B.—Karpeyev, D.—Kaushik, D. et al.: PETSc Users Manual Revision 3.7. Technical Report ANL-95/11 Rev 3.7, Argonne National Laboratory (ANL), Argonne, United States, 2016.

[5] Brown, K. J.—Sujeeth, A. K.—Lee, H. J.—Rompf, T.—Chafi, H.—Odersky, M.—Olukotun, K.: A Heterogeneous Parallel Framework for Domain-Specific Languages. 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2011, pp. 89–100, doi: 10.1109/PACT.2011.15.

[6] Chamberlain, B. L.—Callahan, D.—Zima, H. P.: Parallel Programmability and the Chapel Language. The International Journal of High Performance Computing Applications, Vol. 21, 2007, No. 3, pp. 291–312, doi: 10.1177/1094342007078442.

[7] Charles, P.—Grothoff, C.—Saraswat, V.—Donawa, C.—Kielstra, A.—Ebcioglu, K.—von Praun, C.—Sarkar, V.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05), 2005, ACM SIGPLAN Notices, Vol. 40, 2005, No. 10, pp. 519–538, doi: 10.1145/1103845.1094852.

[8] Edwards, H. C.—Trott, C. R.—Sunderland, D.: Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. Journal of Parallel and Distributed Computing, Vol. 74, 2014, No. 12, pp. 3202–3216, doi: 10.1016/j.jpdc.2014.07.003.

[9] Jordan, H.—Gschwandtner, P.—Thoman, P.—Zangerl, P.—Hirsch, A.—Fahringer, T.—Heller, T.—Fey, D.: The AllScale Framework Architecture. Parallel Computing, Vol. 99, 2020, Art. No. 102648, doi: 10.1016/j.parco.2020.102648.

[10] Jordan, H.—Heller, T.—Gschwandtner, P.—Zangerl, P.—Thoman, P.—Fey, D.—Fahringer, T.: The AllScale Runtime Application Model. 2018 IEEE International Conference on Cluster Computing (CLUSTER), 2018, pp. 445–455, doi: 10.1109/CLUSTER.2018.00088.

[11] Jordan, H.—Iakymchuk, R.—Fahringer, T.—Thoman, P.—Heller, T. et al.: D2.4 – AllScale System Architecture (b). May 2018, `http://www.allscale.eu/docs/D2.4%20-%20AllScale%20System%20Architecture%20(b).pdf`.

[12] Jordan, H.—Thoman, P.—Zangerl, P.—Heller, T.—Fahringer, T.: A Context-Aware Primitive for Nested Recursive Parallelism. In: Desprez, F. et al. (Eds.): Euro-Par 2016: Parallel Processing Workshops. Springer, Cham, Lecture Notes in Computer Science, Vol. 10104, 2017, pp. 149–161, doi: 10.1007/978-3-319-58943-5_12.

[13] Kale, L. V.—Krishnan, S.: CHARM++: A Portable Concurrent Object Oriented System Based on C++. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93), 1993, ACM SIGPLAN Notices, Vol. 28, 1993, No. 10, pp. 91–108, doi: 10.1145/167962.165874.

[14] Markidis, S.—Lapenta, G.—Rizwan-Uddin: Multi-Scale Simulations of Plasma with iPIC3D. Mathematics and Computers in Simulation, Vol. 80, 2010, No. 7, pp. 1509–1519, doi: 10.1016/j.matcom.2009.08.038.

[15] McCabe, T. J.: A Complexity Measure. IEEE Transactions on Software Engineering, Vol. SE-2, 1976, No. 4, pp. 308–320, doi: 10.1109/TSE.1976.233837.

[16] NUMECA International. 2019, `https://www.numeca.com/`.

[17] O'Donncha, F.—Iakymchuk, R.—Akhriev, A.—Gschwandtner, P.—Thoman, P.—Heller, T.—Aguilar, X.—Dichev, K.—Gillan, C.—Markidis, S.—Laure, E.—Ragnoli, E.—Vassiliadis, V.—Johnston, M.—Jordan, H.—Fahringer, T.: AllScale Toolchain Pilot Applications: PDE Based Solvers Using a Parallel Development Environment. Computer Physics Communications, Vol. 251, 2020, Art. No. 107089, doi: 10.1016/j.cpc.2019.107089.

[18] Leissa, R.—Boesche, K.—Hack, S.—Pérard-Gayot, A.—Membarth, R.—Slusallek, P.—Müller, A.—Schmidt, B.: AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. Proceedings of the ACM

on Programming Languages, Vol. 2, 2018, Issue OOPSLA, Art. No. 119, doi: 10.1145/3276489.

[19] Silvano, C.—Agosta, G.—Cherubin, S.—Gadioli, D.—Palermo, G.—Bartolini, A.—Benini, L.—Martinovič, J.—Palkovič, M.—Slaninová, K.—Bispo, J. A.—Cardoso, J. M. P.—Abreu, R.—Pinto, P.—Cavazzoni, C.—Sanna, N.—Beccari, A. R.—Cmar, R.—Rohou, E.: The ANTAREX Approach to Autotuning and Adaptivity for Energy Efficient HPC Systems. Proceedings of the ACM International Conference on Computing Frontiers (CF '16), ACM, 2016, pp. 288–293, doi: 10.1145/2903150.2903470.

[20] Steuwer, M.—Remmelg, T.—Dubach, C.: LIFT: A Functional Data-Parallel IR for High-Performance GPU Code Generation. 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2017, pp. 74–85, doi: 10.1109/CGO.2017.7863730.

[21] Thorson, G.—Woodacre, M.: SGI UV2: A Fused Computation and Data Analysis Machine. Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12), IEEE, 2012, Art. No. 105, doi: 10.1109/SC.2012.102.

Philipp Gschwandtner is a senior scientist at the Research Center HPC at the University of Innsbruck, Austria. He completed his Ph.D. in 2017 with a thesis on performance and energy analysis and optimization of parallel programs. His main research interests lie in high performance computing and parallel programming, including adjacent topics such as scientific computing, program optimization, API design, runtime systems, and compiler research.

Herbert Jordan is a former PostDoc at the University of Innsbruck, Austria. After completing his Ph.D. in 2014 on the Insieme optimizing compiler infrastructure he became the Scientific Coordinator of the EU H2020 project AllScale, aimed at exposing nested recursive parallelism for distributed memory. His main research interests are in programming language and API design, static program analysis, code transformations, and performance optimizations.

Peter Thoman is Assistant Professor of computer science at the University of Innsbruck. His research focus has been to improve the effective performance and programmability of complex, highly parallel systems. He has created and contributed to various APIs for HPC and GPGPU platforms, such as the Celerity high-level single-source platform for GPU Clusters. He also investigates runtime systems, as well as compiler-based tools for performance optimization as well as improved developer support.

Thomas Fahringer is Professor of computer science at the University of Innsbruck since 2003. His research focuses on supporting researchers in science and engineering by developing, analysing, and optimizing parallel and distributed applications. Fahringer was involved in numerous national and international research projects including 15 EU funded projects, published 5 books, 40 journal and magazine articles, and more than 200 reviewed conference papers. He is a member of the IEEE and ACM.