The AllScale Runtime Application Model (incl. Appendix)

Herbert Jordan, Philipp Gschwandtner,
Peter Zangerl, Peter Thoman and Thomas Fahringer
University of Innsbruck, 6020 Innsbruck, Austria
Email: {herbert,philipp,peterz,petert,tf}@dps.uibk.ac.at

Thomas Heller and Dietmar Fey
Friedrich-Alexander University of Erlangen-Nuremberg
91058 Erlangen, Germany
Email: {thomas.heller,dietmar.fey}@fau.de

***Abstract*—Contemporary state-of-the-art runtime systems underlying widely utilized general purpose parallel programming languages and libraries like OpenMP, MPI, or OpenCL provide the foundation for accessing the parallel capabilities of modern computing architectures. In the tradition of their respective imperative host languages those runtime systems' main focus is on providing means for the distribution and synchronization of operations — while the organization and management of manipulated data is left to application developers. Consequently, the distribution of data remains inaccessible to those runtime systems. However, many desirable system-level features depend on a runtime system's ability to exercise control on the distribution of data. Thus, program models underlying traditional systems lack the potential for the support of those features.**

In this paper, we present a novel application model granting parallel runtime systems system-wide control over the distribution of user-defined shared data structures. Our model utilizes the high-level nature of parallel programming languages, in particular, the usage of well-typed data structures and the associated hiding of implementation details from the application developers. By being based on a generalization of such data structures and extending the resulting abstraction with features facilitating the automated management of the distribution of those, our model enables runtime systems to dynamically influence the placement and replication of shared data. This paper covers a rigorous formal description of our application model, as well as details on our prototype implementation and experimental results demonstrating its ability to efficiently and scalably manage various data structures in real-world environments.

1. Introduction

The vast majority of programming languages used today for the development of high performance applications are imperative languages. Their core features comprise means to express the order of operations to be performed to achieve the desired objective. Parallel libraries and language extensions like pthreads, OpenMP, MPI, OpenCL, or CUDA extend those capabilities by enabling the specification of partial orders of operations, facilitating the effective utilization of parallel resources. Following the tradition of their respective host languages, these extensions are themselves focused on the orchestration of operations.

However, besides operations, every computation process needs to be concerned with data. Data constitutes the input and output of programs and provides the substrate all operations act upon. To ease the task of programming, data gets organized in data structures — higher level abstractions enabling modular reasoning over applications. Only a few, simple data structures like arrays are directly supported by common programming languages. More complex structures like lists, trees, graphs, sets, maps, or meshes are emergent features supported by programming languages through the power of composition.

A common practice for the development of parallel high performance (HPC) codes is to start the design of programs by outlining an essential data structure the program will operate on. For instance, a finite element simulation will perform its operations on some sort of mesh, while the simulation of the gravitational forces between stars will be based on some tree structure grouping elements by their spatial relation. In a subsequent step, a partitioning scheme for the envisioned data structure is devised and finally implemented using a parallel language. The actual computation will then build on top of the designed structure.

Consequentially, the data structure design forms the foundation of many high-performance applications. However, due to being an emergent feature of the composition of language features, these structures are beyond the reach of contemporary parallel runtime systems. In fact, none of the parallel languages enumerated above provide any direct support for data structures beyond arrays — coinciding with the level of data structure support provided by their host languages. Higher level constructs are to be obtained through composition.

As it constitutes the foundation of HPC applications, the management of data structures, in particular, their distribution among address spaces, is essential for the realization of a variety of desirable system-level features. Inter-node load balancing, the offloading of computation to GPUs, the dynamic adaptation to changes in resource availability, or the checkpointing and restarting of computation all depend on the manipulation of the distribution of the underlying data structure. Contemporary general-purpose runtime sys-

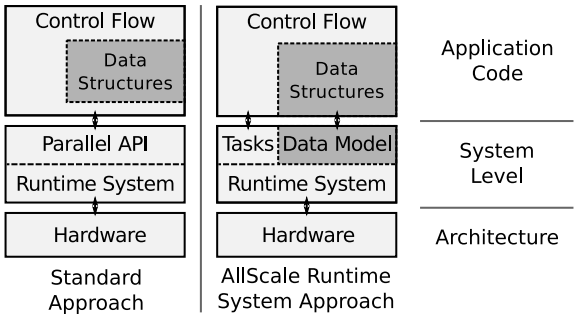


Figure 1: Standard vs. data aware runtime systems.

tem designs have limited potential for these operations, due to their lack of influence on and control over the data structures used. As such, application developers generally need to integrate these features manually as required.

In this paper, we present the AllScale runtime application model, one of the theoretical foundations of the AllScale runtime system. Its key novelty is the provisioning of generic support for the dynamic distribution and management of user-defined data structures for distributed memory environments, as outlined by Fig. 1. Its utilization relieves developers from the associated data management responsibilities, without losing flexibility in the design of partitioning schemes. Furthermore, by taking control of this crucial aspect of HPC applications, the potential for the integration of advanced runtime-system-level services is introduced. Our key contributions are:

- formalization of a novel parallel application model providing unprecedented management access to user-defined data structures to the runtime system,
- provisioning of a programming API and source-to-source compiler interfacing with a prototype implementation of our runtime model, and
- evaluation of the capabilities of our approach through a set of example applications.

Our model has been developed in the context of the AllScale project [1], aiming for the research of improved programming models for HPC applications based on advanced compiler and runtime technology.

The remainder of this paper is structured as follows: Section 2 provides a formal specification of our application model, before Section 3 describes its implementation in our prototype runtime system. Section 4 shows the performance evaluation of three example codes and Section 5 compares our approach with related work. Finally, Section 6 provides conclusive remarks and an outlook towards future work.

2. Application Model

The AllScale Runtime Application Model comprises three major components: a data model, a task model and an architecture model. The former two describe data objects and tasks managed during the execution of an application, while the architecture model provides an abstraction of the underlying hardware infrastructure.

In this section we provide a rigorous abstract formal definition of the main elements of the utilized models, followed by a specification of the full application state model and valid state transitions – thus valid application/runtime interactions. The resulting model constitutes a specification for implementations and provides a reference to reason about valid system states, state transitions, invariants, and dynamic system properties.

2.1. Data Model

The first model provides an abstraction of data objects to be managed by the runtime system. For the model covered in this section we focus on the bare essential requirements the runtime imposes on concrete implementations of data structures following this model. Nevertheless, examples outlining concrete implementations are provided. Section 3 covers actual implementation details.

The foundation of the data model is an abstraction of arbitrary data structure instances referred to as *data item*s. Instances of data structures like arrays, trees, maps, or graphs provide means to organize sets of logically addressable data elements. This basic concept is covered by the following definition.

Definition 2.1 (Data Item)**.** Let $\mathbb{D}$ be the set of all data structure instances, $\mathbb{E}$ the set of all logical addresses of data elements within those, $2^{\mathbb{E}}$ the power set of $\mathbb{E}$, and *elems* : $\mathbb{D} \to 2^{\mathbb{E}}$ a function assigning each *data item* $d \in \mathbb{D}$ its finite set of element addresses $\textit{elems}(d) \subseteq \mathbb{E}$.

Example 2.1 (Data Items)**.** Let $d_a \in \mathbb{D}$ represent a 1D array *A* of 20 data elements ($A[0] = e_1, \ldots, A[19] = e_{20}$), then $\textit{elems}(d_a) = \{e_1, \ldots, e_{20}\}$. Similar, let $d_t \in \mathbb{D}$ be a balanced binary tree *T* of height 4 containing 15 nodes $n_1, \ldots, n_{15}$. Then $\textit{elems}(d_t) = \{n_1, \ldots, n_{15}\}$.

Due to their property of being assemblies of individually addressable data elements, data items can be decomposed and distributed among multiple address spaces. This is the implicit basic principle of all HPC applications sharing a global view on common data.

To facilitate the automated management of the distribution of data items, subsets of addressable elements need to be addressable. Such an addressable subset is referred to as a *region*.

Definition 2.2 (Region)**.** Let $d \in \mathbb{D}$ be a data item. Then a set of element addresses $r \subseteq \textit{elems}(d) \subseteq \mathbb{E}$ is a region of data item d. Let the set of all regions be denoted by $\mathbb{R}$.

Example 2.2 (Regions)**.** Let $d_a \in \mathbb{D}$ be a 3D array of 100^3 addressable elements $\{e_{(0,0,0)}, \ldots, e_{(99,99,99)}\}$. Then the box of elements $\{e_{(i,j,k)} \mid 10 \leq i,j,k \leq 20\}$ is a region of d_a. So is the set $\{e_{(i,i,i)} \mid 14 \leq i \leq 30\}$.

