INSTITUT FUER INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITAET MUENCHEN

Bachelor Thesis

Python-Bindings for the DASH C++ Template Library

Josef Schaeffer

INSTITUT FUER INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITAET MUENCHEN

Bachelor Thesis

Python-Bindings for the DASH C++ Template Library

Josef Schaeffer

Supervisor: Prof. Dr. Dieter Kranzlmueller
Advisor: Tobias Fuchs

Submission Date: June the 7th, 2017

Declaration of originality

I hereby declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such.

Munich, on the 7$^{\text{th}}$ of June 2017

.....

Josef Schaeffer

Abstract

In research areas such as scientific computing and machine learning, Python has established itself as a quasi-standard as a programming language and open source ecosystem over the last decade. Python is based on an interpreter runtime environment, but has been designed from the outset as a domain-specific work environment for statistical calculations and data visualization, and emphasis is placed on simple and robust extensibility through C ++ modules. In order to integrate native compiled, performance-optimized libraries such as LAPACK in Python, until only a few years ago the only way to encapsulate it with C-bindings was to provide it as a module for the Python Runtime API. The rigid and restrictive C API made it difficult to map the full range of functions of an existing library, and in the case of C ++ implementations, such as the DASH Template Library, usually impossible.

From the Python developer community, various solutions and tools have been developed to facilitate the development of native libraries as Python extensions. In addition to the described programmatic problems, conceptual differences are a challenge for mapping C ++ semantics to the slightly typed, interpreted Python environment.

In this thesis the use of a container of the DASH C ++ Template Library for Python extensions is to be investigated. We introduce our Python-binding of the DASH C++ Template Library and provide an implementation of a mapping between conceptual differences between C++ and Python and evaluate the implementation.

Aufgabenstellung

In Forschungsbereichen wie Scientific Computing und Machine Learning hat sich Python im Laufe der letzten zehn Jahre als Programmiersprache und Open-Source-Oekosystem als ein Quasi Standard etabliert. Python basiert auf einer Interpreter-Laufzeit-umgebung, wurde aber von Anfang an als domaenenspezifische Arbeitsumgebung fuer statistische Berechnungen und Datenvisualisierung konzipiert und Wert auf die einfache und robuste Erweiterbarkeit durch C++-Module gelegt. Um nativ kompilierte, performance-optimierte Libraries wie LAPACK in Python einzubinden gab es bis vor wenigen Jahren nur die Moeglichkeit, diese mit C-Bindings zu kapseln und fuer die Python Runtime-API als Modul bereitzustellen. Die starre und restriktive C-API machte es schwer, den vollen Funktionsumfang einer vorhandenen Library abzubilden, und im Fall von C++-Implementierungen, wie der DASH Template Library, in der Regel unmoeglich.

Aus der Python-Entwicklergemeinschaft sind verschiedene Loesungsansaetze und Hilfsmittel entstanden, um die Entwicklung von nativen Libraries als Python-Extensions zu erleichtern. Neben den beschriebenen programmatischen Problemen sind vor allem konzeptionelle Unterschiede eine Herausforderung fuer die Abbildung von C++-Semantiken auf die schwach getypte, interpretierte Python-Umgebung.

In Rahmen dieser Arbeit soll die Verwendung von einem Container der DASH C++ Template Library fuer Python-Extensions untersucht werden.

Посветио мом мишу-мужу

Contents

1 Introduction

About half a century ago, the development of high performance computing (HPC) systems started, of machines that are characterised by delivering extreme computing power or storage capacities. High Performance computing has changed scientific research fundamentally. Simulation as the third pillar of scientific methodology was impossible until having the computing power of those machines and has lead to entirely new insights into our world. HPC systems are machines of high complexity, making their programming a difficult task. Todays HPC systems typically consist of multi-processor and multi-node systems with distributed memory in complex hierarchical topologies. To utilize these systems, scientists need knowledge and experience to use those machines. HPC application developers must take care of an efficient memory management for data structures to reduce effort of moving data on a HPC system. Writing algorithms that leverage the computing power of a multi-node system is asking a deep understanding and knowledge of multi-processor systems, memory management and High Performance computing system topologies. Many programming standards have been developed to make programming of HPC easier. We see the Message Passing Interface, the de-facto interface standard in inter-process communication and we see PGAS, the Partitioned Global Address Space programming model that makes it easier to manage distributed memory resources.

We have developed DASH, a C++ Template Library for Distributed Data Structures with Support for Hierarchical Locality for HPC and Data-Driven Science as a Deutsche Forschungsgemeinschaft project. DASH provides abstract data structures taking the topology of a distributed computing system into account and is also offering parallelized versions of the algorithms of the C++ Standard Template Library. DASH is aimed at programmers to write HPC programs while not having to deal with implementation details of a HPC system and making the program portable but still optimized between different HPC systems. To exploit the full potential of HPC systems, C++ expertise is indispensable as DASH is targeted at C/C++ developers.

In contrast, Python emerged as a de-facto standard for heavy data processing within the data science community and in the industry over the past decade in the scientific computing domain, especially of machine learning. One major reason of the popularity of Python is not only that it is easy to use, but also that it offers the possibility to be easily expendable. It is possible to *bind* native C/C++ libraries for usage in Python as so called modules. On one hand, low development costs are often asked for, especially in the industry, and rapid prototyping is essential due to the speediness of the development, for which C++ might not always be practical

due to its complexity. On the other hand, Python libraries are asked for that offer high performance as well as easiness of use to process data at a high speed. NumPy and SciPy have been developed, widespread scientific Python libraries. NumPy consists of a numerical array and mathematical operations on it being very optimized for fast calculations. SciPy is a collection of powerful libraries that can be used for mathematics, science, and engineering. Some computing centers have already recognized the trend and are offering managed Python ecosystems like the Texas Computing Center to organize the deployment and the optimization of Python in their computer center.

We have decided to provide our DASH library to the Python community to acknowledge this.

From the Python developer community, various solutions and tools have been developed to facilitate the development of native C++ libraries such as DASH as Python extensions. Conceptual differences of C++ and Python are a challenge for mapping C++ semantics to the weak-typed, interpreted Python environment.

In this work, we begin in the second chapter with an characterisation of high performance computing and introduce MPI and PGAS. We give an overview of the features of the C++ Template Library DASH. Thereafter we present concepts of the Python and C++ programming languages, show current usages of Python in the scientific and HPC community and compare the two languages on a conceptual level. In the third chapter, we evaluate available approaches for *binding*, i.e. coupling, native C++ libraries with the Python extensions runtime using suitable criteria with respect to DASH and determine an approach that satisfies our criteria. After that in the forth chapter, we explain the usage of a Python-binding tool, pybind11. Thereafter we present our implementation of a Python-binding for the DASH C++ library, PyDASH. In the fifth chapter, we validate our implementation PyDASH while we derive the validity of the implementation from use cases with respect to the conceptual differences that have been presented before. In the final chapter we give a conclusion and an outlook to future developments.

```
1 import pydash
2 pydash.initialize(0, "")
3
4 myid   = pydash.myid().id()
5 nunits = pydash.nunits()
6 print("Hello World! My unit id:{} team size:{}".format(myid,nunits))
7
8 pydash.finalize()
```

Listing 1.1: Hello World in PyDASH

```
1 $ mpirun -n 3 python hello.py
2 Hello World! My unit id: 0 team size: 3
3 Hello World! My unit id: 2 team size: 3
4 Hello World! My unit id: 1 team size: 3
```

Listing 1.2: Execution of Hello World

2 Background and Related Work

We have analysed the problem statement of this thesis. The problem statement has several dimensions, while each dimension represents aspects of the conceptual scope. In this work, we cover each dimension as far as it is required to understand the problem statement. Figure 2.1 illustrates the conceptual scope.

