H2020 FETHPC-1-2014

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

Project Number 671603

D4.1 – AllScale runtime system interface specification

WP4: Unified runtime system for extreme scales

Version: 1.0
Author(s): Thomas Heller (FAU), Arne Hendricks (FAU), Hebert Jordan (UIBK), Peter Thoman (UIBK)
Date: 28/03/16

Due date:	PM6
Submission date:	day/month/year
Project start date:	01/10/2015
Project duration:	36 months
Deliverable lead organization	FAU
Version:	1.00
Status	Final
Author(s):	Thomas Heller (FAU) Arne Hendricks (FAU) Herbert Jordan (UIBK) Peter Thoman (UIBK)
Reviewer(s)	Kiril Dichev (QUB), Thomas Fahringer (UIBK)

Dissemination level	
PU	*Public*

Acknowledgements

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

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

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

More information

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

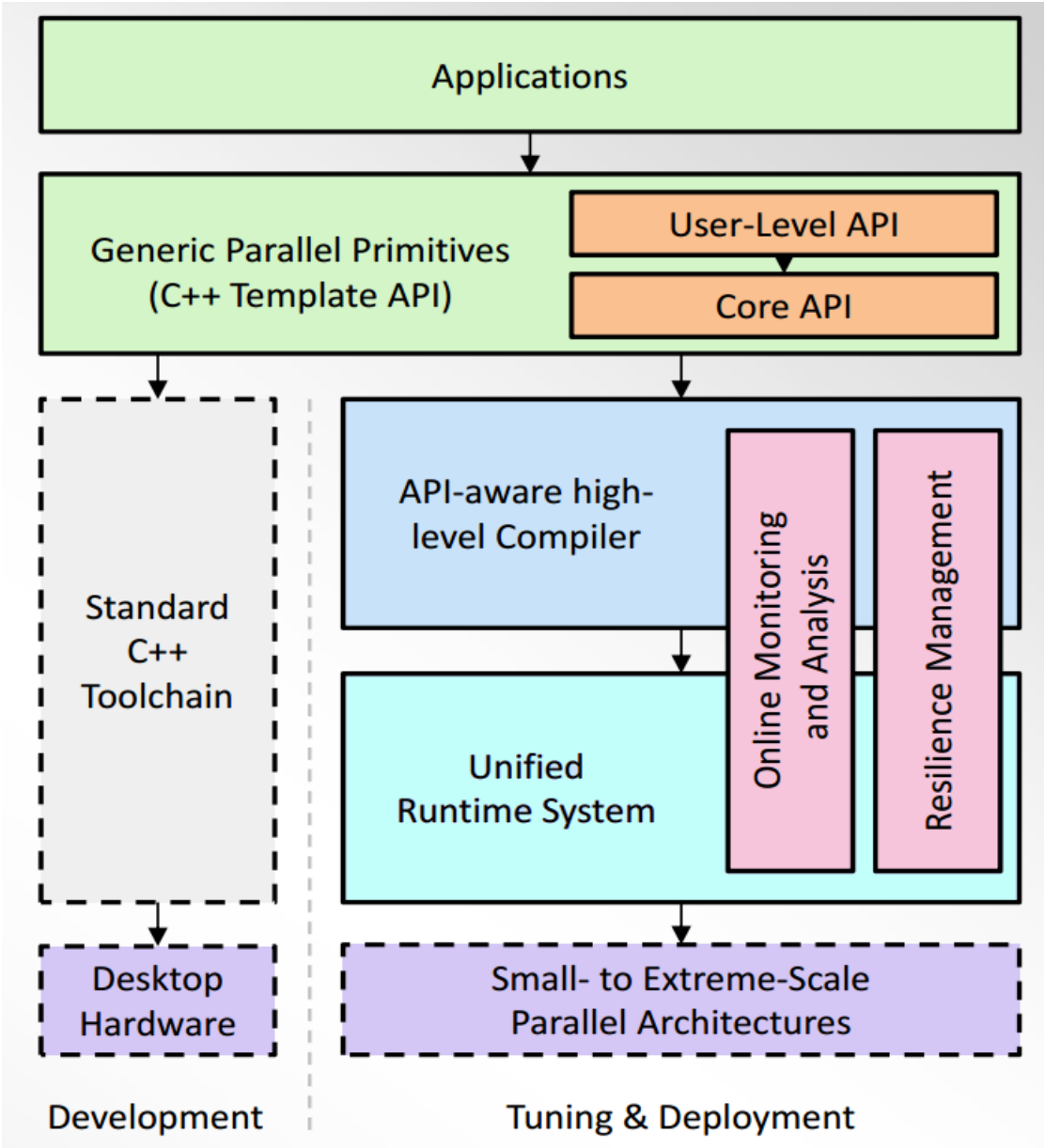
Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	21/02/16	First draft	Thomas Heller
0.2	03/03/16	Second draft	Arne Hendricks
0.3	07/03/16	Third draft	Arne Hendricks
0.4	16/03/16	Final draft	Arne Hendricks
1.0	19/03/16	Final version	Thomas Heller

Table of Contents

1 Introduction

The runtime system interface is the layer describing the structure of an application that can be managed and tuned by the AllScale runtime system. The fundamental functionality and properties of this essential interface will be determined by Task T2.1 and incorporated in the overall system architecture by Task T2.2. Simultaneously, to aid the interface specification within task T2.2, we formalize the involved entities through actual specifications thereof using C++ declarations.
The runtime interface is connected to the tools and entities developed by the other work packages, such as work package 5, and can be seen in the lower diagram, highlighted in cyan.



The main interacting areas are with the compiler, the online monitoring tool and the resilience manager. Due to the high level of coupling to other components it

is very important for the resulting runtime interface specification to find a model of expressing parallel (recursive) tasks, data dependencies, IO characteristics, and hardware requirements as well as means for offering multiple, exchangeable implementations constituting different variations of the same code fragment and meta-data capturing resilience properties of the provided codes, in particular failure compensation strategies.