Since there might be billions of addressable elements for individual data items, enumerating them explicitly is not very efficient and in many cases prohibitively expensive. Thus, efficient means for defining regions, as hinted by the implicit notation utilized in Example 2.2, are required.

Section 3 provides examples of such. In this section we focus on functional aspects of our model.

Note that our definitions target the logical addresses of stored elements, not their physical or virtual memory addresses, nor their values. Actual values can be modeled by a function $val : \mathbb{D} \times \mathbb{E} \to \mathbb{X}$ for some value domain $\mathbb{X}$. This value function would then have to be updated along the evolution of the system state whenever the state of an addressed element is updated. However, since this is not relevant for the content of this paper we omit the evolution of the value state of data items from our model.

2.2. Task Model

The second part of our model covers *tasks*. Tasks are the active entities of applications performing operations on data items. In the AllScale model, each task can be specified through one or more alternative implementations, referred to as (task) *variants*.

Definition 2.3 (Task)**.** Let $\mathbb{T}$ be the set of all *tasks*, $\mathbb{V}$ be the set of all (task) *variants*, and $var : \mathbb{T} \to 2^{\mathbb{V}} \setminus \emptyset$ be the function assigning each task its finite set of variants.

Example 2.3 (Task)**.** Let $t \in \mathbb{T}$ be a task computing the sum of a sub-range of array elements, $v_s \in \mathbb{V}$ be a sequential implementation and $v_p \in \mathbb{V}$ be a parallel variant dividing the task in half and spawning two sub-tasks to perform the computation. Then $var(t) = \{v_s, v_p\}$ reflects the fact that the runtime may choose between these two alternatives.

Without loss of generality we can assume that

$$\forall t_1, t_2 \in \mathbb{T} : t_1 \neq t_2 \Rightarrow var(t_1) \cap var(t_2) = \emptyset$$

is satisfied. Thus, there is no pair of tasks sharing a common variant. Furthermore, we generally assume that the different variants of a task are *computationally equivalent*. Thus, the evaluation of a variant of a task leads to the same result as any other variant of the same task. While a rigorous formalization of this property is beyond the scope of this paper and not essential for its content, we would like to point out one of its consequences: if any variant is terminating, all variants are required to do so.

In each program, an entry-point task will form the initial point of an application.

Definition 2.4 (Program)**.** A program is given by its entry point task $t_0 \in \mathbb{T}$. The set of all programs is denoted as $\mathbb{P} \subset \mathbb{T}$.

To accomplish their objectives, tasks can interact with the runtime system to request runtime-coordinated services. These operations are referred to as *actions*.

Definition 2.5 (Action)**.** The set of actions $\mathbb{A}$ is defined by

$$\mathbb{A} = \{spawn(t), sync(t), create(d), destroy(d), end\}$$

for all tasks $t \in \mathbb{T} \setminus \mathbb{P}$ and data items $d \in \mathbb{D}$.

Actions are service requests toward the runtime system triggered by tasks. The *spawn* action requests the runtime system to schedule a new task, while *sync* requests the suspension of the current task until a given task has been completed. The action *create* introduces a new data item to the runtime system, while its counterpart *destroy* requests the destruction of a data item. Finally, the action *end* signals the termination of the current task.

The following definition covers means to model the evaluation of tasks and the triggering of actions.

Definition 2.6 (Task Execution)**.** Let $\mathbb{S}$ be a set of abstract task-local execution states, $init : \mathbb{V} \to \mathbb{S}$ a function assigning each variant an initial state, and the function $step : \mathbb{V} \times \mathbb{S} \to \mathbb{S} \times \mathbb{A}$ describe the *transition function* of task variants.

A state $s \in \mathbb{S}$ summarizes the task-local state information maintained by a task e.g. on the stack or heap. Given a terminating task variant $v \in \mathbb{V}$, its execution trace is defined by a sequence of states $s_0, \ldots, s_n$, where $s_0 := init(v)$ and for all $0 \leq i < n$ we have $(s_{i+1}, a_{i+1}) := step(v, s_i)$, and $a_n = end$. The action sequence $a_1, \ldots, a_n$ is the sequence of commands issued to the runtime system.

Finally, to allow the runtime system to associate tasks with their required data, variants are needed to state their *data requirements*. Here, we distinguish between *read-only* and *read/write* access.

Definition 2.7 (Data Requirements)**.** Let $v \in \mathbb{V}$ be a variant and $d \in \mathbb{D}$ be a data item. The function $read : \mathbb{V} \times \mathbb{D} \to 2^{\mathbb{E}}$ obtains the set of elements $read(v, d) \subseteq elems(d) \subseteq \mathbb{E}$ in data item d during the execution of v. Correspondingly, the function $write : \mathbb{V} \times \mathbb{D} \to 2^{\mathbb{E}}$ obtains the set $write(v, d) \subseteq elems(d) \subseteq \mathbb{E}$ of updated elements in data item d.

Note that for the vast majority of pairs $(v, d) \in \mathbb{V} \times \mathbb{E}$ the read sets $read(v, d)$ and write sets $write(v, d)$ will be empty. Only for actually accessed data items this will not be the case.

Finally, w.l.o.g. we impose the following restrictions on tasks:

$$\exists f \in \mathbb{T} \to \mathbb{V} \times \mathbb{S} : \forall t \in \mathbb{T} \setminus \mathbb{P} : step(f(t)) = (s', spawn(t))$$

Thus, every task t that is not the entry point of a program has a unique spawn point $f(t)$.

2.3. Architecture Model

The third component of our model provides an abstract description of the hardware architecture. The key elements required for a functional description are compute units (e.g. CPU cores, GPUs,...) for processing tasks, memory address spaces (e.g. main memory, GPU device memory,...), and edges between those two to describe which compute unit can directly access data in which memory unit.

Definition 2.8 (Architecture Model)**.** Let C be a set of compute units, M be a set of address spaces, and $L \subseteq C \times M$ a set of links connecting compute units with accessible address spaces. Then the system model is given by the bipartite graph $(C \uplus M, L)$[1].

1. we use $\uplus$ to denote the union of disjoint sets

Example 2.4 (Architecture)**.** A distributed memory system comprising 2 nodes, each forming its own address space m_A and m_B, and being equipped with 4 CPU cores – $c_{A1}, \ldots, c_{A4}$ and $c_{B1}, \ldots, c_{B4}$ respectively – can be modeled as the bipartite graph $(C \uplus M, L)$ where $C = \{c_{A1}, \ldots, c_{B4}\}$, $M = \{m_A, m_B\}$ and $L = \{(c_{xi}, m_x) \mid x \in \{A, B\} \wedge 1 \leq i \leq 4\}$.

As for other components, we restricted the architecture model in this section to the strictly necessary functional details. In particular, we do not include network topology details or cache hierarchies in our model. Nevertheless, those details are considered by our implementation covered by Section 3.

2.4. Execution Model

The definitions so far covered static aspects of applications. To model the dynamic evolution of an application managed by the AllScale runtime system, the state space of the evaluation as well as state transitions are defined.

Definition 2.9 (System State)**.** The state of an application processed by the AllScale runtime system is given by a tuple

$$(Q, R, B, D, L_r, L_w, (C \uplus M, L))$$

where

- $Q \subseteq \mathbb{T}$ is a set of enqueued, yet not started tasks
- $R \subseteq C \times \mathbb{V} \times \mathbb{S}$ describes the state of running variant executions; an entry $(c, v, s) \in R$ describes a variant v running on compute unit c with its current state s
- $B \subseteq C \times \mathbb{V} \times \mathbb{S} \times \mathbb{T}$ lists suspended variants; an entry $(c, v, s, t) \in B$ describes a variant v with its state s waiting on compute unit c for the completion of task t
- $D \subseteq M \times \mathbb{D} \times \mathbb{E}$ describes the distribution state of data; an entry $(m, d, e) \in D$ states that element e of data item d is present in address space m
- $L_r \subseteq \mathbb{V} \times M \times \mathbb{D} \times \mathbb{E}$ enumerates data elements read locked; an entry (v, m, d, e) states that in address space m the element e of data item d is read locked for the duration of the execution of v
- $L_w \subseteq \mathbb{V} \times M \times \mathbb{D} \times \mathbb{E}$ analogous to L_r for write locks
- $(C \uplus M, L)$ the model of the hardware architecture a program is processed on

The set of all system states is denoted by $\mathcal{S}$.