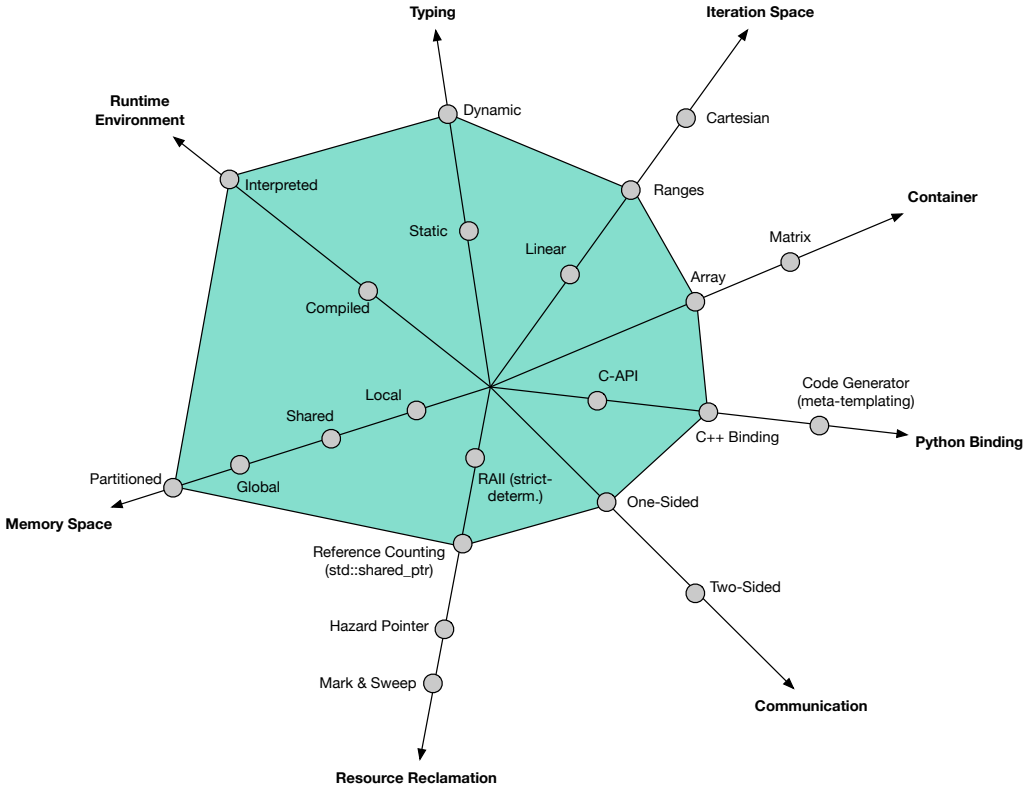

Figure 2.1: Starplot illustrating the conceptual scope of the problem statement, with axis labels representing aspects of the dimensions as they are discussed in the remainder of this work.

In this chapter, we explain the technical backgrounds of this work and present related work to it. We begin with an introduction to High Performance Computing, MPI and PGAS. In addition, we present DASH and give an overview of its features. Thereafter we present concepts of the Python and C++ programming languages and show the current usage of Python in the Scientific and HPC

community. Concluding the chapter, we compare the two languages on a conceptual level with a focus on those concepts that are important for remainder of this work.

2.1 Background

In this section, we provide background knowledge for the understanding of this thesis. We introduce High Performance Computing (HPC) characteristics, MPI, PGAS and DASH.

2.1.1 High Performance Computing

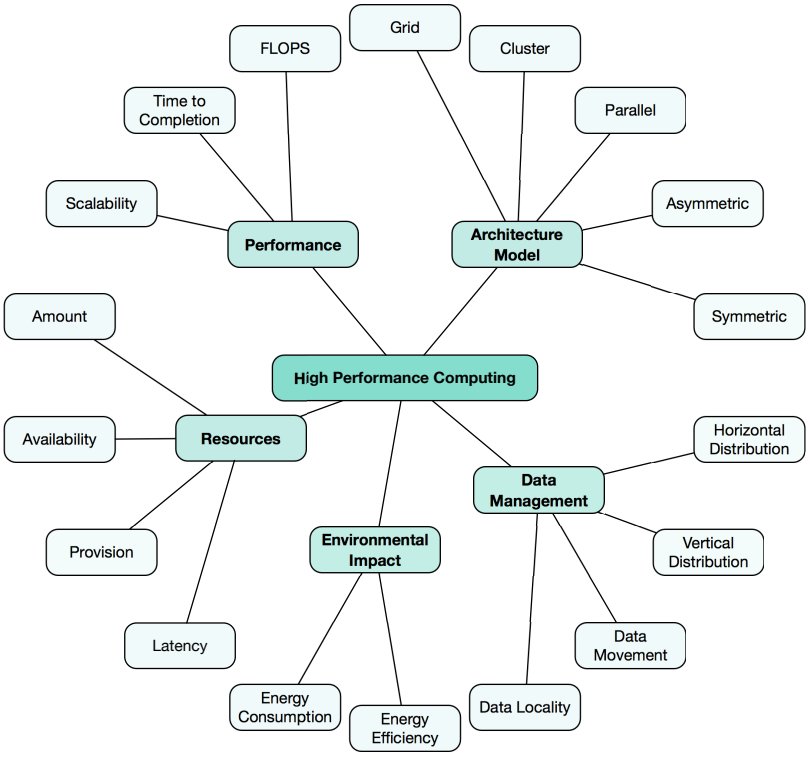

Figure 2.2: Graph describing characteristics and their properties of High Performance Computing (HPC).

HPC in general refers to computing systems that are characterised by the ability to deliver extreme computing power or storage capacities compared to standard workplace computers [1]. As of fast changing industry trends and capabilities, it is hard to define HPC. Therefore we chose an approach to give characteristics of HPC that are rather constant and which can be used to separate HPC from other computing systems.

Figure 2.2 shows characteristics of HPC with their properties.

Single characteristics of the figure can be applied to a variety of computing systems, but the combination of the characteristics with a focus on the maximizing (or minimizing, respectively) of their properties can help to characterize HPC. For example, HPC systems often have a massive parallel architecture model, reaching up to thousands of computing cores. HPC systems consume a lot of energy for computation and cooling, making their maintenance a high cost and asking for an energy efficient design, e.g. the amount of energy spend for a computational operation. They are characterized by providing a large amount of memory with low latency and availability to the whole system, e.g. by shared memory. Data on a HPC system can be distributed horizontally over the machine, while data locality is asked for to keep communication overhead small. The outstanding characteristic of a HPC system is its performance. Measured with FLOPS, short for *Floating Point Operations Per Second*, HPC system offer a very high amount of operations per second, while the time to completion of a computing job is minimal compared to a workstation PC.

2.1.2 MPI

The Message Passing Interface (MPI)[1] is a platform-independent standard for an interface of a message passing library for distributed-memory parallel computing. MPI is based on the consensus of the MPI Forum, consisting of IT companies like Intel, HP, Microsoft and IBM, developers and researchers, is also supported by a large number of open source developers and enjoys their wide-range support. It specifies semantics of a set of library routines and is available for C and Fortran 90. It offers one-sided communication and collective communication operations. Programs using MPI are to a high degree portable because MPI is platform-independent. The standard is implemented by various libraries, and system vendors such aus IBM and Cray offer their own, optimized MPI-implementations e.g. OpenMPI, MPICH, MVAPICH, Intel MPI, IBM MPI.

2.1.3 PGAS

The Partitioned Global Address Space (PGAS) Programming Model provides an abstraction of a global memory address space and presents distributed memory as shared memory [2]. It aims to combine the advantages gained by a global address space as well the performance benefits gained by data locality.

PGAS is characterised by the following traits [3]: A local-view programming model for local and remote data, global address space (which is directly accessible by any process), communication to resolve remote references (sometimes compiler-introduced), one-sided communication for improved inter-process performance and support for distributed data structures.

[1] `http://www.mpi-forum.org/`, visited on 01. June 2017, 08:00 a.m.

For example, Unified Parallel C (UPC), UPC++[4, 5], Co-Array Fortran (CAF), Charm++, STAPL and Chapel [6, 7] are implementations of a PGAS programming system [2].

2.1.4 DASH C++ Template Library

Distributed **D**ata **S**tructures with Support for **H**ierarchical Locality for HPC and Data-Driven Science or, in short *DASH*, is a data-structure oriented compiler agnostic C++ template library[2] implementing a PGAS programming model [8, 9]. It extends PGAS, as it is not classifying data into remote and local, but into various degrees of locality [10]. It reflects the hierarchical machine topology of HPC systems from racks and nodes down to node-level components like NUMA domains and cores. DASH follows the SPMD (Single Program Multiple Data) model with hierarchical additions and is based on existing communication backends. It is open-source software under the BSD-3 license and currently developed as a project founded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG).