2 Runtime Interface

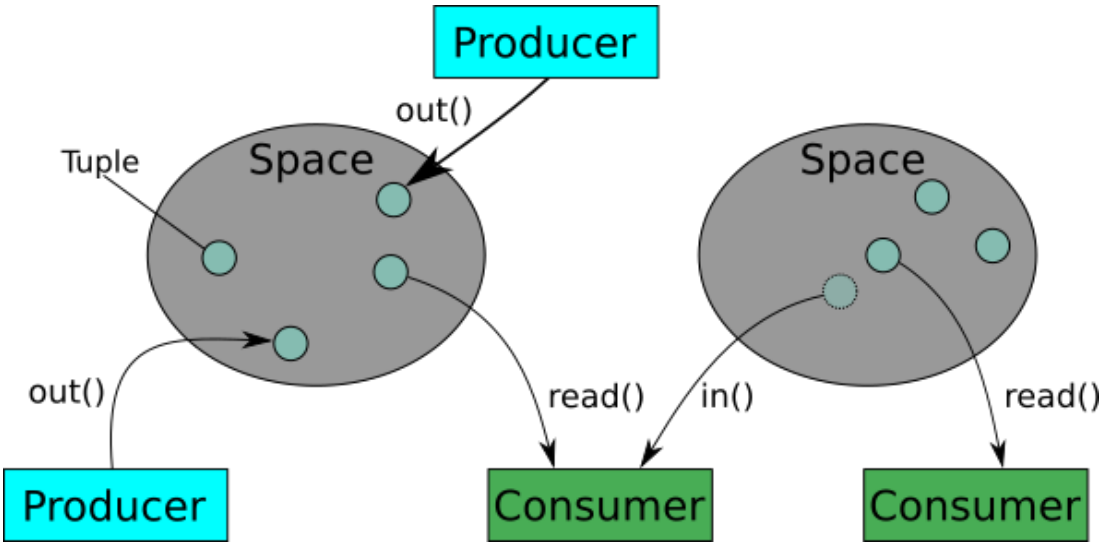
The runtime interface is based on a theoretical foundation with similarities to the tuple-space paradigm. This paradigm defines a collection of tuples to be accessed and used concurrently. It was introduced by David Gelernter at University of Yale in the 1980s by the Linda coordination-language. Applications using Linda and its tuple-spaces realize communication among processes by defining a collection of tuples (Key-Value pairs) and information is exchanged by writing/loading (consuming/producing) tuples. It was an early approach to a distributed shared memory, and the key concepts beneath it, i.e. modeling an application's communication in terms of producing and consuming shared resources, is similar to modern approaches such as the AllScale runtime interface.

2.1 Tuple-Spaces

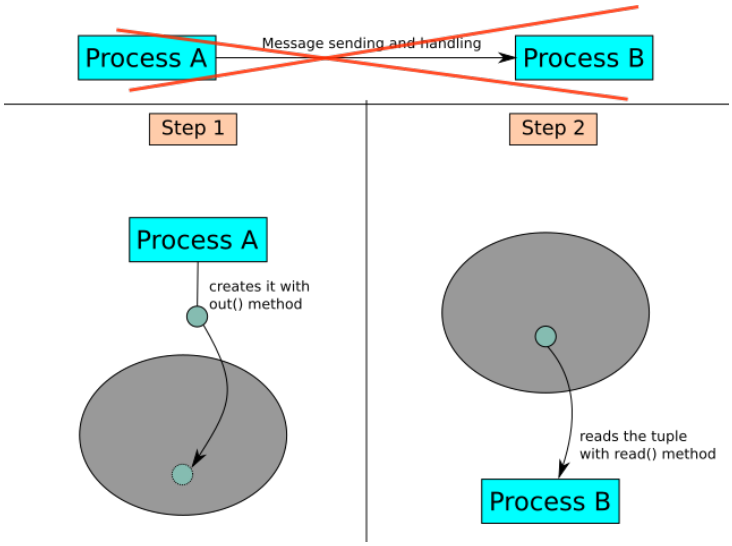
A tuple-space holds a number of Key-Value pairs, which can be concurrently manipulated. The communication is handled by applying read and store operations on these tuples, commonly referred to as *consume* and *produce* operations. The key concept behind it, i.e. dividing an application's tasks by means of producing and consuming shared resources, is very similar to modern approaches such as the AllScale runtime-interface. A tuple-space environment offers at least three operations:

- A method to insert a tuple into the space, originally referred to as the out() method
- A method to withdraw tuples from the tuple-space, the in() method
- A method to read a tuple from the tuple-space without removing it, the read() method.

The following diagrams describe the interaction of 4 consumers with 2 distinct tuple-spaces, inserting, reading and deleting tuples:



Instead of passing messages between two processes A and B, it is now possible for them to communicate by accessing the tuple spaces: process A creates one or more tuples encoding the message to be conveyed, adds them to the tuple-space from where process B may consume them by withdrawing or reading them. The behavior is described in the figure below:



A number of interesting properties originate from the tuple space paradigm, rooted in its principle of communication orthogonality (neither sender nor receiver have prior knowledge about each other): an uncoupling of communication both in regards of time and space. Space uncoupling refers to the concept of distribution of resources, allowing 1:N communication as well as 1:1 between processes. Time uncoupling, on the other hand, means that a producing process can run to completion and terminate, while its produced tuples remain in the tuple-space once they are added, available for other processes to be consumed. This concept has remained very important over the years, and is also

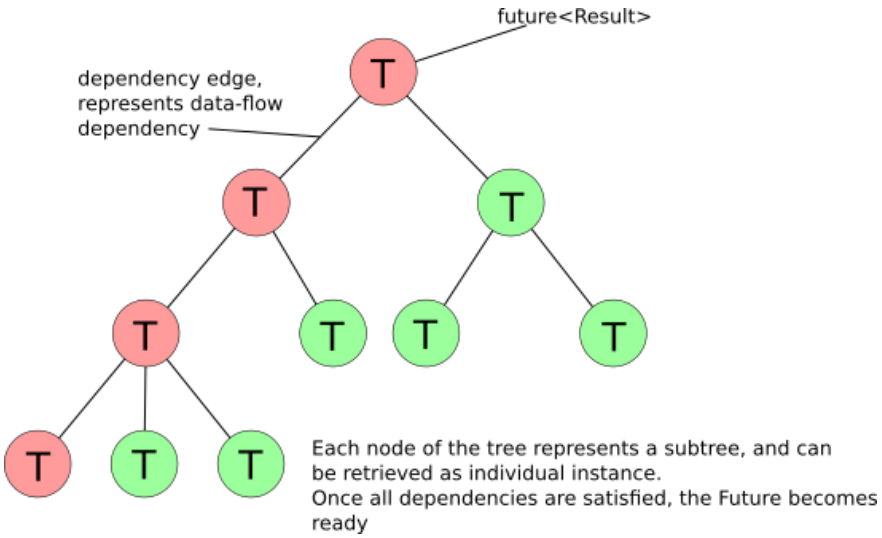
applied by HPX with its reference counting within the Active Global Address Space (AGAS).