Each state summarizes a snapshot of the management information to be maintained by the runtime system for processing an AllScale application at each moment in time. It covers the execution state of tasks, the distribution of data items, as well as active access permissions to data in the various address spaces.

To cover the dynamic behavior over time, valid state transitions are specified.

Definition 2.10 (State Transitions)**.** The binary *state transition* relation $\rightarrow$: $\mathcal{S} \times \mathcal{S}$ is defined by the inference rules enumerated in Fig. 2 and Fig. 3.

Each rule in Fig. 2 and Fig. 3 specifies the effect of an active or passive interaction of the processed application with the underlying AllScale runtime system. There are five task-scheduling related operations:

- *(start)* ... at any time the runtime system is allowed to take a task t from Q, pick one of its variants $v \in \mathit{var}(t)$, and start processing it on a compute unit c having v's data requirements satisfied; by doing so, all the data elements accessed by v get locked
- *(spawn)* ... during processing, any variant v may spawn a new task t, which gets enqueued in Q
- *(sync)* ... variants may also synchronize on other tasks, moving the synchronizing variant from the set of running variants R to the set of blocked variants B
- *(continue)* ... whenever the runtime system discovers that the task t a blocked variant v is waiting on has been completed, it may continue processing v by moving it back to R
- *(end)* ... once a task is completed, its state information is discarded and its held data element locks are released

Furthermore, five additional rules cover data management issues:

- *(create)* ... tasks may dynamically create new data items during execution; initially no locks will be granted, nor will memory space be allocated
- *(init)* ... the runtime may, at any time, allocate memory in address spaces for data elements not yet allocated anywhere throughout the system
- *(migrate)* ... the runtime may also move data from a source memory space m_s to a destination memory space m_d in case no locks are currently held on the corresponding elements
- *(replicate)* ... the runtime may, furthermore, replicate data in case there is no write lock on the source locations
- *(destroy)* ... tasks may release data items when no longer needed; all associated data elements and locks are deleted

Operations *spawn*, *sync*, *end*, *create*, and *destroy* are triggered by the processing of tasks, while *start*, *continue*, *init*, *migrate*, and *replicate* are controlled by the runtime system. While the runtime system has to react upon the former, the latter can be utilized to (pro-)actively enforce scheduling and data management policies.

For clarity and brevity we assume a static architecture model in this section. Extension of our model covering dynamic environments where compute nodes may join or leave (crash) can be formulated, but exceed the scope of this paper.

Finally, the execution of a program is modeled by its traces.

Definition 2.11 (Trace)**.** Let $(C \uplus M, L)$ be an architecture and $t_0 \in \mathbb{P}$ be a program. A trace of t_0 is a sequence $s_0, s_1, \ldots \in \mathcal{S}$ where $s_0 = (\{t_0\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, (C \uplus M, L))$ and $\forall i > 0 : s_{i-1} \rightarrow s_i$. A trace *terminates* by reaching a state $s_t = (\emptyset, \emptyset, \emptyset, D_t, \emptyset, \emptyset, (C \uplus M, L))$ for some $D_t \subseteq M \times \mathbb{D} \times \mathbb{E}$.

$$\frac{t \in Q \quad v \in \mathit{var}(t) \quad \exists c \in C : \exists m \in \mathbb{D} \to M : \forall d \in \mathbb{D} : (c, m(d)) \in L \wedge \forall e : \mathit{read}(v,d) \cup \mathit{write}(v,d) : (m(d), d, e) \in D \quad D \cap D_w = \emptyset}{(Q, R, B, D, L_r, L_w, (C \uplus M, L)) \to (Q \setminus \{t\}, R \uplus \{(c, v, \mathit{init}(v))\}, B, D, L'_r, L'_w, (C \uplus M, L))} \ (start)$$

where $D_w = \bigcup_{d \in \mathbb{D}} \{(m, d, e) \mid m \in M \setminus \{m(d)\} \wedge e \in \mathit{write}(v, d)\}$
and $L'_r = L_r \uplus \bigcup_{d \in \mathbb{D}} \{(v, m(d), d, e) \mid e \in \mathit{read}(v, d)\}$
and $L'_w = L_w \uplus \bigcup_{d \in \mathbb{D}} \{(v, m(d), d, e) \mid e \in \mathit{write}(v, d)\}$

$$\frac{(c, v, s) \in R \quad \mathit{step}(v, s) = (s', \mathit{spawn}(t))}{(Q, R, B, D, L_r, L_w, (C \uplus M, L)) \to (Q \cup \{t\}, (R \setminus \{(c, v, s)\}) \uplus \{(c, v, s')\}, B, D, L_r, L_w, (C \uplus M, L))} \ (spawn)$$

$$\frac{(c, v, s) \in R \quad \mathit{step}(v, s) = (s', \mathit{sync}(t))}{(Q, R, B, D, L_r, L_w, (C \uplus M, L)) \to (Q, R \setminus \{(c, v, s)\}, B \uplus \{(c, v, s', t)\}, D, L_r, L_w, (C \uplus M, L))} \ (sync)$$

$$\frac{(c, v, s, t) \in B \quad t \notin Q \quad \nexists v' \in \mathit{var}(t) : \exists (c, s, t) : (c, v', s) \in R \vee (c, v', s, t) \in B}{(Q, R, B, D, L_r, L_w, (C \uplus M, L)) \to (Q, R \uplus \{(c, v, s)\}, B \setminus \{(c, v, s, t)\}, D, L_r, L_w, (C \uplus M, L))} \ (continue)$$

$$\frac{(c, v, s) \in R \quad \mathit{step}(v, s) = (s', \mathit{end}) \quad L_v = \{v\} \times M \times \mathbb{D} \times \mathbb{E}}{(Q, R, B, D, L_r, L_w, (C \uplus M, L)) \to (Q, R \setminus \{(c, v, s)\}, B, D, L_r \setminus L_v, L_w \setminus L_v, (C \uplus M, L))} \ (end)$$

Figure 2: Task related state transition rules.

$$\frac{(c, v, s) \in R \quad \mathit{step}(v, s) = (s', \mathit{create}(d))}{(Q, R, B, D, L_r, L_w, (C \uplus M, L)) \to (Q, (R \setminus \{(c, v, s)\}) \uplus \{(c, v, s')\}, B, D, L_r, L_w, (C \uplus M, L))} \ (create)$$

$$\frac{m \in M \quad d \in \mathbb{D} \quad E \subseteq \mathit{elems}(d) \quad E \neq \emptyset \quad D \cap (M \times \{d\} \times E) = \emptyset}{(Q, R, B, D, L_r, L_w, (C \uplus M, L)) \to (Q, R, B, D \uplus (\{m\} \times \{d\} \times E), L_r, L_w, (C \uplus M, L))} \ (init)$$

$$\frac{d \in \mathbb{D} \quad E \subseteq \mathit{elems}(d) \quad E \neq \emptyset \quad m_s, m_d \in M \quad (L_r \cup L_w) \cap (\mathbb{V} \times \{m_s, m_d\} \times \{d\} \times E) = \emptyset}{(Q, R, B, D, L_r, L_w, (C \uplus M, L)) \to (Q, R, B, (D \setminus (\{m_s\} \times \{d\} \times E)) \cup (\{m_d\} \times \{d\} \times E), L_r, L_w, (C \uplus M, L))} \ (migrate)$$

$$\frac{d \in \mathbb{D} \quad E \subseteq \mathit{elems}(d) \quad E \neq \emptyset \quad m_s, m_d \in M \quad L_w \cap (\mathbb{V} \times \{m_s\} \times \{d\} \times E) = \emptyset \quad (L_r \cup L_w) \cap (\mathbb{V} \times \{m_d\} \times \{d\} \times E) = \emptyset}{(Q, R, B, D, L_r, L_w, (C \uplus M, L)) \to (Q, R, B, D \cup (\{m_d\} \times \{d\} \times E), L_r, L_w, (C \uplus M, L))} \ (replicate)$$

$$\frac{(c, v, s) \in R \quad \mathit{step}(v, s) = (s', \mathit{destroy}(d)) \quad L_d = \mathbb{V} \times M \times \{d\} \times \mathbb{E}}{(Q, R, B, D, L_r, L_w, (C \uplus M, L)) \to (Q, (R \setminus \{(c, v, s)\}) \uplus \{(c, v, s')\}, B, D \setminus (M \times \{d\} \times \mathbb{E}), L_r \setminus L_d, L_w \setminus L_d, (C \uplus M, L))} \ (destroy)$$

Figure 3: Data related state transition rules.