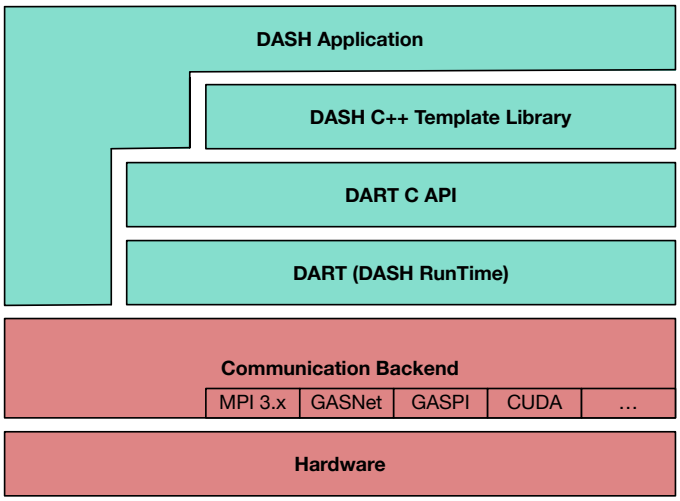

Figure 2.3: Hierarchy of DART and DASH [11]. The green marked layers are components of DASH, the red marked layers are existing components/software

DART and the hierarchy of DASH From a technical perspective, DASH is build on top of DART [2, 8], the **DA**sh **R**un**T**ime, a PGAS runtime system that follows the SPMD execution model. DART offers a plain C-based interface. It is responsible for providing services to the DASH library, including the definition of semantics and the abstraction of the underlying hardware, e.g. the definition of

[2] `http://www.dash-project.org`

units and teams, one-sided access operations and provides collective synchronization mechanisms. Most importantly, DART defines a global memory abstraction. In Fig. 2.3, we see the hierarchy of DASH with an DASH Application at the top. At the bottom of the figure, we have the hardware layer, and on top of the hardware the communication backend like MPI or GASNet. Between the communication backend and DASH, the DART runtime is located, offering an API to the DASH C++ Template library.

Global Pointers and Global References DASH offers the global pointer object `GlobPtr<T>` for the datatype `T`. A global pointer specifies the location of an element in global memory. `GlobPtr<T>` wraps the global pointer provided by DART. A global pointer `GlobPtr<T>` can be converted to a C++ pointer `(T*)` if it is pointing to an address which is local to the calling unit. If not, it will result in a `nullptr`. Dereferencing a global pointer creates a `GlobRef<T>` object which is a global reference proxy object, another type offered by DASH. It resembles C++ references and supports implicit conversion to `T`. Data of a container (e.g. array) accessed by a unit may not be stored in the unit's local memory, so the C++ native reference is not possible as a return value. If a unit tries accessing data stored in a container, it is returned a proxy reference `GlobRef<T>`. It is possible to create a `GlobRef<T>` from a `GlobPtr<T>`. Further, DASH offers a `GlobIter<T>` datatype with a random access opertator to operate over global memory. A `GlobIter<T>` can be converted to a `GlobPtr<T>`.

Units & Teams Units [8] are individual participants with computational and storage capabilities in a DART/DASH program. They correspond to thread/process/image in other PGAS implementations (cf. threads in UPC [4] and images in Co-Array Fortran). Their implementation details remain within the runtime system. The number of units of a DASH program is defined at its start-up and remains unchanged until the end of its execution. Units are organized in teams. Teams are an ordered sets of units that can be dynamically created and destroyed at the runtime of a program. Teams form the basis for all collective synchronization, communication and memory allocation operations in DASH. All units are

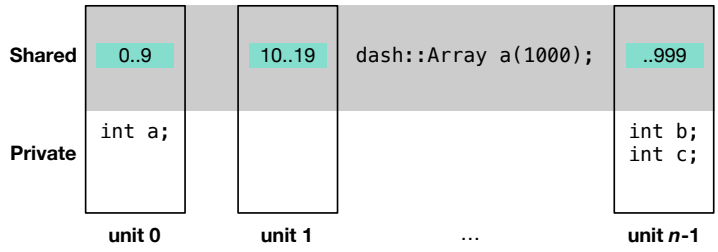

Figure 2.4: DASH implementing a PGAS-like programming model [11]. It shows a container, the DASH array that is distributed over several units.

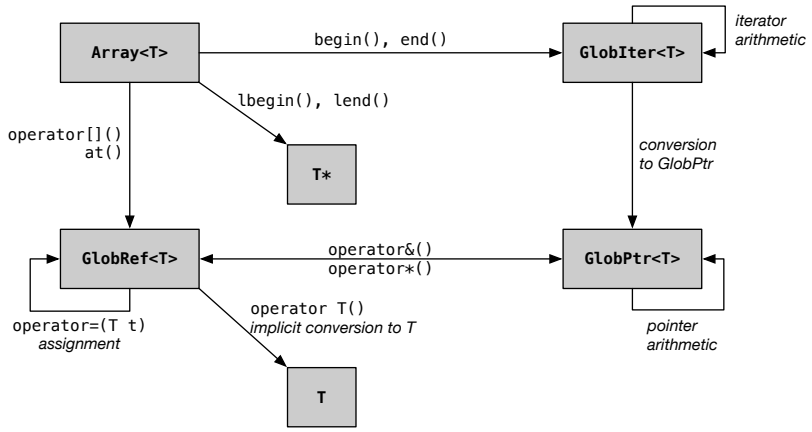

Figure 2.5: Interplay of central abstractions in DASH [11]

members of the root team `dash::Team::All()`, from which all other teams are derived using methods offered by DASH. Subteams are subsets of a parent team. Teams can be constructed to reflect the machine topology by predicting so called *Locality Domain Hierarchies*, taking into account various information e.g. from PAPI, `hwloc` [12] and the operation system about the machine's topology.

Data Distribution Patterns DASH offers patterns for the distribution of data. In general, the mapping of an index space onto units is controlled by a pattern [8]. Patterns need no specification of a datatype and no memory allocation is performed. They guarantee that a similar mapping is used for different containers [8].

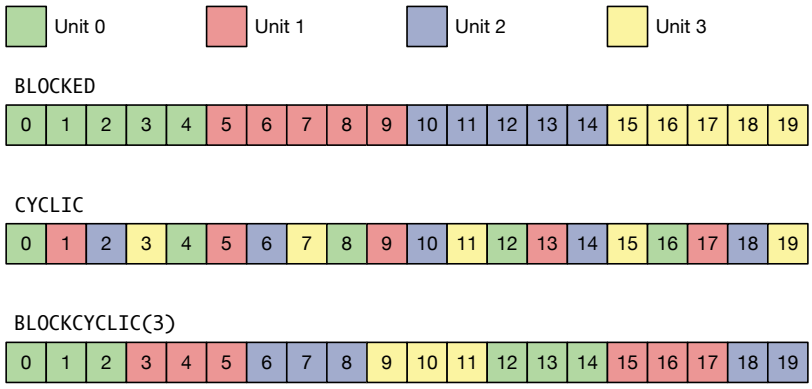

Figure 2.6: The one-dimensional data distribution patterns supported by DASH. [11]. The data is distributed with `BLOCKED`, `CYCLIC` and `BLOCKCYCLIC(3)` patterns.

Patterns offer different distribution schemes, e.g. `BLOCKED`, `CYCLIC`, `BLOCKCYCLIC()`. For instance, `BLOCKED` will let every unit store a contiguous block of elements of size number of elements divided by number of units [11]. Figure 2.6 illustrates a one-dimensional data distribution with `BLOCKED`, `CYCLIC` and `BLOCKCYCLIC(3)` patterns.

Container Concepts DASH offers hierarchical PGAS-like abstractions for data, e.g. multidimensional arrays [13], lists, hash tables, etc. It follows a global-view approach, but also supports local-view programming with explicit method calls [8, 11]. Algorithms of the C++ Standard Library (STL) [14] can be used, because DASH containers implement the STL container interface conventions.