The producers and consumers can be spread across a distributed address space - they are still able to access the tuples, and a tuple is usually unique in the tuple-space. It should be clear from the diagram that the model leads to a form of shared, distributed memory space, because tuples can actually reside on different physical nodes but may still be accessed by the other members

The details of these concepts can be found in: Linda in Context (1989).
The kind reader might also be interested in the approach of the Intel Concurrent Collections, a template library for C++ parallel and distributed applications, introducing a data and control flow model distantly similar to ours and influenced by the concepts of tuple-spaces.

2.2 AllScale Runtime Interface

We will now continue with an introduction and explanation of the fundamental concepts and entities of the AllScale runtime interface. Our theoretical foundation of managing concurrently used resources is influenced by the tuple-space paradigm. Our concept will add to this general idea what we call decomposable tuples. Decomposable tuples are represented by a hierarchical tree of futures. Each task that is spawned within the scheduler will return such a future. Each future represents the completion and return value of a recursive task.
The figure below shows such a dependency tree of futures:



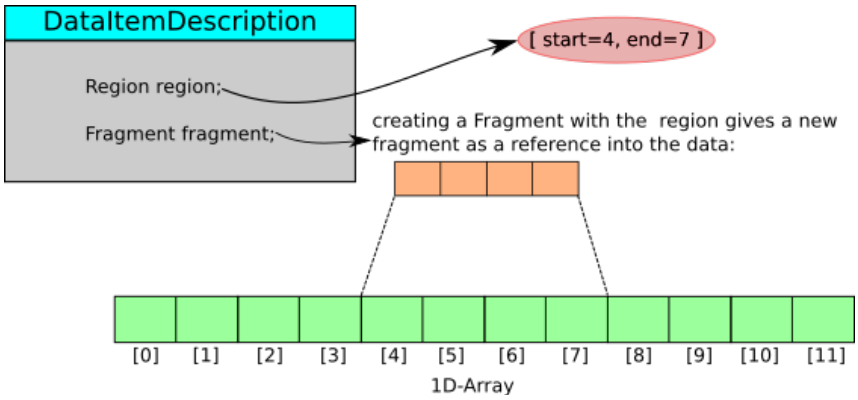
The final result might depend upon the completion of a tree of other results, which is, for instance, the case when recursively calculating Fibonacci numbers. By relying on HPX as the underlying runtime API, we can construct these trees of futures using Data-Flow control techniques and additionally execute applications without thread-suspension or waiting. This will allow the runtime to be as resource efficient as possible and not waste any memory in the stack segment by having a great number of already allocated but suspended threads. The later

feature will be of great use when dealing with extreme scale concurrency leading to billions of concurrent tasks such as targeted by the AllScale environment.

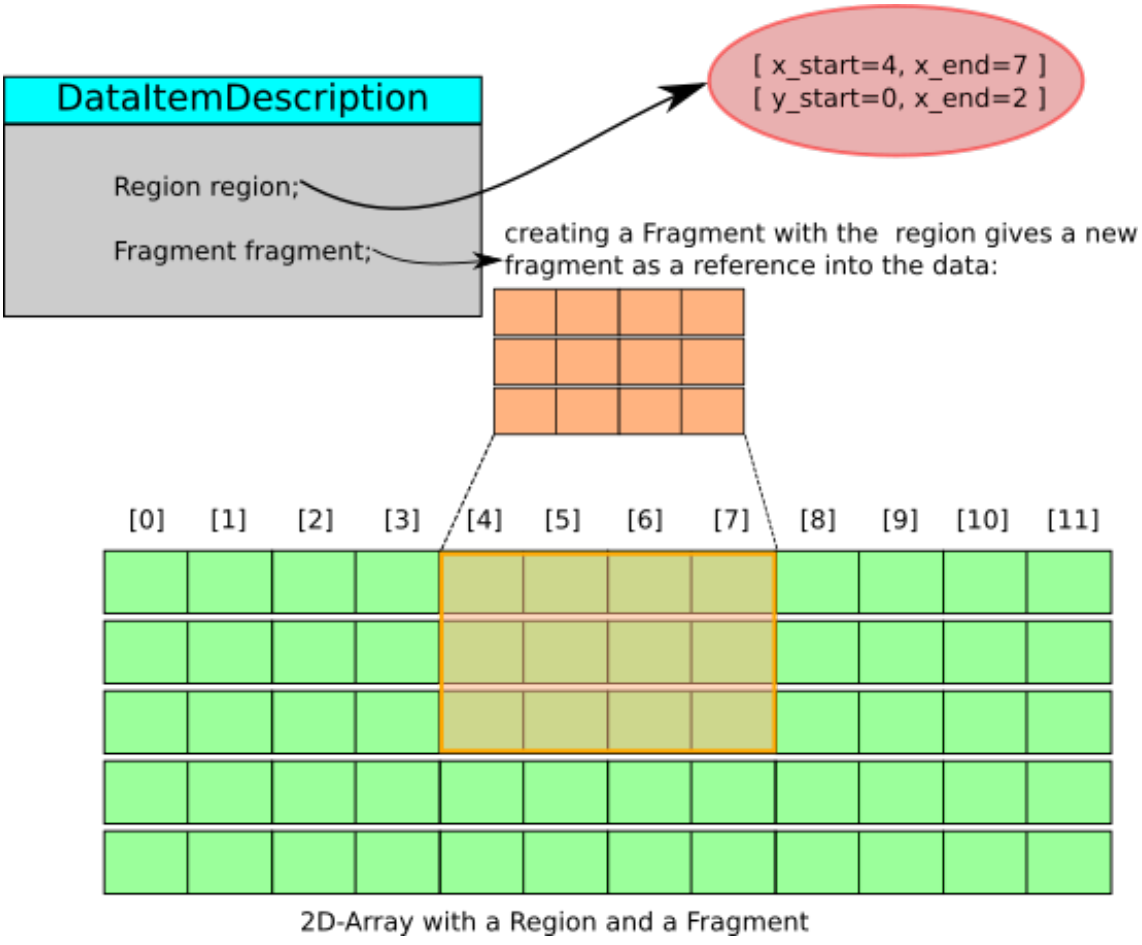
Our main foundational entities are called **WorkItem** and **DataItem**. Instances of those two can be modeled as tuples in a tuple-space. Similar to the tuples in the diagram above, our entities need to be accessed concurrently and might reside physically spread on a cluster of nodes.
The runtime interface offers methods equivalent to put() and get() in a tuple-space , the write() or store() as well as the load() method, which will be described in more detail later.
Differing to the tuple-space pictured above, in our case it is also possible to retrieve a Tree of Futures to tuples.
A tree of futures is called "*Treeture*" and it represents the dependencies among nested tasks to be computed before being able to obtain the value promised by the task associated to the root node's future.
The usage of *Treetures* combined with operations on the future tree, such as union and intersect, enables us to have set-like behavior enhancing the tuple-space's abilities by adding the possibility of retrieving a (partial) tuple, which corresponds to a (sub-) tree of futures.