In our model each operation is atomic and no state transition may overlap with others. Although modeling the concurrent execution of tasks, this eliminates any parallelism. To utilize parallel resources, implementations are allowed to perform overlapping transitions. However, the observable behavior of a program executed in parallel must be equivalent to the observable result of some sequential trace of the program.

2.5. Model Properties

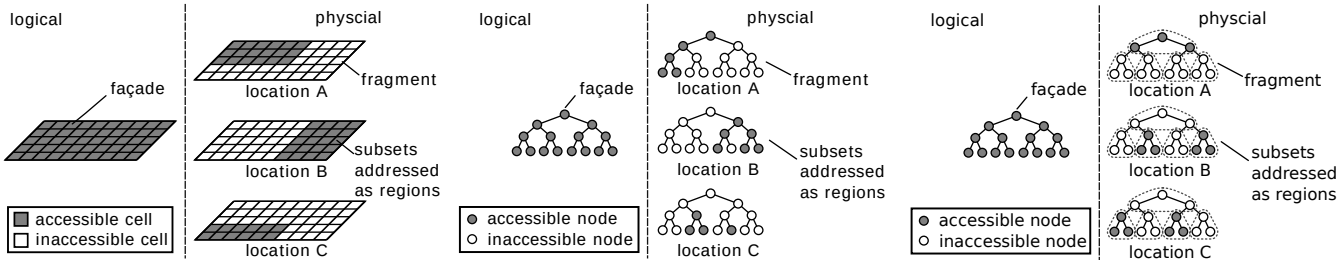
Beside others, the following properties can be proven for our model (sketches for these proofs can be found in Appendix A):

- *single-execution:* in a terminating trace, for the entry point and each spawned task exactly one variant will be selected and processed exactly once
- *termination:* if a deadlock-free program has a terminating trace, all of its traces not including infinite initialization, migration, and replication sequences will eventually terminate
- *satisfied requirements:* variants are only processed on compute units where all required data is available for the duration of their processing
- *exclusive writes:* a data element being write locked in some memory address space is not replicated anywhere else in the system at the same time, nor can such replicates be created
- *data preservation:* the runtime system cannot delete data that is not explicitly destroyed; the runtime can, however, remove replicated data

In particular the *termination* property ensures that sensible scheduling of runtime operations does not influence the termination of a program. The *exclusive writes* property, on the other hand, ensures that the runtime system can not introduce race conditions through scheduling and/or data management decisions.

3. Implementation

The AllScale environment provides an implementation of our application model based on C++. It comprises three



(a) regular 2D grid (b) binary tree, flexible regions (c) binary tree, blocked regions

Figure 4: Example data item organizations.

major components: implementations of templated data structures facilitating instances to be managed as data items, a runtime system utilizing those to coordinate the processing of programs on distributed memory systems, and a source-to-source compiler component converting a high-level task API into the format required by our model.

3.1. Data Items

Our C++ implementations of data items are required to provide three components as illustrated by Fig. 4: a façade type, a fragment type, and a region type. The façade type defines the logical view on the data structure to the end user, i.e. the application developer. It provides data-structure-specific operations, like field accesses or iterators. The fragment type, on the other hand, is the runtime's view on the data structure. Fragments are capable of maintaining subsets of elements of a data structure within some address space. Finally, regions provide the necessary means to address the subset of elements maintained within fragments, as introduced by Definition 2.2.

A large variety of data structures, ranging from simple scalars, ordinary arrays, over multi-dimensional grids, various types of trees, graphs, sets, and maps can be implemented using this interface. The key element for the efficient distribution of those, however, is the region type – thus the means to address subsets of elements.

Region types have to satisfy several criteria: first, they have to be closed under union, intersection, and set-difference. Thus, for instance, using simple axis aligned bounding boxes for describing regions in e.g. a 2D grid would not be sufficient, since boxes are neither closed under union nor set-difference. Second, representations ought to be efficient, both in space and runtime complexity. Thus, explicit element enumerations, while technically sound, are less practical. Finally, region types must be able to accurately express regions of interest for the algorithm applied on the associated data structure.

The last criteria implies that there is not a single ideal region type for every kind of data structure. There might be several different alternatives application developers may chose from, to adapt the data item implementation to their needs – similar to choosing between e.g. linked lists and array lists for a respective use case when performing algorithmic optimizations.

Fig. 4 outlines three example data item implementations provided by our prototype implementation. Fig. 4a illustrates a 2D version of our N-dimensional grid implementation, utilizing sets of axis-aligned bounding boxes to describe regions. Unlike individual boxes, sets are closed under intersection and set-difference and are thus valid region types.

Fig. 4b and Fig. 4c outline the structure of two binary tree data items, equipped with different region schemes. In Fig. 4b regions are defined through two sets of sub-trees, each identified by its respective root node. The first set enumerates included sub-trees, while the second set enumerates excluded sub-trees nested within the included trees. Thus, the data partitioning illustrated in Fig. 4b can be represented by listing at most three nodes to characterize the regions covered by the individual regions. This scheme provides the flexibility to express arbitrary node distributions among tree fragments.

However, in some cases the flexibility provided by the scheme of Fig. 4b is not required. More coarse grained blocking like outlined in Fig. 4c might be sufficient. In this scheme, the overall tree is divided into one root tree of height h and 2^h sub-trees. Thus, a simple bit-mask of length $2^h + 1$ is sufficient to model regions, providing a much more efficient scheme, yet less flexible distribution options. Depending on the algorithm though, those might not be required.

Data item implementations, as well as a set of parallel algorithms applicable on them are provided by the AllScale API [2]. The AllScale API is a small header-only library associated to the AllScale compiler and our runtime system providing a user interface to develop applications utilizing the provided infrastructure.

3.2. Runtime System

The AllScale Runtime System implementation [3] is based on the HPX distributed memory runtime system [4]. HPX offers a task based parallel programming library, handling the scheduling of tasks in a distributed memory environment as well as means for services globally addressable through remote procedure calls. By default, the HPX runtime system maintains a single process per node within a distributed memory cluster. Each of these processes manages a pool of worker threads, to harness intra-node parallelism.

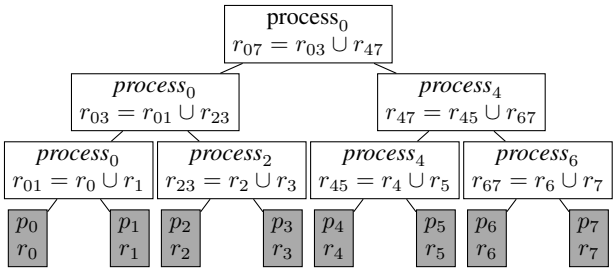


Figure 5: Hierarchical, distributed data storage index.

Communication between processes is realized through a compact, exchangeable communication layer. MPI, plain TCP, or *libfabric* based implementations are available.

Our runtime system prototype extends upon HPX by adding a *data item manager*, an adapted task scheduler, an extended monitoring infrastructure [5], and a resilience manager [6] – the latter two are possible due to the AllScale runtime model, but their details are beyond the scope of this paper.

The objective of the data item manager and the scheduler is to maintain a consistent view on the system state, react to task-triggered actions and steer the program execution by scheduling tasks and managing the distribution of data.

As covered in the previous section, the overall state information to be maintained by the runtime system is given by the tuple

$$(Q, R, B, D, L_r, L_w, (C \uplus M, L))$$

The overall state is maintained in a distributed fashion throughout the system, exploiting locality. Enqueued tasks (Q) are stored within node-local queues at the locality where they have been created, yet may be stolen by other nodes. Running and blocked tasks (R and B) are equally maintained within node-local structures, but may not be moved to other nodes since their task-private state can not be migrated.

The data distribution D is managed by keeping track of locally present regions of data items. Thus, a data item manager instance in each AllScale process maintains fragments of data items and actively manages contained data by performing *resizing*, *import*, and *export* operations. Furthermore, the data item manager keeps track of the lock states L_r and L_w of locally maintained data item regions. Finally, information regarding the hardware model $(C \uplus M, L)$ is maintained by the underlying HPX system.

When scheduling application tasks, in particular in the context of a *start* transition, the runtime is frequently tasked with locating regions of data items being distributed throughout the system. For instance, before being allowed to start a variant v of a task all data required by v must be located and moved to a common compute locality or — preferably — the variant v must be moved to a locality where the required data is already present. To speed up the process of locating required data, a distributed index structure as outlined by Fig. 5 is maintained.