A common distributed container [8] is the one-dimensional array, the DASH Array. It represents a generic, fix-sized container class, corresponding to the STL array class `std::array`. A DASH array is either constructed over a specified team or the root team `dash::Team::All()` as a default, and the data of the array is distributed to the team's units.

```
1 /* Allocation of a globally shared array of 100 integers */
2 dash::Array<int> array_1(100);
3
4 /* array_2 is allocated over team Team::All().split(2) */
5 dash::Array<int> array_2 (1000, dash::Team::All().split(2));
6
7 /* Allocation of array_3 using the BLOCKCYCLIC pattern scheme */
8 dash::Array<int> array_3 (20, dash::BLOCKCYCLIC (3) )
```

Listing 2.1: Usage of a DASH Array [11]

Listing 2.1 shows three use cases of a DASH array. In line 2 we can see the allocation of a globally shared array of 100 integers. The array is distributed by default as `BLOCKED` over all units, as no pattern and no team is specified. In line 7 we allocate an array to the half of all teams. We allocate an array using a pattern scheme in line 10, using the `BLOCKCYCLIC` pattern.

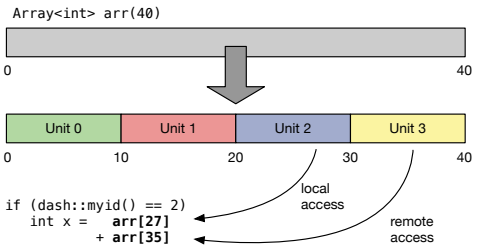

Figure 2.7: Local and remote access. [11]

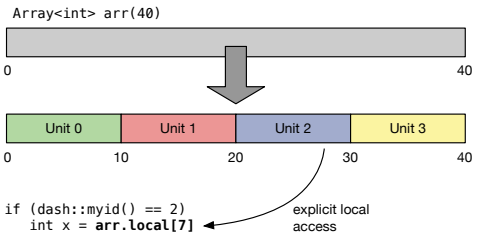

Figure 2.8: Explicit local access. [11]

Figures 2.7 and 2.8 show the access of an array. In both figures, local data is accessed that is stored at the (global) array position 27. In Fig. 2.7 also remote data is accessed. We can explicitly ask to access local data, which can be seen in figure 2.8. This can lead to an performance increase compared to non-local access.

Algorithms Additionally to the container concepts, iterator-based algorithms of the C++ STL have been rewritten in DASH. For example, `std::find()` can be called with `dash::find()`.[3]

Conclusion and Outlook DASH container concepts follow C++ STL interface conventions. Therefore, porting existing *STL-conform* C++ programs to DASH can be seen as a process of changing `std::` to `dash::` gradually. This lowers the efforts to start writing programs with DASH that can exploit the capabilities of (parallel) machines.

According to the project's roadmap, it is planned to include dynamically growing and shrinking container and a general task-based execution model in the runtime.

Example To conclude the section about DASH, we are presenting a program using the DASH library in List. 2.2.

With `dash::init(&argc, &argv)` in line 8, the programmer initializes the runtime, and with `dash::finalize()` (line 30), the programmer can release resources and end the runtime. In line 10, local data is being initialized. In line 13, a DASH array is created and filled with 0 (line 15). In line 17, a global reference is set on the last element. The unit with ID 0 (line 19) sets a global pointer to the last element. The method `dash::barrier()` (or `dash::Team::All().barrier()`) is a barrier to synchronize all units (line 25). Comparable to MPI, `myid()` (or `dash::Team::All().myid()`) returns the ID of the active unit (line 27 and 28). To get the number of all units, the programmer can call `size()` (not in this example).

2.2 Comparing the Python and C++ Programming Languages

In this section we introduce the Python and C++ programming languages. We present current uses of Python in the scientific and HPC community. Following this, we compare the conceptual differences of Python and C++ that are relevant for the scope of this thesis.

[3] Currently, theses algorithms are provided: `copy()`, `copy_async()`, `all_of()`, `any_of()`, `none_of()`, `accumulate()`, `transform()`, `generate()`, `fill()`, `for_each()`, `find()`, `min_elment()`, and `max_element()`.

```
#include <libdash.h>
#include <iostream>

using namespace std;

int main(int argc, char * argv[])
{
  dash::init(&argc, &argv);
  /* private scalar and array */
  int p; doubles[20];
  /* globally shared array of 1000 integers */
  dash::Array<int> a(1000);
  /* initialize array to 0 in parallel */
  dash::fill(a.begin(), a.end(), 0);
  /* global reference to last element */
  dash::GlobRef<int> gref = a[999];

  if (dash::myid() == 0) {
    /* global pointer to last element */
    dash::GlobPtr<int> gptr = a.end() - 1;
    (* gptr) = 42;
  }

  dash::barrier();

  cout << dash::myid() << " " << gref << endl;
  cout << dash::myid() << " " << a[0] << endl;

  dash::finalize();
}
```

Listing 2.2: Stand-alone DASH program with important DASH structures [11]

2.2.1 Python

Python is a high-level, interpreted and object-oriented programming language [15]. It has a simple, clear and concise syntax and is characterized by dynamic typing. Python is not only available on most modern computing platforms, making Python code portable, but Python also has a large standard library [16] which can be applied to many different kinds of problems, e.g. string processing, internet protocols, software engineering and operating system interfaces. Python is expandable by writing extensions in C and C++ [17], an important feature for our work (more in chapter 4). Programs and algorithms can be written in C and C++, compiled and encapsulated as so-called *modules*, e.g. in order to implement new built-in object types and to call C library functions and system calls [17]. Modules can be imported into a Python environment and accessed within there [18].

Runtime Python is executed in a run-time interpreter environment. The most frequently used interpreter is CPython[4], which is the reference implementation of the Python language and is written in C. Another commonly used interpreter of Python is PyPy. It is an alternative implementing of the Python language [19] written in Python itself. PyPy is applying a just-in-time compiler (JIT) on the Python code that is executed and compiles it to machine code, trimmed to speed [19].

Scope of an object The accessibility of an object is defined by the scope of the object. Python offers different scopes of an variable [20]. If an object is constructed inside a local scope, e.g. inside a function definition, without assignment to a variable, the object is assigned to the local scope. Aside from local scope, Python (since version 3) also offers the ability for nonlocal and global scope, which can be declared explicitly, e.g. for nonlocal with the keyword `nonlocal` before variable definition.

Garbage Collection and Reference Counting Memory management in Python is handled by a Garbage Collector (GC). In general, a GC is an automated method of managing, especially of freeing up resources. Figure 2.9 shows four different types of GCs.

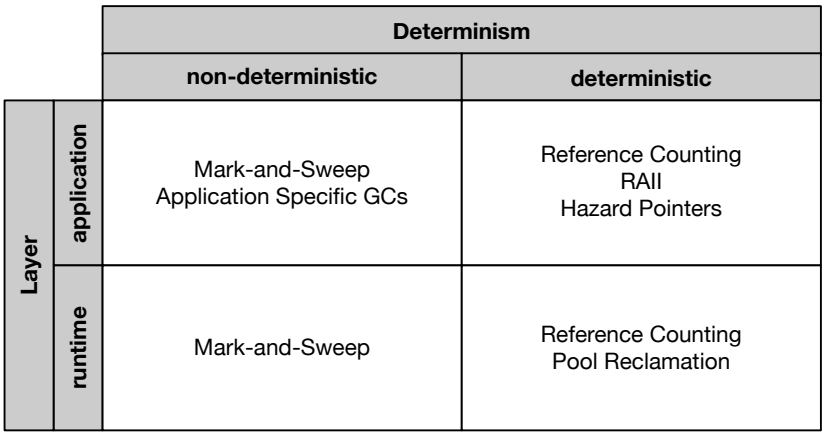

Figure 2.9: Matrix classification of four different types of Garbage Collectors (GC). GCs can be classified by their layer (application and runtime) and their determinism (deterministic and non-deterministic). Matrix entries are examples of such GCs.

It can not be predicted when resources is freed if a non-deterministic GC is given, e.g. running the same program on the same machine can lead to a different usage of resources. This is in contrast to a deterministic GC, where it is possible

[4] `https://www.python.org`, visited on 01. June 2017, 08:00 a.m.

to predict resource freeing beforehand if constraints are not changed. Layers in the figure describe that either the application itself is implementing its own GC or the runtime offers a GC. All combinations of runtime / application based GCs and deterministic / non-deterministic GCs exist (see Fig. 2.9 for examples).