2.3 Key Concepts explained

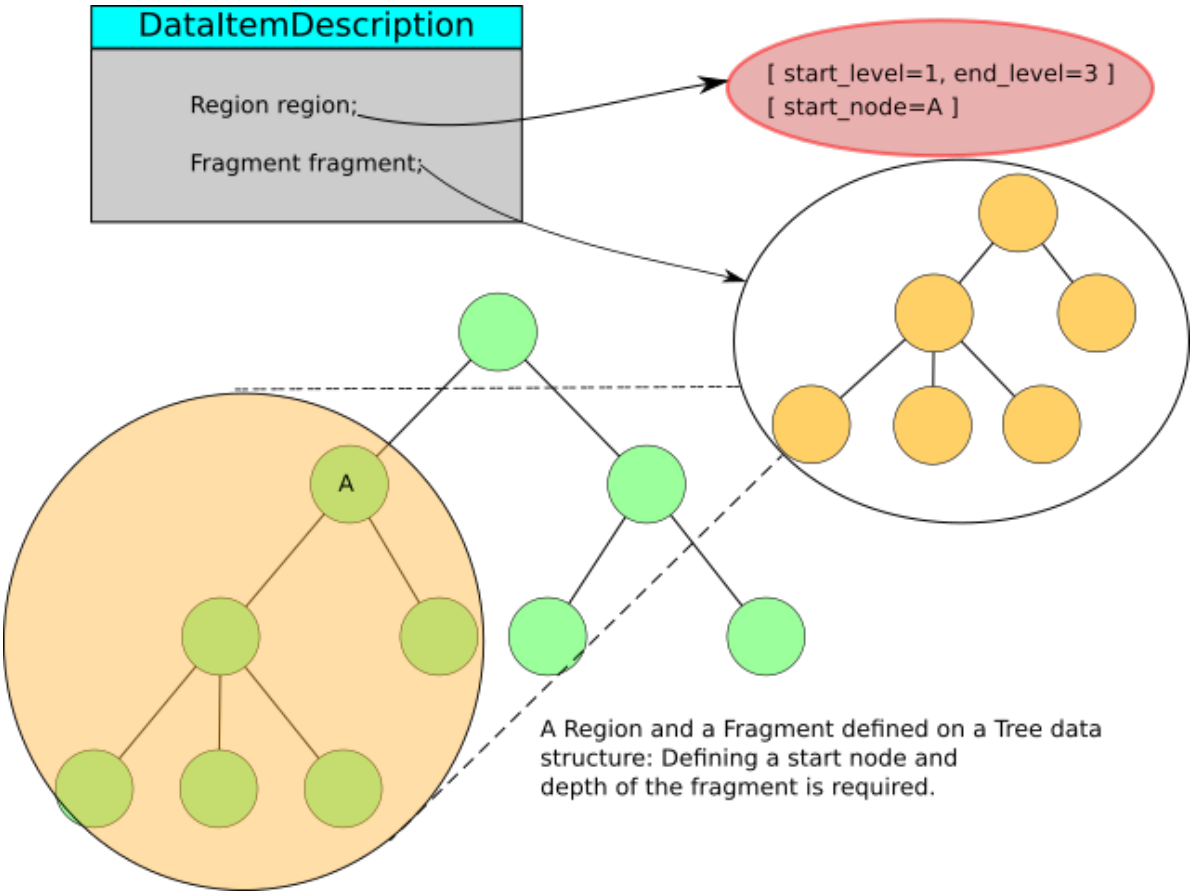
We will now introduce some of the main concepts of the runtime interface in an informal way, without actual source code. Further down in the document we will show some of the details of the planned implementation of the runtime interface to provide a more detailed picture of the runtime to the reader.
To explain the concepts, we will run the reader through some simple examples, from top to bottom. An application, e.g. a simple stencil, is run in parallel on 4 nodes of a cluster. The data is stored in a 1D array of values.
Initially the recursive split-up of tasks is done by the AllScale compiler; it provides implementations of tasks in the form of **WorkItems**. These **WorkItems** are our entities that describe what has to be done (in our example performing computations on the stencil cells) and on which data. The data can be spread over the network. Within the stencil example, there will be the need to access data which resided physically on another node in the cluster.
Regarding the management of data, there are three concepts that need further explanation: **Regions**, **Fragments**, and **CollectionFacades**.
A **Region** exposes information to address subsets of elements within a collection. In an array like collection, regions might be realized by describing sets of indices or, more compact, as an interval of indices. For tree shaped data structures, regions may be addressed by addressing root nodes of subtrees to be addressed.
The actual data storage is realized by **Fragments**, which corresponds to the object instantiated on the individual nodes. Each fragment contains a share of the overall collection and data may be moved between fragment instances. The data within fragments is thereby addressed by the associated region structure. The following figure visualizes the relation between regions and fragments for our 1D array example:



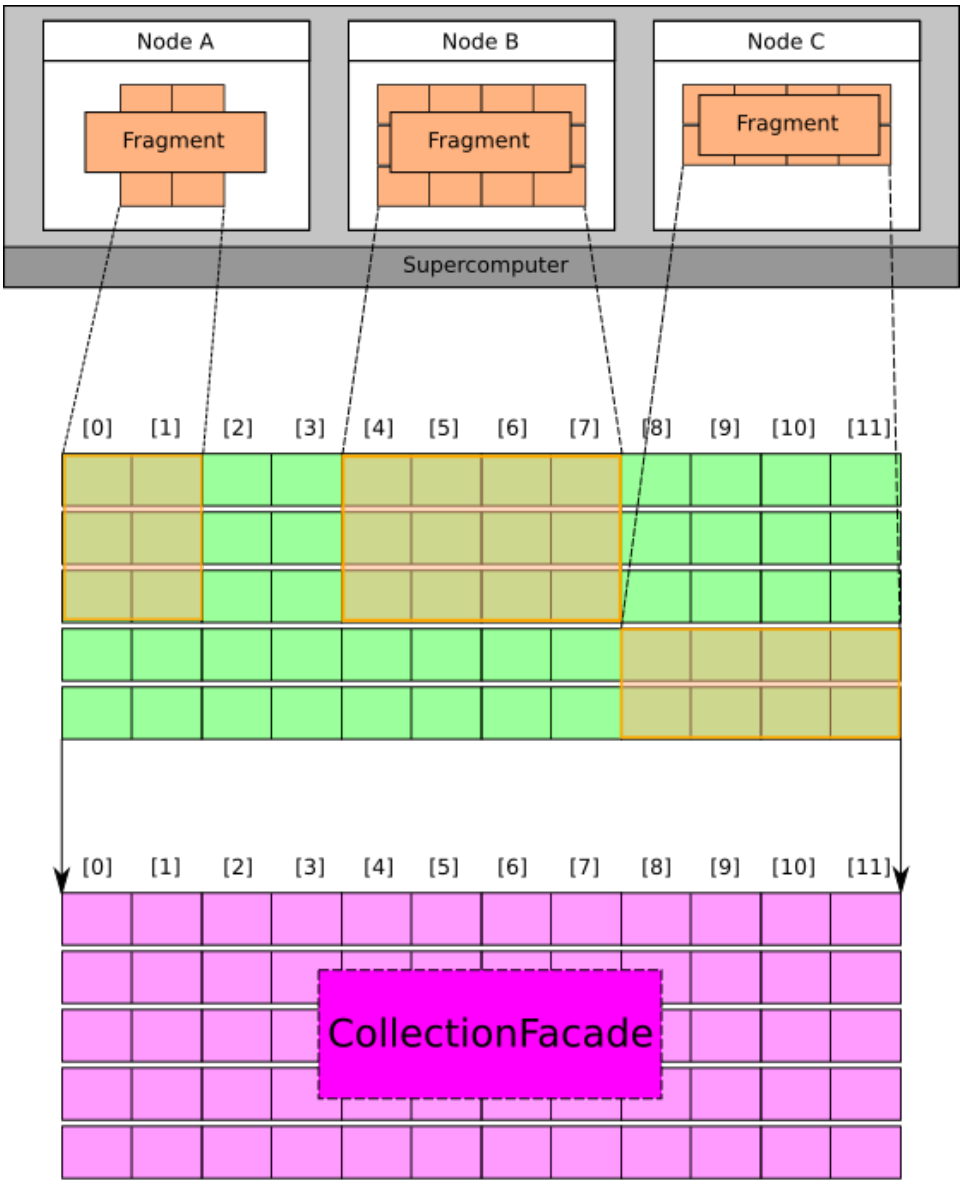
A **Region** is defined, covering indices 4 to 7, and can then be used to create a fragment, including the data addressable by those indices. Depending on the underlying data structure, regions need to include different parameters: in the simple 1D example these are just start and end indices (column indices), but for a 2D array information about the row is needed as well:



As mentioned above, the **Region** concept is versatile, and can also be used to access structures like a tree:

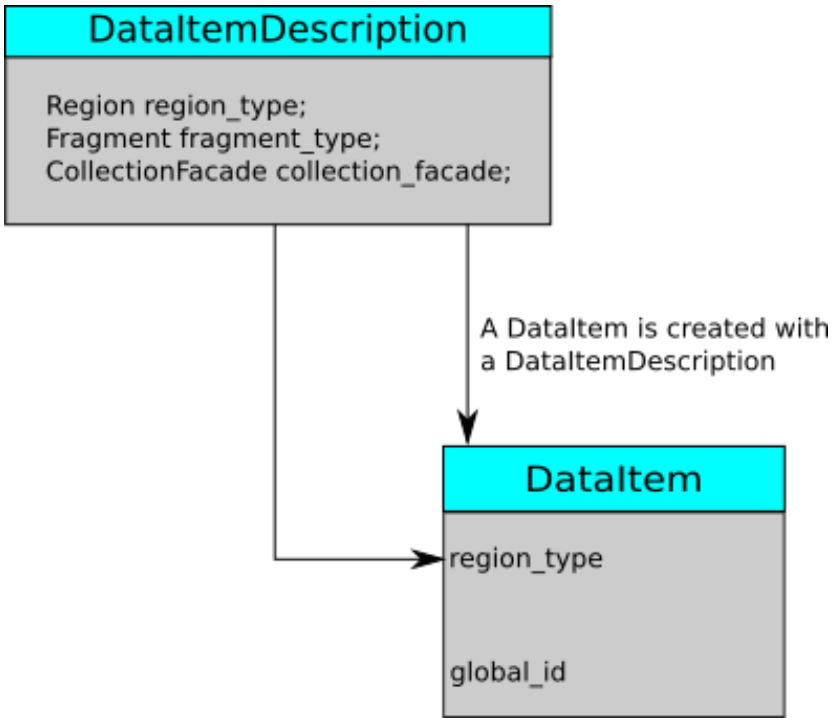


In order to provide a unified API to the user, hiding the internal data management, we introduce **CollectionFacades,** which are a user-facing wrappers to create and manipulate distributed data collections.
A **CollectionFacade** appears like it is accessing globally shared data, while in fact it is accessing localized **Fragments**, managed by the runtime system. This makes managing the data possible in a distributed environment. The interaction of Fragments and **CollectionFacades** in a distributed environment is illustrated in the figure below.