Algorithm 1 Region location resolution.

Input: $d \in \mathbb{D}$ ...a data item
Input: $r \in \mathbb{R}$ ...a region of d to be located
Output: $m \subseteq \mathbb{R} \times \mathbb{N}$ ...a relation mapping region segments to hosting process IDs, such that $\bigcup_{(x,j) \in m} x \subseteq r$

```
 1: function LOOKUP(d,r)
 2:   return local_process.RESOLVE(d,r,1)
 3: end function
 4:
 5: function RESOLVE(d,r,l)
 6:   i := <local process ID>
 7:   m := ∅
 8:   if l == 1 then
 9:     // leaf level - add local share to result
10:     r_i = <region of d covered by local process i>
11:     if r ∩ r_i ≠ ∅ then
12:       m := m ⊎ {(r ∩ r_i, i)}
13:       r := r \ r_i
14:     end if
15:   else
16:     // inner level - check children
17:     r_l = <region of d covered by left subtree of process i on level l>
18:     if r ∩ r_l ≠ ∅ then
19:       m := m ⊎ process[i].RESOLVE(d,r,l − 1)
20:       r := r \ r_l
21:     end if
22:     r_r = <region of d covered by right subtree of process i on level l>
23:     if r ∩ r_r ≠ ∅ then
24:       m := m ⊎ process[i + 2^(l−1)].RESOLVE(d,r,l − 1)
25:       r := r \ r_r
26:     end if
27:   end if
28:
29:   // if fully resolved => done
30:   if r = ∅ then return m end if
31:
32:   // escalate to parent
33:   if l is not the root level then
34:     m := m ⊎ process[2^l ⌊i/2^l⌋].RESOLVE(d,r,l + 1)
35:   end if
36:
37:   // done
38:   return m
39: end function
```

All AllScale runtime processes are organized in a binary hierarchy such that process i is the child of process $2^l \lfloor i/2^l \rfloor$ on level l, where level 1 is the leaf level. Note that due to this arrangement, the role of inner nodes is assumed by the left child of those nodes in the hierarchy. Each leaf node stores the region covered by its locally present data item fragments, while inner nodes maintain the regions covered by their left and right sub-trees. Thus, each process has to maintain up to $\mathcal{O}(\log_2(P))$ regions, where P is the number of involved processes. For each managed data item, a separate hierarchical index is maintained.

Algorithm 1 outlines the distributed procedure initiated whenever the locality of a region r of a data item d has to be obtained by a process. The lookup request is forwarded to a recursive tree-traversal algorithm starting on the leaf level (line 2). In each step of the traversal, it is first tested whether the currently visited node in the process hierarchy is a leaf node or inner node (line 8). In case of a leaf node, the locally maintained region of a data item is compared to

Algorithm 2 Inter-process task scheduling.

Input: $t \in \mathbb{T}$ ...task to be scheduled

```
1: procedure ASSIGN_TO_NODE(t)
2:   // select the variant to be processed
3:   v = scheduler_policy.PICK_VARIANT(t)
4:   if ∃ process i : all requirements of v are covered by i then
5:     // schedule task on node fulfilling all requirements
6:     process[i].enqueue(v)
7:   else if ∃ process i : all write-reqs. of v are covered by i then
8:     // schedule task on node fulfilling all write requirements
9:     process[i].enqueue(v)
10:  else
11:    // let scheduling policy decide where to place task
12:    i := scheduler_policy.PICK_TARGET(v)
13:    process[i].enqueue(v)
14:  end if
15: end procedure
```

the requested region (line 11). If so, corresponding locality information is added to the result (line 12) and the set of remaining elements to be located is reduced accordingly (line 13). In case of visiting an inner node, the left and right sub-trees are consulted for their contributions (lines 16-26). If after processing the current node all requested elements could be resolved, the traversal is terminated (line 30). Otherwise the parent node is consulted (line 34).

Algorithm 1 constitutes a greedy heuristic for obtaining a list of sources to retrieve data from when being tasked with coalescing a given region in a single address space. It is thus utilized for creating replicas of read-only data required by a task variant whenever its target location has been fixed. The variant of a task to be processed as well as the locality it ought to be processed on is determined by a customized, data requirement aware scheduler algorithm.

Algorithm 2 provides a high-level overview of the currently implemented task distribution heuristic utilized by our prototype implementation. Whenever a task is scheduled, in a first step a customizable scheduling policy is consulted to select the variant to be executed (line 3). This policy considers the set of available variants, properties of those like being sequential or spawning additional sub-tasks, as well as runtime system data like task queue lengths and worker idle rates. Once a variant is selected, it is dispatched to a process fulfilling all data requirements (line 6) or, if not available, to a node covering all write requirements (line 9). If neither of those is available, the scheduling policy will be once more consulted to select a desirable locality (line 12) to which the task is finally forwarded (line 13).

The scheduling policy is responsible for obtaining adequate task granularity and load distribution throughout the system. In particular during the initialization phase of applications, it is responsible for spreading out tasks such that data items get evenly distributed throughout the system. In later phases, by monitoring the workload distribution among various processes, the scheduling policy may decide to migrate data between nodes, which will implicitly lead to the redirection of future tasks to the newly designated localities. Thus, inter-node load balancing is achieved through actively managing the distribution of data.

3.3. Compiler

The last component of our prototype implementation is the AllScale compiler [7], a source-to-source compiler based on the Insieme compiler framework [8], [9]. Its basic role is to convert an input program using AllScale's high-level parallel C/C++ API based on the *prec* operator [10] into application code fitting the model expected by the runtime system. Its major tasks are:

- the identification of parallel tasks in the input code and the generation of code variants for each of those; for each task a serial and parallel implementation variant is made available to the runtime system where possible
- the integration of data requirements by associating a function computing requirements with each code variant; data requirements are obtained through high-level static program analysis based on Insieme's analysis framework [11]
- the restructuring of user code addressing data item façades to interface with the runtime's data item manager; thus replace user managed data items by runtime managed instances
- the addition of serialization code for user defined types to facilitate the migration of data

As a source-to-source compiler, the AllScale compiler converts a user provided C++ input program into C++ target code to be compiled against the AllScale runtime system. The resulting binary can be executed on a given target architecture like any other HPX application.

3.4. Example Application

Fig. 6a outlines a simple, sequential version of a two-dimensional stencil kernel [12] implemented in C++, as it might be present in many physics applications that solve e.g. heat diffusion equations. Lines 1 and 2 allocate two buffers, to which line 3 obtains pointers. The first loop nest (lines 5–8) initializes the first buffer, before the time loop starting on line 10 performs the actual simulation of the diffusion process. In each time step the loops in lines 11–17 update each element in the buffer based on the state of the same element and its neighbors in the previous time step. Finally, line 18 ensures that after each time step the role of buffers `A` and `B` is swapped.

The AllScale version of Fig. 6b has been derived from the sequential version by applying two modifications: First, the initializer loop nest (lines 5–7) and update loop nest (lines 12–17) have been parallelized using the 2D version of the `pfor` function provided by the AllScale API. Second, the underlying array data structures have been exchanged by API-provided two-dimensional `Grid` instances implementing the data item interface. Within these grids, regions are addressed through sets of axis-aligned bounding boxes (see Fig. 4a).

This example demonstrates that user applications can be ported to our model without increasing code complexity.

```
double fieldA[N][N];
double fieldB[N][N];
auto A = fieldA, B = fieldB;
// initialize the field
for(int x = 0 ; x < N ; ++x) {
  for(int y = 0 ; y < N ; ++y) {
    A[x][y] = ... ; // some value
} }
// gradually compute the solution
for(int t = 0; t < T; ++t) {
  for(int x = 1 ; x < N-1 ; ++x) {
    for(int y = 1 ; y < N-1 ; ++y) {
      B[x][y] = A[x][y] + c * (
        A[x][y-1] + A[x][y+1] +
        A[x-1][y] + A[x+1][y] - 4*A[x][y]
      );
  } }
  swap(A,B);
}
// the solution is now stored in A
```

(a) sequential version

```
Grid<double,2> A({N,N}); // 2D grid
Grid<double,2> B({N,N}); // 2D grid

// initialize the field
pfor({0,0},{N,N},[&](auto p) {
  A[p] = ... ; // some value
});

// gradually compute the solution
for(int t = 0; t < T; ++t) {

  pfor({1,1}, {N-1,N-1}, [&](auto p) {
    B[p] = A[p] + c * (
      A[{p.x,p.y-1}] + A[{p.x,p.y+1}] +
      A[{p.x-1,p.y}] + A[{p.x+1,p.y}] - 4*A[p]
    );
  });
  swap(A,B);
}
// the solution is now stored in A
```

(b) parallel, distributed memory version

Figure 6: Comparison of 2D stencil implementations.