Python uses an garbage collector [21] with an automatic memory allocation and deallocation method. Every time a new object is created, the acquisition of memory is handled automatically by the GC of the Python runtime. For the deallocation of memory, Python uses *reference counting* to detect objects that are no longer needed.[5] In general, reference counting is a method to keep reference of the usage of an object: every time the object is acquired, the reference count is increased by 1, every time it is released, the counter is reduced by 1. If the reference counter reaches 0, the object's memory is freed by the GC automatically, because it ican be assumed that the object is no longer used by anyone. Figure 2.10 shows reference counting on an object with 3 accessors.

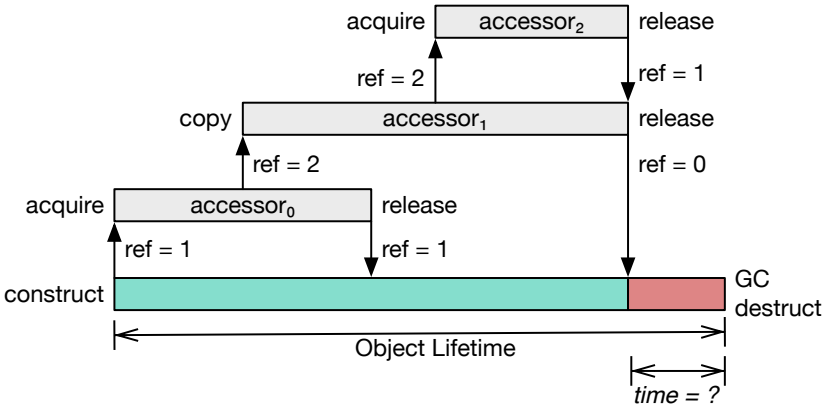

Figure 2.10: Reference counting on an object with three accessors. Accessor_0 acquires the object after construction with reference counter (RC) is set to 1. While accessor_0 holds the object, accessor_1 copies the object and increases the RC by 1 to 2. Accessor_0 releases the object and reduces the RC by 1 while the object is hold by accessor_1. After object is released by accessor_0, accessor_2 is acquiring the object from accessor_1. The RC is then increased to 2. accessor_2 is releasing the object while it is held by accessor_1, reducing the RC by 1 to be 1. After that, the object is released by accessor_1 and RC reduces by 1. The RC is set to 0. The object, which is no longer referenced, is collected by the GC the next time it is running. The time between the garbage collection and the moment, the object's RC is 0, is not defined. After the GC collected the object, it's resources is freed.

Garbage Collection in Python is deterministic and but can be both application

[5] In this work, we only consider Python 2.0 and higher, as the interpreter of older versions of Python used different garbage collectors.

and runtime-based, because it is possible to explicitly call the GC with `import gc` and `gc.collect` in an application but also use automatic reference counting in the runtime (`gc.enable()`)

Python in Scientific and High Performance Computing

The following related works cover each different aspects of our research. We present a Python extension for a scientific environment and look at different approaches of Python in the area of HPC where it is used in various contexts and on different levels, from the provision of accesses to HPC systems to the usage in programs and algorithms.

NumPy N-dimensional arrays are introduced by the open-source NumPy package [22] as a part of the SciPy ecosystem[6]. It was first published in 2006 and is written in C [23]. The package provides a high-level data structure, the *NumPy array*, which is a multidimensional collection of elements as well as a set of mathematical functions and operations on the NumPy array. NumPy arrays can be seen as the standard representation for numerical data [22] in the Python community. The usage of NumPy is intuitive, e.g. transposing a matrix `x` is as simple as calling `x.T`. NumPy is implementing mainly three ways to improve computational performance: It improves performance by using vectorising calculations instead of for-loops, by combining arrays to execute numerical computation on those arrays using a technique called broadcasting in order to minimise the amount of operations and with it's own memory description semantics, called memory mapping, that make it possible to manipulate data without the need to copy all data from a disk but only directly addressing the data needed.

As of their wide spread in the Python community and their diverse user base, reaching from the academic world to industry, from game development to space exploration, we see a clear demand for efficient and optimised Python extensions that can exploit the performance of a machine. NumPy substantiates the feasibility of such an extension.

PyTACC The number of Python users in HPC are growing and becoming a significant portion of the HPC community. Recognising this, the Texas Advanced Computing Center (TACC) has developed its own best practises to provide its users a managed Python ecosystem within its HPC environment [24]. The aim of the administration of the TACC was to "provide a Python environment that maximises usability, flexibility, and performance"[24]. Using the Lmod module system[7], a hierarchical module system to encapsulate the dependencies of Python distributions, versions, libraries etc., Python installations are deployed by the Linux RPM Package Manager within the TACC and also maintained with it.

[6] `https://www.scipy.org`, visited on 01. June 2017, 08:00 a.m.
[7] `https://github.com/TACC/Lmod`, visited on 01. June 2017, 08:00 a.m.

Starting a Python application induces the loading of many files and modules into system memory. To avoid I/O bottlenecks, it was found out that installing dynamic libraries and Python packages locally can prevent those bottlenecks. Using compilers and compiler flags that are optimal for the TACC computer system architecture, it was possible to optimise frequently used Python packages, therefore many users are profiting (indirectly) by using those packages without explicitly needing to optimise their code for the specific TACC system architecture. Different kinds of benchmarks where undertaken to measure the performance of the Python installations. For example, it could be proven that the selection of architecture specific compiler flags when building Python could provoke major improvements to overall productivity considering the number of Python jobs run at TACC – even though only small performance achievements were gained by those compiler flags. TACC also offers education material and tutorials for its users to improve their knowledge of their "HPC-optimised" Python and to maximise their performance achievements.

mpi4py MPI for Python (mpi4py) is an extension module [25] that implements MPI to be used with Python. It is implemented with Cython, a programming language for writing C extensions for the Python language[8]. Important features like process groups and communication domains, intra-communicators, point-to-point communication and extended collective operations are available. Semantics satisfy the standard specifications of the MPI Forum.

```
1  from mpi4py import MPI
2  import numpy as np
3  comm = MPI.COMM_WORLD  // Represents all processes available at start
       -up
4  rank = comm.Get_rank() // Internal process number
5  array1 = np.arange(2**16, dtype=np.float64)
6  array2 = np.empty(2**16, dtype=np.float64)
7  wt = MPI.Wtime() // Returns an elapsed time on the calling processor.
8  if rank == 0:
9      /* Performs a standard-mode blocking send: */
10     comm.Send([array1, MPI.DOUBLE], 1, tag=0)
11     /* Performs a standard-mode blocking receive: */
12     comm.Recv([array2, MPI.DOUBLE], 1, tag=7)
13 elif rank == 1:
14 comm.Recv([array2, MPI.DOUBLE], 0, tag=0)
15 comm.Send([array1, MPI.DOUBLE], 0, tag=7)
16 wt = MPI.Wtime() - wt
```

Listing 2.3: Python Program using mpi4py

Listing 2.3 shows an example program using mpi4py [26]. We can see that MPI standard methods are available, e.g. in line 3 with `MPI.COMM_WORLD` to get the

[8] `http://cython.org`, visited on 01. June 2017, 08:00 a.m.

number of processes at start-up or in line 4 with `Get_rank()` to get the caller's own process number. In line 10 and 13 we can see the standard blocking send and receive methods. Concluding it can be stated that the usage of MPI in Python via mpi4py is similar to it's usage in C. mpi4py covers parts of the conceptual scope of our thesis, but lacks aspects like partitioned and global memory space.

PyGAS PyGAS [27] is a prototype implementation of a PGAS extension library to Python and can be used as an module with a simple `import pygas`. It follows the Partitioned Global Address Space memory model and the SPMD execution model for which it is designed. It requires the GASNet Communication Library, "a networt primitive implementing parallel global address space SPMD languages"[9], where it is built upon. As no modification to the Python interpreter is required, it can be run from any Python installation base. One interpreter is executed per SPMD rank, and one or multiple threads per interpreter. The Proxy object is the primary PyGAS primitive for representing shared entities [27]. Proxies are initialised with an actual object and operations on those proxies are caught by Python's attribute resolution system, transmitted over the network and executed on the actual object [27]. The Proxy object can be viewed as a pointer-to-shared, storing its owner's rank and the address of the actual object [27]. Whichever thread called a method on a Proxy object, the method is executed on a thread with proximity to the actual object. According to [27] PyGAS has significant productivity advantages while being comparable to mpi4py in its performance. We agree with the authors that the ability to rapid prototyping with PyGAS might justify a slight loss of performance. The usage of PyGAS follows the usage of PGAS[10] in other programming languages. Listing 2.4 gives an example program in Python using PyGAS.