2D-Array with distributed Fragments on a Supercomputer.
The purple colored CollectionFacade is what the user faces, and can as seen in the figure, be physically distributed over different nodes of a computer.

As we covered the **Region**, **Fragment**, and **CollectionFacade** concepts, we will now introduce the **DataItem**.
In order to create a **DataItem**, a **DataItemDescription** is created, which basically exposed type information associated to a **DataItem**. Thus it summarizes what types are used to process the data, including the types for Regions, Fragments, and CollectionFacades.

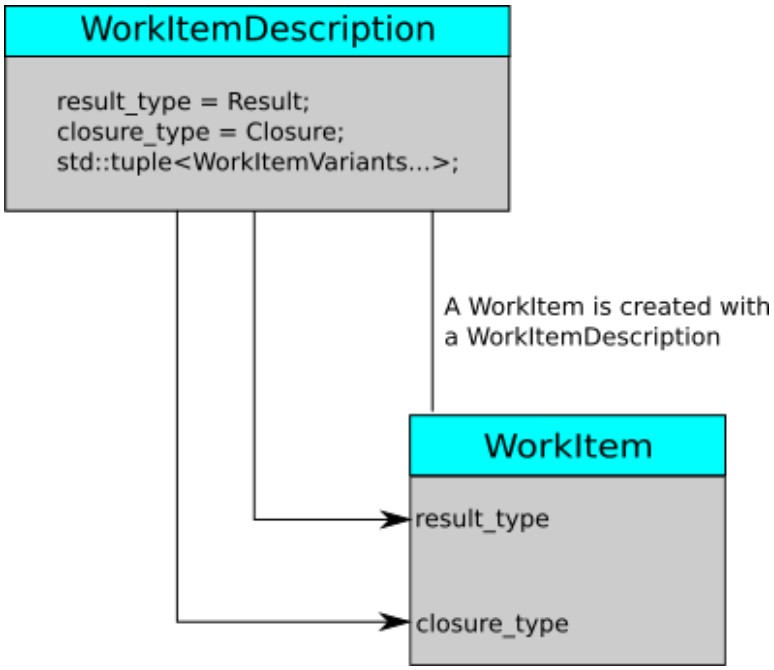


2.3.1 DataItem

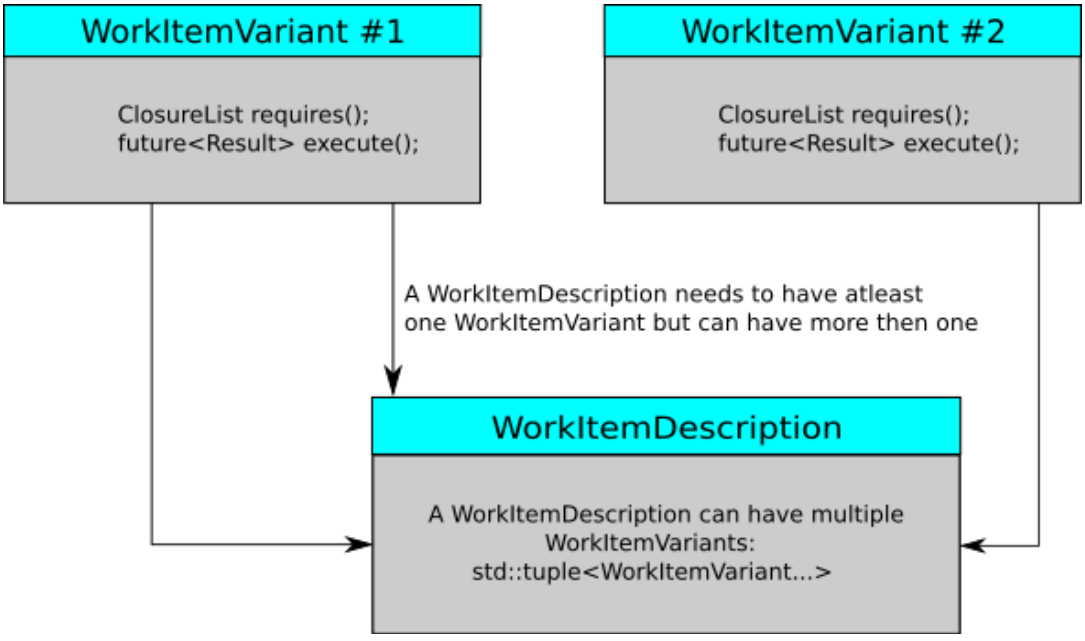
The **DataItem** itself is a symbolic representation to some managed data element. Typical user-level code might access elements in an array just by using an array, because this approach will fail to suffice when working in a distributed system, the data needs to be somehow wrapped by a management-layer. This is what **DataItems** do.
Each **DataItem** has a unique symbolic name: a global identifier. It also uses a **Region** instance to describe the full size of the (virtual) data item. It is the runtimes job to take the full size of the data item, partition it into smaller sub-regions and create fragments covering those sub-regions throughout the available nodes. This data distribution may later be adjusted dynamically, to facilitate load balancing/management operations or to react on changes in the infrastructure, like joining or failing nodes.

2.3.2 WorkItem

Having explained **DataItem**, it is time to illustrate the **WorkItem** concept.
A **WorkItem** is the runtime interface to describe a task, with additional data dependencies and various code variants.



It is constructed by utilizing a template, called **WorkItemDescription**. This template contains information about the result we expect from the **WorkItem**, as well as the type of data / parameters the task needs to be processed, called "*Closure*" type. It can also contain multiple implementations of the actual task. This is covered by *tuple<WorkItemVariants...>* field in the description.
We need to define the term *WorkItemVariant* at least roughly, because it is essential for understanding how **WorkItems** are constructed: A *WorkItemVariant* is a template class that is used to describe one implementation of a work task.