4. Evaluation

To evaluate the capabilities of our model, we examine the performance of three applications: stencil, an established micro-benchmark; iPiC3D, a real-world particle-in-cell simulator; and two-point correlation (TPC), a data-mining primitive based on tree traversals.

4.1. Setup

The stencil application has been derived from the Parallel Research Kernels [12] and was already introduced in Section 3.4. The second, iPiC3D [13], simulates the behavior of charged particles interacting with each other in the presence of electromagnetic fields. The data structures used for this simulation are three regular 3D grids — two holding electromagnetic field data, while an additional grid holds lists of particles. The third application, TPC [14], is a two-point correlation benchmark that computes the number of points within a certain distance of a given query point in 7D space. For each query, TPC performs a pruned, parallel kd-tree traversal. It is widely used in statistics and data mining. We ported each of our three applications to the AllScale model and MPI to provide a reference.

Table 1 summarizes our three applications by outlining their central data structures, the problem sizes used in our evaluation, and the collected performance metric.

For our evaluation we used up to 64 nodes inside the RRZE Meggie Cluster[2]. Each node is equipped with two Intel Xeon E5-2630 v4 processors, 10 cores each, and 64 GB of main memory. The nodes are connected via Intel OmniPath in a fat tree topology. All codes are compiled with GCC 7.3.0 using Intel MPI 2018.2.

4.2. Results and Discussion

Fig. 7 summarizes our evaluation. For stencil and iPiC3D, the results of the AllScale and MPI versions show comparable performance and scalability. For TPC, however, MPI obtains higher performance, while AllScale can only gain performance improvements up to 8 nodes.

For all three benchmarks, our prototype implementation manages to effectively distribute the user-defined data structures among multiple nodes. Furthermore, the results of the first two applications demonstrate that the implicit data management scheme employed by our model can provide performance comparable to state-of-the-art MPI based implementations depending on explicit user-managed data distributions. Thus, our system's increased control on an application does not incur an inherent performance penalty.

In principle, the same holds true for the TPC benchmark. However, unlike the first two, TPC spawns a large number of inherently small tasks to be forwarded to localities owning traversed kd-tree nodes. The resulting high inter-node communication overhead for transferring tasks diminishes overall performance and grows dominant for larger node counts. To mitigate, the MPI version aggregates multiple queries to reduce latency sensitivity and improve bandwidth utilization. However, such an optimization, while technically possible, has not yet been integrated into our prototype.

5. Related Work

Conventional, low-level HPC infrastructures comprising combinations of MPI, OpenMP, OpenCL, CUDA, and Cilk are the de facto standard for building HPC applications, despite their lack of higher level data management services. When utilizing those, the decomposition and distribution management of data structures has to be incorporated into the application code. Thus, the developers' view on data structures as a self-contained entity is abandoned and replaced by the explicit handling of subsections of those data structures distributed among various address spaces.

Programming models based on Partitioned Global Address Space (PGAS) attempt to mitigate on the latter by providing a global, shared address space within which data

2. https://www.anleitungen.rrze.fau.de/hpc/meggie-cluster/

TABLE 1: List of target application codes.

Name	Description	Data Structure	Problem Size	Performance Metric
stencil	2D stencil kernel [12]	regular 2D grid	$20,000^2$ elements per node	FLOPS
iPiC3D	particle-in-cell simulator [13]	multiple regular 3D grids	$48 \cdot 10^6$ particles per node	particle updates per second
TPC	two-point-correlation search [14]	kd-tree	2^{29} points in $[0, 100)^7$ with radius 20	queries per second

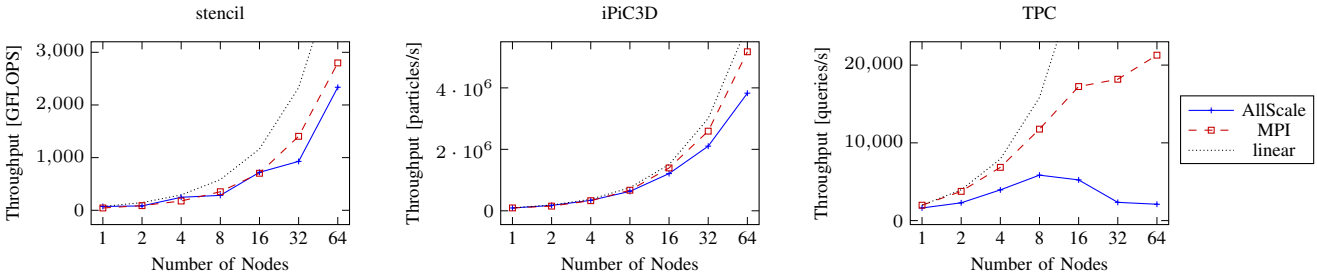


Figure 7: Throughput scaling of the three target applications.

structures can be placed without the need for explicit deconstruction. Languages like Unified Parallel C [15] or Coarray Fortran [16] implement this model by extending their host languages with *partitioned* arrays whose elements are statically distributed throughout multiple physical address spaces. Higher level user-defined data structures or the dynamic redistribution of data structures are not supported. More sophisticated PGAS incarnations such as Chapel [17] and X10 [18] introduce the support for user-defined data structures in PGAS environments, yet managing the distribution of those remains the developers' obligation.

On the hardware level, virtual shared memory systems [19] provide solutions where the data distribution among physical address spaces is managed on a memory page level granularity. Multiple physical address spaces are connected to one large virtual address space that applications may transparently access. While capable of processing arbitrary applications without the need of any modifications, the lack of insights into managed data structures and synchronization granularity requirements of processed algorithms causes scalability issues beyond a few thousand cores.

On the application level, programming models taking complete control over the data management have been introduced. Systems like Spark [20] or Hadoop [21] are centered around the management of data on which operations may be applied on. However, the structure of data and algorithms to be applied is limited.

DSL development frameworks such as Lift [22], Delite [23], and AnyDSL [24] provide environments for the implementation of high performance DSLs. In each of those, data management is handled effectively by the system. Yet, the range of supported data structures and operations is limited by the design of the corresponding DSL.

More general purpose parallel C++ based frameworks like the RAJA portability layer [25] or PHAST [26] utilize C++' modern features to provide a higher level abstraction for achieving portability among underlying parallel APIs such as OpenMP, Cilk, OpenCL and CUDA. Also new standards like SYCL [27] utilize the same C++ features to improve maintainability of heterogeneous, high-performance code bases. However, while abstracting away compiler, API, or platform specific directives to enable portability, none of these offer data structure distribution capabilities.

Data focused parallel C++ library based frameworks like STAPL [28] and Kokkos [29] are exercising control over parallel algorithms and data structures similar to our architecture. Both provide fixed sets of library-defined, (multidimensional) array focused generic containers whose distribution is managed by the underlying runtime system. Neither utilizes abstractions opening up the option of integrating a more general class of structures as supported by our model.

In our own evolutionary lineage, the AllScale project improves upon the concepts developed by the Insieme compiler and runtime system project [8], by generalizing towards user-definable data structures. While core compiler features are reused, the runtime system is replaced with the HPX runtime system [4]. HPX on its own, however, does not include automated data management.

6. Conclusion and Future Work

The AllScale Application Model provides novel control over the distribution of user-defined data structures to underlying runtime systems. This control opens up the possibility of integrating higher-level services, including the transparent exchange and migration of data and tasks as well as inter-node load balancing support, into generic, application-independent runtime systems. Our prototype implementation establishes the practical realizability of our approach, showcasing, in particular, the usability advantages of our system from an HPC application developer's point of view. Furthermore, our iPiC3D port demonstrates its applicability to real-world use cases.

Current development efforts aim at closing the performance gap to handcrafted MPI-based implementations. Furthermore, techniques for inter-node load balancing and runtime system based task checkpointing are the subject of ongoing investigations.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme as part of the FETHPC AllScale project under grant agreement No 671603. This research used resources of the Regionales RechenZentrum Erlangen (RRZE).

References

[1] AllScale Consortium, "AllScale," 2018. [Online]. Available: http://www.allscale.eu

[2] ——, "AllScale API," 2018. [Online]. Available: https://github.com/allscale/allscale_api

[3] ——, "The AllScale Runtime System," 2018. [Online]. Available: https://github.com/allscale/allscale_runtime

[4] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. New York, NY, USA: ACM, 2014, pp. 6:1–6:11.