As we can see in this listing, we have PGAS constructs available and their usage is similar to C. E.g., in line 4, we can get the number of all processes with `THREADS` and in line 7 the own thread index (with `MYTHREAD`). If we compare this to listing 2.3, we can see that the semantics are also very similar to MPI.

While PyGAS covers already many aspects of the scope of our thesis and already provides solutions, it has remained in a prototype stadium. [27]

Conclusion We have seen that Python is already in use in the scientific and in the HPC community. We see a clear demand for efficient and optimised scientific extensions to Python. There are already projects to deal with the distribution and management of Python installation bases in a HPC environment like PyTACC. We have also seen that there are already experiments and functional implementations of communication backends like MPI and PGAS to help Python run on HPC systems and support a parallel and distributed computing environment with

[9] `https://gasnet.lbl.gov`, visited on 01. June 2017, 08:00 a.m.

[10] `https://github.com/mbdriscoll/pygas/blob/master/misc/examples/grid.py`, visited on 01. June 2017, 08:00 a.m.

```
from pygas import *

# THREADS is numer of threads working independently
assert THREADS is 4
# MYTHREAD specifies own thread index (from 0 until THREADS-1)
row      = MYTHREAD / 2
col      = MYTHREAD % 2
rowteam = TEAM_WORLD.split(row, col) # Split TEAM_WORLD in sub-teams
colteam = TEAM_WORLD.split(col, row)
r        = rowteam.broadcast(MYTHREAD, from_rank=0)
c        = colteam.broadcast(MYTHREAD, from_rank=0)

assert r == row * 2
assert c == col

TEAM_WORLD.barrier() # Barrier for all threads in TEAM_WORLD

if MYTHREAD == 0:
    print "Success."
```

Listing 2.4: Python program using PyGAS

communication between computing nodes (mpi4pi, PyGAS).

2.2.2 C++

C++ is an ISO-standardized general purpose, paradigm-agnostic programming language based on the C programming language [28]. It is one of the most used and widest spread programming languages[11] and it is available for most computing platforms. C++ can use the hardware of the machine where it is executed efficiently and allows low- and high-level development. The C++ Standard Template Library (STL) [14] defines a collection of classes and functions providing utilities, generic containers, algorithms and input/output and many other features [29]. The STL is part of the C++ ISO Standard [30].

Value Categories Theoretical background on so-called value categories is required to understand the management of the ownership and the lifetime of an object in C++. In C++, each expression, e.g. an operator with is operands, a literal or a variable name, consists of two independent properties: a *non-reference type* and a *value category*. The category is either *prvalue*, *xvalue* or *lvalue*, categorised according to the taxonomy as to be seen in picture 2.11.

They are formally defined as in references [32] and [31]. For the remainder of this work, we will use the following, simplified definition: We consider lvalues as variables that are referenced by a variable name such that their address can be

[11] `http://www.lextrait.com/vincent/implementations.html`, visited on 01. June 2017, 08:00 a.m.

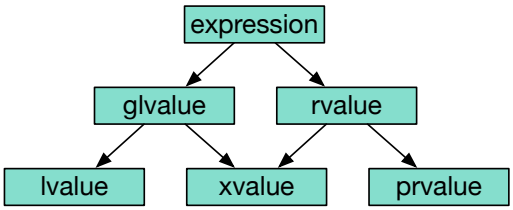

Figure 2.11: Taxonomy of Value Categories [31]. We read the taxonomy as following: a glvalue is an lvalue or an xvalue, a prvalue is an rvalue that is not an xvalue, and so on.

obtained using the address-of operator `&`, while it is not possible to give an address to a rvalue. This might not include each detail of the definition as in the before mentioned references, but this definition is a pragmatic approach and is sufficient enough for the understanding of this thesis.

Ownership Ownership is defined by an object that owns a resource. We differentiate between *unique* and *shared* ownership. In the case of unique ownership, the resource is owned by one object and released by the deconstructor if the object does not exist anymore. In the case of shared ownership, a resource is owned by multiple objects. The resource is released if all objects do not exist anymore [33].

Move Semantics The storage in a computer is usually divided into memory and hard drive or flash storage. The performance of data management can be increased by reducing copying data on memory or between the memory and the hard drive/flash storage. Instead of copying data, temporary objects, e.g. return values of functions, should be *moved*, meaning handing over the address of an object instead of the actual values and the ownership over the object. To support *moving data*, C++ offers so-called move semantics [34] (since C++11). One element of the move semantics are *move constructors*. A move constructor assigns resources held by the argument (e.g. pointers to dynamically-allocated objects), instead of copying them. It leaves the argument in a valid but indeterminate state. For example, if an object is moved, it may leave the argument in an empty state [34]. Move semantics rely on *rvalue references* which were introduced in C++11 [35]. The lifetime of a temporary object is extended by rvalue references and lvalue references to const, but objects are not modifiable through lvalue references [35]. Listing 2.5 shows a comparison of rvalue and lvalue references [35].

Resource Management In resource management, namely memory management, we differentiate between *Resource Acquisition* (RA) and *Resource Reclamation* (RR). RA is the process of reserving a resource and marking it as *occupied*, while RR is the process of releasing a resource and marking it as *available*.

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string s1 = "Test";
// std::string&& r1 = s1; /* error: can't bind to lvalue */

    const std::string& r2 = s1 + s1; /* OK: lvalue ref to const
     extends lifetime */
// r2 += "Test"; /* error: can't modify through ref to const */

    std::string&& r3 = s1 + s1; /* OK: rvalue ref extends lifetime */
    r3 += "Test"; /* OK: can modify through ref to non-const */
    std::cout << r3 << '\n';
}
```

Listing 2.5: *rvalue references* vs. *lvalue references* [35]. We see that it is not possible to assign a lvalue to a rvalue reference. We can bind a lvalue reference to a constant. It is not possible to modify a constant value. We can bind a temporary lvalue to a rvalue reference. Rvalue references can be modified.

An object is created by the constructor in C++. During the creation, or more specifically, the *initialisation*, of an object, resources, i.e. memory, are *allocated*, in other terms, *acquired*. If an object is no longer referenced, e.g. because it is out of scope, the destructor is called to destruct, or more specifically, to *finalise*, the object. While this happens, the memory is released or, in other terms, resources are *deallocated*.

In C++ [36], a common practice[12] is *Resource acquisition is initialization* (RAII) that is used in many object-oriented languages (c.f. section 14.4 in [33]). In RAII, it is guaranteed by the language that the resources are held from end of initialisation until begin of finalisation, because "holding the resources is a class invariant"[13]. The guarantee is only given if the object is not out of scope but alive.

Resource management in C++ can be seen as as deterministic and application-based garbage collection (c.f. section 2.2.1 for definitions). It is deterministic, because an object is immediately deallocated after it no longer referenced and application-based, because the deallocation is invoked by the flow of the program, the application, and not the runtime.

[12] `http://www.stroustrup.com/bs_faq2.html#finally`, visited on 01. June 2017, 08:00 a.m.
[13] `https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization`, visited on 01. June 2017, 08:00 a.m.

Table 2.1: Comparison of the Syntax between Python and C++

Description	Python	C++
Including a module/library	`from math import *`	`#include <cmath>`
Assignment Operators	`=, +=, -=, *=, /=, %=`	same
Type Integer Value	`int, long`	`short, int, long`
Type Decimal Value	`float`	`float, double, long double`
Type Boolean	`True/False` or `(not 0)/0`	`true/false or (!0)/0`
Type Character	**none**	`char`
Type String	`str`	`char mystring[50], string`
For-Statement	`for i in range(10):`	`for(i = 0; i < 10; i++)`
If-Statement	`if x != 3:`	`if (x != 3)`
While-Statement	`while x != 3:`	`while (x != 3)`
Break out of a loop	`break`	**same**
Function definition	`def myfunction():`	`int myfunction()`
Function call	`myfunction()`	**same**
And Operator	`and`	&&
Or Operator	`or`	||
Not Operator	`not`	`!`
Comparison Operators	`==, !=, <, <=, >, >=`	**same**
Arithmetic Operators	`+, -, *, /, %`	**same**
Comments	`#`	`//, /* */`
Pre/Post In-/Decrement Operators	**none**	`++x, x++, --x, x--`
Code Blocks	indentation, ;	{ }
Statement Separator	end of line	;
Constants	**none**	`const int interest = 42`

2.2.3 Conceptual Differences between Python and C++

In this section, we will compare the Python and C++ programming languages with respect to the requirements of our Python-binding of DASH (c.f. chapter 4). Generally speaking, Python is a general-purpose, dynamically typed, interpreted, interactive, high-level programming language whereas C++ is a statically typed, multi-paradigm and compiled programming language with the ability for low and high-level development. Statically typing means that is is required in C++ to explicitly declare types, while in Python this is not required but happening dynamically. In fact, in Python a function can accept an parameter of any type, and (might) give a return value of any type. Python code is exectued in-time through the Python interpreter whereas C++ code needs compilation before execution. This induces that Python code needs to be interpreted every time it is executed, leading to a poorer performance than C++. Table 2.1 shows syntactic differences between Python and C++ [15]. We notice that Python expressions are often shorter than C++, e.g. the for-statement. This makes Python code in general shorter than C++ code what supports *rapid prototyping*.

Iterators and Ranges The *Iterator Concept* is a concept of the C++ Standard Template Library (STL). It can be iterated over a container if it implements the iterator concept without an index of the container or even knowledge of the

container type. Iterators can be viewed as a generalised form of pointers. For example, STL algorithms are specified that they use iterators to access containers, and access containers not directly. Iterators are not available in Python. Although Python offers a language construct called *iterators*, they are not comparable to the C++ STL concept. A Python iterator can be seen as a *range* [37]. To exemplify the differences between iterators and ranges, see List. 2.6 and List. 2.7. Both listings show the iteration over an array. Listing 2.6 shows an iterator over an array. It is noticeable that it is possible to iterate not only in one direction, but to manipulate the iteration order within the iteration scope as in line 10 where the iterator is reduced by 2. This is not possible with ranges. In a range, the program can only get to the next value. Listing 2.7 shows the iteration over an array behaving like a range.