It consists, among other things, of a *requires* function which returns a list of data dependencies imposed by the execution of the corresponding code variant. Those requirements list sub-regions of **DataItems** and access modes (e.g. ReadOnly or ReadWrite) which will have to be accessible on any node supposed to processes the associated task. The *WorkItemVariant* also contains the *execute* method, which is the actual implementation of the computation. This is where the user code will be located. The return value of the *execute* method is what we call a "*Treeture*".
At least one *WorkItemVariant* is needed to generate a **WorkItemDescription**, which in turn is needed to generate an actual **WorkItem**. However, there can be multiple variants, e.g. when the compiler is synthesizing specialized code versions for specific use case (sequential execution, checkpoint creation) or target architectures (various accelerators).
There are also two entities to administrate these items: the scheduler and a manager for **DataItems** which will be presented later.
Having discussed the basic elements in an informal and abstract, way we will now continue to describe the same components in regards of their actual representation in the interface, i.e. written code.

2.3.3 Region - Realization

As explained, a region instance addresses a subset of elements of a collection. Regions are needed to reference sets of elements in data objects. They have to support a set of operations. One very important being the load operation, which loads a `Region` from an archive obtained from the network. Other operations include union and intersection of Regions of the same type, need for the management of data partitions.

```
/// Description on how the data is accessed
enum access_privilige
{
    read_only,
    write_first,
    read_write,
    write_only,
    accelerator_read_access,
    accelerator_write_access,
    accelerator_read_write_access,
};

template <typename Region>
struct is_region;

/// Calculates the union of two regions
Region merge(Region const & region1, Region const& region2);

/// Calculates the intersection of two regions
Region intersection(Region const & region1, Region const& region2);

/// \return true if the region is empty, false otherwise
bool empty(Regions const& region);

/// Loads a Region from an archive which has been obtained from the network
template <typename Archive, typename Region>
void load(Archive & archive, Region & region);

/// Saves a Region to an archive which is supposed to be sent over the
```

```
    /// network
    template <typename Archive, typename Region>
    void save(Archive & archive, Region & region);
```

2.3.4 Fragment - Realization

A `Fragment` is a storage for a sub-set of elements of a collection, which is addressed by a Region:

```
    /// A Fragment is a reference into a collection of data with a specific
    /// viewpoint expressed with a Region
    Fragment create(Region const& region);
    void resize(Fragment const& fragment, Region const& region);
    OutFragment mask(Fragment const& fragment, Region const& region);
    void insert(Fragment & destination, Fragment const& source, Region
    const& region);
    void save(Archive& ar, Fragment const& fragment);
    void load(Archive& ar, Fragment & fragment);
```

2.3.5 DataItem - Realization

The static type part of a `data_item` can now be described using a template parameter `DataItemDescription`, which is used to create it.
A `DataItemDescription` contains information about the types used to process the data item:

```
   /// Concept CollectionFacade:
   /// A CollectionFacade is the user facing type which is logically
representing
    /// a global view on the data, however, it might only have access to a
acquired
    /// fragment which is just a subset
    /// Information about the types used to process a data item
    template <typename Region, typename Fragment, typename CollectionFacade>
    class data_item_description
    {
        using region_type = Region;
        using fragment_type = Fragment;
        using collection_facade = CollectionFacade;
    };
```

Where `CollectionFacade` is the interface to the end-user, a global view to the distributed data collection.
The `data_item` itself contains the full size of the data item in the form of a region_type value, and a unique identifier to identify the `data_item` globally:

```
    /// A symbolic representation to some managed data element (collection)
    template <typename DataItemDescription>
    class data_item
    {
        using region_type = typename DataItemDescription::region_type;
        /// Creates a data item with a unique identifier
        data_item();
        /// Returns the unique identifier to a data item
        id_type get_id() const;
    };
```

Creation, acquisition and deletion of `data_items` is the responsibility of the `data_item_manager`, which offers a create, destroy, acquire, and release method and is the interface to the compiler:

```
class data_item_manager
    {
        /// creates a symbolic instance for a data item
        template <typename Executor, typename DataItemDescription>
        data_item<DataItemDescription>
        create(typename DataItemDescription::region_type const& size, Executor
= this_executor);

        /// destroy a symbolic instance for a data item
        template <typename DataItemDescription>
        void destroy(data_item<DataItemDescription> item);

        /// Called in the generated code (from the compiler)
        /// when the data is accessed
        template <typename DataItemDescription>
        typename DataItemDescription::collection_type acquire(
            requirement<DataItemDescription> const& requirement);

        /// Called in the generated code (from the compiler)
        /// when the data is not needed anymore
        template <typename DataItemDescription>
        void release(
            requirement<DataItemDescription> const& requirement);
        /// \returns the locations of where the fragments to data items
        /// that the passed regions contains are located
        template <typename DataItemDescription>
        future<std::vector<
            std::pair<typename DataItemDescription::region_type, locality>
        > >
        locate(
            requirement<DataItemDescription> const& requirement);

        /// \param shopping_list a list of requirements with a locality hint
        ///   indicating on what should be gathered
        ///
        /// \return This function returns a future that becomes ready when the
        /// data has been succesfully copied to the system
        template <typename Executor, typename DataItemDescription>
        future<void>
        gather(
            Executor const& executor,
            std::vector<
                std::pair<requirement<DataItemDescription>, locality>
            > const & shopping_list);
    };
```

`data_items` are interfaced with the `work_items` by the `WorkItemVariant`, a generic concept that features a `requires(Closure const& closure)` method, which returns a list of **requirements**.
A **requirement** references a `data_item`, the sub-region of the item to be accessed, and access privileges needed for the associated task to be processed.

```
    template <typename Result>
    class WorkItemVariant
    {
        /// \return If the work item variant was generated;
        constexpr bool valid;
```

```
        /// \tparam Closure List of captured variables that is the used
        ///                  data items and passed parameters
        ///  \return A list of requirements
        template <typename Closure>
        static unspecified requires(Closure const& closure);

        /// Executes the given variant using the captured variables in the
closure
        /// once all requirements have been fullfilled
        template <typename Closure>
        static treeture<Result> execute(Closure const& closure);

        template <typename Closure, typename NonFunctionalProperty>
        static typename NonFunctionalProperty::result_type
        non_functional_property(Closure const& closure);
    };
```