[5] X. Aguilar, "On-Demand, On-Line Monitoring Infrastructure (a)," AllScale Consortium, Deliverable D5.2, 2017. [Online]. Available: http://www.allscale.eu/docs/D5.2%20-%20On-Demand,%20On-Line%20Monitoring%20Infrastructure%20(a).pdf

[6] K. Dichev and C. Gillan, "Resilience Manager," AllScale Consortium, Deliverable D5.7, 2018. [Online]. Available: http://www.allscale.eu/docs/D5.7%20Resilience%20Manager.pdf

[7] AllScale Consortium, "AllScale Compiler," 2018. [Online]. Available: https://github.com/allscale/allscale_compiler

[8] University of Innsbruck, "Insieme Compiler Project," 2018. [Online]. Available: http://www.insieme-compiler.org

[9] H. Jordan, "Insieme - A Compiler Infrastructure for Parallel Programs," Ph.D. dissertation, University of Innsbruck, 2014.

[10] H. Jordan, P. Thoman, P. Zangerl, T. Heller, and T. Fahringer, "A Context-aware Primitive for Nested Recursive Parallelism," in *Fifth International Workshop on Multicore Software Engineering (IWMSE16)*. Berlin, Heidelberg: Springer, 2016, pp. 1–12.

[11] A. Hirsch, "Insieme's Haskell-based Analysis Toolkit," Master's thesis, University of Innsbruck, Innsbruck, Austria, 2017.

[12] R. F. V. der Wijngaart and T. G. Mattson, "The parallel research kernels," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2014, pp. 1–6.

[13] S. Markidis, G. Lapenta, and Rizwan-uddin, "Multi-scale simulations of plasma with ipic3d," *Math. Comput. Simul.*, vol. 80, no. 7, Mar. 2010.

[14] A. G. Gray and A. W. Moore, "'n-body' problems in statistical learning," in *Proceedings of the 13th International Conference on Neural Information Processing Systems*, ser. NIPS'00. Cambridge, MA, USA: MIT Press, 2000, pp. 500–506. [Online]. Available: http://dl.acm.org/citation.cfm?id=3008751.3008824

[15] T. El-Ghazawi and L. Smith, "Upc: Unified parallel c," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188483

[16] A. Shterenlikht, L. Margetts, L. Cebamanos, and D. Henty, "Fortran 2008 coarrays," *SIGPLAN Fortran Forum*, vol. 34, no. 1, pp. 10–30, Apr. 2015. [Online]. Available: http://doi.acm.org/10.1145/2754942.2754944

[17] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2005, pp. 519–538.

[19] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, Nov. 1989. [Online]. Available: http://doi.acm.org/10.1145/75104.75105

[20] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[21] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[22] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A Functional Data-parallel IR for High-performance GPU Code Generation," in *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*. IEEE, 2017, pp. 74–85.

[23] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A Heterogeneous Parallel Framework for Domain-specific Languages," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 89–100.

[24] Saarland University, "AnyDSL," 2018. [Online]. Available: https://anydsl.github.io

[25] R. D. Hornung and J. A. Keasler, "The RAJA Portability Layer: Overview and Status," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2014.

[26] B. Peccerillo and S. Bartolini, "PHAST Library—Enabling Single-Source and High Performance Code for GPUs and Multi-cores," in *High Performance Computing & Simulation (HPCS), 2017 International Conference on*. IEEE, 2017, pp. 715–718.

[27] Khronos OpenCL Working Group, "SYCL Specification 1.2.1," Khronos OpenCL Working Group, Tech. Rep., 2017.

[28] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "STAPL: An Adaptive, Generic Parallel C++ Library," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2001, pp. 193–208.

[29] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

Appendix A.
Proof Sketches

In this appendix section we provide proof sketches for the model properties stated in Section 2.5. It is intended for the reviewers to get insights on the utility of our formal model. Unlike other sections, as an extra, this appendix did not go through the strict quality checks we apply on other sections. We apologize for any mistakes.

A.1. Auxiliary Definitions

For outlining our proof sketches additional definitions are required to facilitate a concise notation. In a first step we define the following short-cut functions to address components of system state tuples.

Definition A.1 (State Component Accessors)**.** Let

$$s = (Q, R, B, D, L_r, L_w, (C \uplus M, L)) \in \mathcal{S}$$

be an abbreviation for an arbitrary system state. Then, the function $q : \mathcal{S} \to 2^{\mathbb{T}}$ is defined by

$$q(s) = Q$$

the function $r : \mathcal{S} \to 2^{C \times \mathbb{V} \times \mathbb{S}}$ is defined by

$$r(s) = R$$

the function $b : \mathcal{S} \to 2^{C \times \mathbb{V} \times \mathbb{S} \times \mathbb{T}}$ is defined by

$$b(s) = B$$

the function $v : \mathcal{S} \to 2^{\mathbb{V}}$ is defined by

$$v(s) = \{v \in \mathbb{V} | \exists c, s, t : (c, v, s) \in r(s) \vee (c, v, s, t) \in b(s)\}$$

the function $d : \mathcal{S} \to 2^{M \times \mathbb{D} \times \mathbb{E}}$ is defined by

$$d(s) = D$$

the function $l_r : \mathcal{S} \to 2^{\mathbb{V} \times M \times \mathbb{D} \times \mathbb{E}}$ is defined by

$$l_r(s) = L_r$$

the function $l_w : \mathcal{S} \to 2^{\mathbb{V} \times M \times \mathbb{D} \times \mathbb{E}}$ is defined by

$$l_w(s) = L_w$$

and the function $l : \mathcal{S} \to 2^{\mathbb{V} \times M \times \mathbb{D} \times \mathbb{E}}$ is defined by

$$l(s) = l_w(s) \cup l_r(s)$$

We also require additional notation for state transitions.

Definition A.2 (State Transition Utilities)**.** The relation $\to^*$: $\mathcal{S} \times \mathcal{S}$ is the reflexive transitive closure of the state transition relation $\to$. Furthermore, let $\to^p$: $\mathcal{S} \times \mathcal{S}$ be defined by the inference rules *(start)*, *(spawn)*, *(sync)*, *(continue)*, *(end)*, *(create)*, and *(destroy)* of Fig. 2 and Fig. 3, and the relation $\to^{*p} = \to^* \times \to^p$.

Thus, the relation $\to^*$ models an arbitrary number of transition steps (including none), the relation $\to^p$ covers state transitions involving progressing tasks, and $\to^{*p}$ an arbitrary sequence of transitions ended by a progress transition.

Furthermore, we define the following auxiliary functions for traces.

Definition A.3 (Trace Utilities)**.** The function $\mathit{start} : \mathbb{P} \to \mathcal{S}$ is defined by

$$\mathit{start}(t) = (\{t_0\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, (C \uplus M, L))$$

and the set F of terminal states is defined as

$$F = \{s \in \mathcal{S} \mid q(s) = r(s) = b(s) = l(s) = \emptyset\} \subset \mathcal{S}$$

Let $\mathcal{T}$ define the set of all traces. Furthermore, the function $\mathit{traces} : \mathbb{P} \to 2^{\mathcal{T}}$ is defined by

$$\mathit{traces}(p) = \{[s_0, s_1, \ldots] \in \mathcal{T} \mid s_0 = \mathit{start}(p)\}$$

the function $\mathit{full\_traces} : \mathbb{P} \to 2^{\mathcal{T}}$ is defined by

$$\mathit{full\_traces}(p) = \left\{t \in \mathit{traces}(p) \mid |t| = \infty \vee t_{|t|-1} \in F\right\}$$

and the function $\mathit{p\_steps} : \mathcal{T} \to \mathbb{N} \cup \{\infty\}$ is defined by

$$\mathit{p\_steps}([s_0, s_1, \ldots]) = |\{i \in \mathbb{N} \mid s_i \to^p s_{i+1}\}|$$

Thus, the function *traces* obtains the set of all traces of a program, while the function *full_traces* obtains all terminated or infinite traces. The function *p_steps* counts the number of progress steps in a given trace.