```
1  int a[5] = {1, 2, 3, 4, 5};
2
3  /* This is an iterator: */
4  for (auto iter = a.begin();
5       iter != a.end();
6       ++iter)
7  {
8    *iter *= 2;
9    /*Not possible with ranges!*/
10   if (cond) iter -= 2;
11 }
```

Listing 2.6: Usage of an Iterator

```
1  int b[5] = {1, 2, 3, 4, 5};
2
3  /* This is a range: */
4  for (int & value_ref : b)
5  {
6
7
8    value_ref *= 2;
9
10
11 }
```

Listing 2.7: Usage of a Range

Although ranges seem therefore inflexible as they only allow one direction of iteration, they can speed up the computation time. Using an iterator to iterate over an array, the program can not *predict* which values it does *not* have to calculate. This is only possible with ranges: one can express transformations or list comprehensions comparable to Haskell. Listing 2.8 illustrates this.

```
1 std::array<int, 10> my_array {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2 my_array | drop(6) | multiply(2) | take(2)
3
4 // Iterator:  {6, 7, 8, 9} -> {12, 14, 16, 18} -> {12, 14}
5 // Ranges: Calculated only {6*2, 7*2} -> {12, 14}
```

Listing 2.8: Comparison of data manipulation in an array using iterators ./. ranges

In the first line, we create an array of length 10 containing int values. In line 2, we see the instructions of the manipulation of the array: the first 6 values of the container are dropped, each of the remaining values of the array are multiplied by 2 and finally, the first 2 values are returned. If the array is an iterator, the first 6 values are dropped, four values are multiplied by 2 and the first two values are returned. Assuming the array is a range, we do not need to calculate all values because we can predict which values have to be manipulated. In this example, only the values at array position 6 and 7 are returned, therefore only these two

values need to be multiplied by 2. This saves computation time. In line 4 and 5 we can see the steps of calculation in an iterator and a range in detail. Therefore we see that a range can help saving computation time because it can predict which values are returned.

A reference implementation of ranges in C++ can be found on GitHub[14] and from the C++ *boost.Library* [38]. There is also a proposal to include ranges into the STL [39].

Lifetime of Objects in C++ and Python The management of the lifetime of an object depends on the memory management of the programming language.

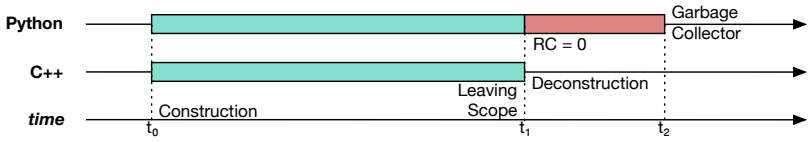

Figure 2.12: Timeline of the lifetime of objects in Python and C++ from construction to release of their resources.

Memory management, and specifically the release of memory, is handled in both Python and C++ by a Garbage Collector. C++ and Python have different types of Garbage Collectors. We have seen that Python uses a deterministic application/runtime-based GC while C++ uses an deterministic application-based GC. Garbage Collection in Python is handled by reference counting, in C++ by calling the deconstructor after an object is no longer needed, e.g. out-of-scope.

Figure 2.12 illustrates the timeline of the lifetime of an object in C++ and Python. Two objects are constructed in Python and C++ at the same time t_0. At t_1, the objects run out-of-scope. In C++, the deconstructor is immediately called and the memory is freed. In Python, the reference counter is reduced by 1 and reaches 0. The object's memory will deallocated by the Garbage Collector.

[14] `https://github.com/ericniebler/range-v3`, visited on 01. June 2017, 08:00 a.m.

3 Evaluation of Binding Alternatives

In this chapter we compare available approaches for coupling native C++ libraries with the Python extensions runtime. "Coupling" C++ libraries with Python is called *binding* C++ to Python [40]. In the evaluation, we consider the following criteria with respect to the DASH library:

Expressiveness Sufficient to maintain DASH container concepts and algorithm semantics.

Extensibility Required efforts to extend Python bindings to new definitions in the DASH library.

Efficiency Static overhead caused by the Python interpreter and static latency overhead in executing latency and memory consumption (temporaries).

Maturity The development status and the stability of an approach.

3.1 Python C-API

The Python C-API is the base of the standard Python interpreter (CPython). API is the shorthand notation for Application Programming Interface. The Python C-API allows developers to access the Python interpreter in C and C++ [41], mainly for two different purposes:

First, it allows writing extension modules that can be called in a Python environment and *extend* the functionality of Python. Second, it can be used to *embedd* Python in an application which means using Python as a component in a larger application.

Expressiveness The Python C-API is very expressive and offers a large set of features of C++ to be bound. It offers all language features that are required to bind DASH definitions to Python.

Extensibility Binding code using the Python C-API requires a lot of hand-written static code, e.g. for parsing arguments or to construct the return type. [40] If DASH definitions change, the static code must be adjusted or extended, what causes extending bindings to be a lot of work with many possible sources of error. Reference counting needs to be treated by hand having a high error rate. Tracking down those errors is a big effort [42].

Efficiency Although in general a lot of hand-written static code is required to wrap C++ code, no redundant but only necessary code is needed to bind the C++ code. This reduces the overhead code to a minimum, making the bindings themselves efficient.

Maturity The Python C-API is not forward compatible due to frequent changes to the Python language by the Python consortium. As changes are not foreseeable, the stability in general cannot be guaranteed. Therefore, the Python C-API is not stable. It is mature due to a long development time.

Conclusion The disadvantages of the Python C-API, namely handling reference counting by hand, show that the Python C-API is not suitable for a DASH Python-binding.

3.2 SWIG

SWIG (Simplified Wrapper and Interface Generator) is an open-source and operation system independent programming tool [43], or more precisely, an interface compiler that is able to connect code written in C and C++ to other programming languages, including Python.

Expressiveness SWIG automatically generates the binding code in C from declarations in header files of C or C++ code using the Python C-API. SWIG supports a wide range of C++ features [44], at least as many as DASH requires currently.

Extensibility If DASH definitions change, extending the binding is as easy as just calling SWIG with the C++ source files and letting SWIG create new bindings. Unfortunately, SWIG introduces its own specialised language for customising inter-language bindings. To use SWIG, the programmer has to learn this language causing extra effort to the programmer. [42]

Efficiency Yet, these advantages come with a draw back: SWIG creates *a lot of* static overhead code, fast reaching a few thousand lines of code, lowering the execution time enormously compared to alternative binding tools [19]. The compile time is extended and exceedingly long [45].

Maturity SWIG has been developed over a long period of time. It is continuously maintained and new features are added on a regular basis.

Conclusion With its low efficiency due to the high overhead, SWIG is not suitable for a DASH Python-binding, as this is in contrast to our efforts to make DASH as well it's binding as efficient as possible.

3.3 Boost.Python

Boost.Python [42] has been developed in the context of *boost*[1], a collection of peer-reviewed open source C++ libraries. It aims to offer '"seamless interoperability between C++ and the Python programming language" [46].

Expressiveness The Python C-API is implemented by Boost.Python. It offers a large feature set [40], while the actual scope of covering C++ language features is only partly documented, leaving us with a certain lack of clarity. Therefore we are not able to answer to what extent we can express all DASH definitions with boost.Python, letting alone unforeseeable future changes of definitions.