2.3.5 WorkItem - Realization

The `WorkItemVariant` exposes an execute method which is passed a `Closure` as parameter (representing the captured variables in the `Closure` once all requirements have been acquired by the use of the `data_item_manager`), and then executes the given variant.
`WorkItemVariant` also offers the possibility to specify additional non-functional-properties characterizing the represented code variant. These can include things such as parallel granularity, resiliency requirements, and memory usage patterns as well as whether or not GPUs or other accelerators are utilized for the computation. This information is made available to the scheduler for conducting scheduling decisions.
`WorkItemVariant`s are used to create `WorkItemDescriptions`:

```
template <typename Result, typename Closure, typename SplitVariant, typename
ProcessVariant,
        typename ...WorkItemVariant>
    class work_item_description
    {
        using result_type = Result;
        using closure_type = Closure;

        using split_variant = SplitVariant;
        using process_variant = ProcessVariant;
        using work_items = std::tuple<WorkItemVariant...>;
    };
```

Each `WorkItemDescription` can contain a number of `WorkItemVariants`, represented by a `std::tuple<WorkItemVariant...>`.
In order to create a `work_item` object, a `WorkItemDescription` is passed as template parameter to the `work_item` class, resulting in a `work_item` featuring the given `WorkItemDescription`.

```
class work_item_base;

    template <typename WorkItemDescription>
    class work_item : work_item_base
    {
        using result_type = typename WorkItemDescription::result_type
        using closure_type = typename WorkItemDescription::closure_type;
```

```
        work_item(closure_type && closure);
    };
```

Another method needed for the scheduling is the run method:

```
    /// Execute a given variant with the closure, this gets called from within
    /// the scheduler
    template <typename Executor, typename WorkItemVariant, typename Closure>
    void run(Executor, Closure && closure);
```

It is passed a `WorkItemVariant` and an associated closure in order to execute the given variant on a hardware resource determined by first parameter being an HPX executor instance.

2.3.6 Scheduler - Realization

Having introduced the basic classes needed for dependency, data and work descriptions, we will now introduce the scheduler interface.
The scheduler is a generic component in the global address space, which is responsible to make decisions regarding where and how to process work items, where to place sub-regions of data items, and how to tune hardware parameters.
It features a `spawn` method, which is responsible for scheduling the execution of a given WorkItem, by first selecting one of its code variants and a location for its execution, followed by forwarding the selected options to the run method outlined above.

```
    struct scheduler
    {
        /// Flushes the queues and aborts all tasks that haven't been started
yet
        void flush();

        /// This function will be generated in the compiler and eventually
        /// spawns a specific WorkItemVariant based on some scheduler decision
        template <typename WorkItemDescription>
        treeture<typename WorkItemDescription::result_type>
        spawn(
            typename WorkItemDescription::closure_type const& closure);

    private:
    };
```

The `spawn` method is invoked by the code generated by the compiler, with a generic parameter `WorkItemDescription`, describing the `work_item` it will be creating. As visible from the signature above, `spawn` will be given a parameter `WorkItemDescription::closure_type const& closure`. It is used to capture the functions “calling context”, including captured parameters and data items.
The `spawn` method returns a Treeture providing a future handle to the results of the work item.
Type information about the type of the result in the Future-Tree are provided by the `WorkItemDescription::result_type` template parameter of the `treeture` return type of the `spawn` method.

2.4 Hardware

The configuration of the hardware is accessible through topology information, that is, local topology of the underlying processor/node with respect to processing units, memory and interconnects.

```
/// Returns a topology of executors that cover:
   ///   - NUMA Domains
   ///   - Symmetric multiprocessing
   ///   - Accelerators
   ///   - Provides different sets of cores to use
   ///   - Different low level scheduling policies:
   ///      * FIFO ordering
   ///      * Work stealing within the set of cores
   ///      * ...
   executor_topology get_topology();
```

Since the scheduler might need to be able to make decisions involving more than one process/locality it should make use of the HPX communication facilities by registering schedulers with a unique name and resolve the names to Global IDs in order to make call scheduler defined actions to make further decisions.

3 Monitoring/Events

The runtime system exposes an API based on HPX performance counters which allows the following:

- Generate AllScale Specific Performance Counter Data
- Globally discoverable and queryable
- Subscribable: event on change or event on change given a specific threshold etc.

This will form the basis for the Perfomance Monitor and Resilience Manager components as they will be able to a) register counters and b) react on the counters to either generate new work items or generate events for the scheduler to react on.

4 Interface to the Performance and Resilience Monitor

Work Package 5 is the task of developing language and tool support for continuous monitoring of application performance and error resilience, as well as support for application-specific, algorithmic error detection and recovery from both errors and performance anomalies due to non-deterministic variability in performance. In order to accomplish this task the runtime needs to be interfaced to the cross-layer resilience monitor.

The interface to the tool developed in WP5 will be based on HPX performance counters. It can thus provide a range of information, including locality-specific performance metrics such as thread queue length, states of threads and work items, and cache statistics. This information can then be used by the performance monitor to inform the dynamic optimizer to make prudent decisions at runtime.

5 Interface to the Resilience Manager

In order to provide resilience support, the runtime system will provide functionality to backup/restore data items and work item states. In the event of a network or node failure, or any other disruption, the resilience manager will orchestrate a recovery process. It has to be discussed which component is detecting network faults. Furthermore, global checkpointing needs to be avoided due to its detrimental effect on performance.

6 Conclusions and Future Work

Our specification allows us to express tasks, data dependencies, and non-functional requirements such as hardware requirements in a parallel-ready way. The usage of template-heavy code assures maximum flexibility for further development as well as performance benefits. Paradigms such as the work and data items have the advantage of offering ways to utilize multiple implementations of variations of the same code fragments. The performance Monitor and resilience manager interfaces are still very early in their planning phase and needs to be further refined based on the insights obtained over the course of the development process.

Bibliography

D. Gelernter - Linda in Context (1989).