A.2. Model Property Proof Sketches

A.2.1. Single-Execution. The single execution property claims that in a terminating trace for the entry point and each spawned task exactly one variant is selected and this variant is processed exactly once. The following two theorems formalizes this property:

Theorem A.1 (Single-Execution Tasks)**.** For any terminating trace $t = [s_0, s_1, \ldots, s_n] \in \mathcal{T}$ there are no two distinct states $s_i, s_j \in \mathcal{S}$ such that

$$\exists t \in \mathbb{T} : t \in q(s_i) \wedge t \notin q(s_{i+1}) \wedge t \in q(s_j) \wedge t \notin q(s_{j+1})$$

Thus, there are no two states followed by the start of the same task.

Since there is only one transition starting the execution of a task t, namely *(start)*, and its effect is the disappearance of t from Q in the corresponding state, we can focus on this effect in the theorem. If there are indeed no traces where the same task can be started twice, the first part of the property is covered.

Single-Execution Tasks. Proof by contradiction: let $t = [s_0, s_1, \ldots] \in \mathcal{T}$ be a trace with two distinct states $s_i, s_j \in \mathcal{S}$ and $t \in \mathbb{T}$ a task such that

$$t \in q(s_i) \wedge t \notin q(s_{i+1}) \wedge t \in q(s_j) \wedge t \notin q(s_{j+1})$$

W.l.o.g. we can assume $i < j$. Thus, there must be a $i < k \leq j$ such that $t \notin q(s_k)$ and $t \in q(s_{k+1})$. Since only spawn transitions add elements to Q, we conclude that $t \notin \mathbb{P}$ and

the transition $s_k \rightarrow s_{k+1}$ must be a spawn. However, since the execution of variants does not loop (the trace terminates) and each task including t has a unique spawn point which must have occurred for a state before state s_i, the task t can not be re-added to Q, position k can not exist, contradicting our initial assumption. □

Theorem A.2 (Single-Execution Variants)**.** For any terminating trace $t = [s_0, s_1, \ldots, s_n] \in \mathcal{T}$ there are no two distinct states $s_i, s_j \in \mathcal{S}$ such that

$$\exists v \in \mathbb{V} : v \notin v(s_i) \wedge v \in v(s_{i+1}) \wedge v \notin v(s_j) \wedge v \in v(s_{j+1})$$

Thus, there are no two states followed by the start of the processing of the same variant.

Single-Execution Variants. It follows from Theorem A.1, the fact that the *(start)* transition only picks a single variant, and the assumption that no two tasks have common variants. □

A.2.2. Termination. The *termination* property claims that if a deadlock free program has a terminating trace all of its traces not including infinite initialization, migration, and replication sequences will eventually terminate.

Before formalizing this property we have to provide a definition of a deadlock free program.

Definition A.4 (Deadlock Free)**.** A program $t \in \mathbb{P}$ is deadlock free iff

$$\forall s \in \mathcal{S} \setminus F : \textit{start}(t) \rightarrow^* s \Rightarrow \exists s' : s \rightarrow^{*p} s'$$

Thus, a program is deadlock free if and only if for each reachable non-terminal state s there is a sequence of state transitions leading to the progress of some task.

Furthermore, we have to impose one additional constraint on program variants not covered in the main body of the paper for brevity: w.l.o.g *no variant may be a no-op*. Thus, the processing of every variant contributes to the overall progress of the application. Since all variants of a task are computationally equivalent, the contribution of each variant is identical. Thus, progress contributed by a task is equal to the contribution of any of its variants.

In a next step we define the set of reachable tasks.

Definition A.5 (Reachable Tasks)**.** Let $p \in \mathbb{P}$ be a program. The set $T_p \subset \mathbb{T}$ defined by

$$T_p = \{t \in \mathbb{T} \mid \exists s \in \mathcal{S} : \textit{start}(p) \rightarrow^* s \wedge t \in q(s)\}$$

covers the set of all reachable tasks of p.

Lemma A.1 (Finite Reachable Tasks)**.** For a program p with a terminating trace t the set T_p is finite.

Proof. (– Sketch –)

- a terminating program p covers a finite amount of work
- reachable tasks form a hierarchy through their spawn points, rooted by the entry point
- for each task the covered work has to be greater than the sum of its child tasks due to the no no-op assumption; thus, since work is finite, the tree can not be infinitely deep
- furthermore, since all variants of each individual task are computationally equivalent and at least one of those has terminated, all tasks terminate; thus no task can spawn an infinite number of sub-tasks, and each node in the hierarchy can only have a finite number of child nodes
- it follows that the task hierarchy is finite, and thus the number of reachable tasks is finite too

□

Definition A.6 (Reachable Variants)**.** Let $p \in \mathbb{P}$ be a program. The set $V_p \subset \mathbb{V}$ defined by

$$V_p = \{v \in \mathbb{V} \mid \exists s \in \mathcal{S} : \textit{start}(p) \rightarrow^* s \wedge v \in v(s)\}$$

covers the set of all reachable variants of p.

Lemma A.2 (Finite Reachable Variants)**.** For a program p with a terminating trace t the set V_p is finite.

Proof. It follows from Lemma A.1 and the definition that each task has a finite number of variants. □

Finally, this *termination* property can be formalized as follows:

Theorem A.3 (Termination)**.** Let $p \in \mathbb{P}$ be a deadlock free program. If there is a terminating trace $t \in \textit{traces}(p)$ then

$$\forall t \in \textit{full\_traces}(p) : \textit{p\_steps}(t) \in \mathbb{N}$$

Proof. Let $p \in \mathbb{P}$ be a deadlock free program with a terminating trace t. Since p terminates, V_p is finite. Furthermore, since all variants must terminate due to the computational equality assumption, every variant can be completed with a finite number of $\rightarrow^p$ transitions involving the progressing of the respective variant. Let $u \in \mathbb{N}$ be the upper boundary for the number of progress transitions required by any variant in V_p to terminate. Then $u_b = u|V_p|$ provides an upper boundary for the number of progress steps $\textit{p\_steps}(t')$ to be performed by any full trace t' of p. □

A.2.3. Satisfied Requirements. The *satisfied requirements* property claims that variants are only processed on compute units where all required data is available for the duration of their processing.

Informally, this can be proven through the following steps:

- let $[s_0, s_1, \ldots] \in \mathcal{T}$ be a trace and $v \in \mathcal{V}$ be a processed variant
- when starting v a set of fresh read and write locks l_r and l_w are set
- write locks can only be removed by *(end)* transitions terminating v, or *(destroy)* transitions; in either cases v will no longer require access to effected data
- the *(start)* transition initiating v also ensures the locked data to be present on the node processing v; thus, initially all data is present
- only *(migrate)* and *(destroy)* transitions can remove data from a memory space; *(destroy)* transitions have

already been covered, and *(migrate)* transitions can not be applied as long as the variant v holds its locks;
- thus, all data required by v is retained at its position for the duration of the execution of v

A.2.4. Exclusive Writes. The *exclusive writes* property claims a data element being write locked in some memory address space is not replicated anywhere else in the system at the same time, nor can such replicates be created

Informally, this can be proven through the following steps:

- write locks are introduced through *(start)* transitions, which ensure that before and after the transition only a single copy of write-protected data is maintained throughout the system
- only *(migrate)* and *(replicate)* transactions may move data to different address spaces or create a copy; neither of those is enabled as long as write locks are held;
- based on those two observations, the property can be proven by induction on the length of traces after the creation of a write lock

A.2.5. Data Preservation. The *data preservation* property claims that the runtime system cannot delete data that is not explicitly destroyed. The runtime can, however, remove replicated data.

Informally, this can be proven through the following steps:

- data may only be lost through *(destroy)* or *(migrate)* transitions
- *(destroy)* transitions are explicitly triggered by the application; the related loss of information is thus accepted;
- *(migrate)* transitions move existing data between address spaces; every element removed from the source address space is added to the target address space; thus no information is lost

While no information can be lost, the runtime system is indeed able to remove replicated data. For instance, let s be a state such that $\{(m_1, d, e), (m_2, d, e)\} \in d(s)$ for two distinct address spaces $m_1, m_2 \in M$ and some data item $d \in \mathbb{D}$ and element $e \in \mathbb{E}$. Thus, the element e of d is replicated among two distinct address spaces. Furthermore, let

$$\nexists v \in \mathbb{V} : (v, m_1, d, e) \in l(s) \vee (v, m_2, d, e) \in l(s)$$

Thus neither copy is lock protected. To eliminate, for instance, the copy in m_1, the runtime system is free to perform a *(migrate)* transition using m_1 as the source address space m_s, m_2 as the destination address space m_d, d as the data item and the range $E = \{e\}$. The effect will be a transition to a state s' without the copy of the element e of data item d in address space m_1. The copy in m_2 persists.