Extensibility The interface of Boost.Python is easy to understand and intuitive, and the static code to bind C++ code is relatively short [42]. It can be used to directly connect C++ classes to Python without the C++-Interface of Python [47]. It does not introduce a separate wrapping language, making its usage more convenient [42].

Efficiency While Boost.Python is using C++ template meta-programming and even C-Macros, e.g. `BOOST_PYTHON_MODULE`, tracing down errors becomes very complicated and time consuming [40].

Maturity We assume that the maintenance of boost.Python is not continued and the development has been abandoned. (The last update has been released in 2015).

Conclusion Mainly due to the abandoned maintaining of boost.Python and the unclear scope of covering C++ language features boost.Python is not suitable for a DASH Python-binding.

3.4 CFFI

The C Foreign Function Interface (CFFI) is a module developed in the context of PyPy [19, 48]. The goal of CFFI is to provide a way to easily access C code from Python applications. It is supporting CPython and PyPy, in which it is built in[2].

Expressiveness CFFI can be imported into a Python session and can be used in different modes to automatically create bindings of C code or to link C binaries with Python. As CFFI only supports C but not C++, we cannot use it for DASH, because we cannot express all DASH definitions in C. Problematically, even some C99 constructs are not supported [48].

[1] `http://www.boost.org`, visited on 01. June 2017, 08:00 a.m.
[2] `https://pypy.org/compat.html`, visited on 01. June 2017, 08:00 a.m.

Extensibility Type declarations and function signatures are parsed by CFFI automatically, therefore they do not need to be rewritten in Python before they can be used [49].

Maturity CFFI is under continuous development with new features added regularly.

Conclusion We find CFFI is a very interesting and promising approach and will watch its future development, but as it does not support C++, it is not a practical candidate for binding a C++ library.

3.5 cppyy

cppyy is a Python module to automatically generate bindings of C++ code to Python at runtime supporting both PyPy and CPython. It requires a backend, the Cling C++ interpreter, based on Clang/LLVM, to parse the C++ code to get the reflection information to build the Python bindings on top [50].

Expressiveness Many language features of C++ are covered and well documented [50], making cppyy expressive, e.g. a templated class `std::vector<int>` in C++ is accessible from Python via `std.vector(int)` using cppyy.

Extensibility cppyy is an automatic binding generator. Creating a new binding for changed DASH definitions is as easy as just execution the binding generator. As the high level constructs of C++ can be reduced to C code by accessing the abstract syntax tree (AST) using the Cling C++ interpreter, CFFI can be applied to wrap the resulting C code. The automatic binding generation is supported by the possibility in C++11 (and above) to give annotations to the code, e.g. to indicate ownership [19].

Efficiency The performance of cppyy seems to be nearly identical to a native execution with a nearly unmeasurable Python overhead. Data access is reduced by two orders of magnitude and the overhead of calls to C++ functions from Python by an order of magnitude compared to the equivalent in CPython [19]. The authors' aim was to bring "the same performance to PyPy for C++ as CFFI [is] provid[ing] for C" [19].

Maturity *No information available.*

Conclusion Although being a very promising approach for C++ bindings, cppyy is not a suitable candidate for a Python-binding, because for us it is not traceable how cppyy generates bindings while we want to control the process.

3.6 PyCXX

PyCXX[3] is a C++ toolset for writing Python bindings based on Boost.Python.

Expressiveness PyCXX is integrating the C++ STL itself in order to write C++ programs as 'natural' as possible. PyCXX does not support wrapping C++ classes as new Python types, what is essential to a DASH binding, because DASH introduces a lot of custom data types [42].

Extensibility The Python C-API is encapsulated in classes to take care of exception handling and reference counting, reducing at least one frequent origin of errors[3].

Efficiency Although the syntax of PyCXX is clear and intuitive, it requires a lot of static overhead code to bind C++ to Python.

Maturity The project seems to be no longer maintained [40]. The stability cannot be guaranteed and we cannot be sure if it is possible to integrate future changes of DASH definitions depending on changes of C++ standards.

Conclusion Because of the abandonment of the project, PyCXX is not a suitable candidate for a Python-binding.

3.7 pybind11

pybind11 is a header-only library to bind C++ code to Python and to embed Python code in C++. It is based on Boost.Python, of which it sees itself a successor[4].

Expressiveness We are able to express all DASH definitions with the available features of pybind11. Among other features, pybind11 can map the following features from C++ to Python, e.g. functions accepting and returning custom data structures, single and multiple inheritance, iterators and ranges, STL data structures, smart pointers with reference counting like `std::shared_ptr` and internal references with correct reference counting.

[3] `https://sourceforge.net/projects/cxx/` and `http://cxx.sourceforge.net`, visited on 01. June 2017, 08:00 a.m.

[4] `https://github.com/pybind/pybind11/blob/master/README.md`, visited on 01. June 2017, 08:00 a.m.

Extensibility The simplicity of use and mature concepts make it easy to extend a binding with new DASH definitions. Writing bindings in pybind11 for C++ is as simple as just telling the compiler with a simplistic syntax which classes, methods and attributes one wants to bind to Python.

Efficiency pybind11 is very efficient. It offers a easy syntax to clearly define ownership of objects. The return object of a function can be annotated to indicate transferred, shared or unique ownership (return value policy, in 4.4 more detailed). This reduces temporaries and unnecessary copies of an object.

Maturity Although this project is relatively new, it has already reached a solid basis and is very mature. Having a fast growing number of supporters and contributors on GitHub with new features added regularly, we are confident that the project becomes even more stable by fully supporting all C++ language constructs.

Conclusion pybind11 covers the requirements for a Python-binding of DASH. It allows us to express all definitions of DASH, new language definitions can be added with small effort. Due to its unique concepts like return value policies, which are not available in any other binding tool that we have evaluated, its bindings become efficient. We are confident that the project is see a strong growth as it already has a fast growing number of supporters.

3.8 Deciding on a Binding Method

In the previous section several binding alternatives have been evaluated using defined criteria. We use the results of the evaluation and present a summary in the Tab. 3.1. For each criterium that was evaluated, we are indicating with the symbol + that the binding approach meets the requirements for a DASH binding, the symbol - indicates that such requirements are not met. As we can see in the table, only pybind11 meets all requirements for a Python-binding. Therefore the decision to choose pybind11 has been easy. In the next chapter, we present our implementation of a Python-binding of the DASH library with pybind11.

Table 3.1: Results of the Evaluation of Binding Alternatives. Symbol + (-) indicates requirements for DASH (not) fullfilled.

	Expressiveness	Extensibility	Efficiency	Maturity
Python C-API	+	-	+	-
SWIG	+	+	-	+
Boost.Python	-	+	-	-
CFFI	-	+	+	+
cppyy	+	+	+	?
PyCXX	+	+	-	-
pybind11	+	+	+	+

4 Binding DASH to Python: PyDASH

We begin this chapter with a reference implementation of a Python-binding of a C++ function with pybind11. By explaining the reference implementation, we show the process of writing a Python-binding.

After that, we present our implementation of a Python-binding for the DASH C++ library, PyDASH. While the DASH-API follows the C++ STL interface conventions, PyDASH does not. It is not supposed to imitate the C++ STL, but is targeted at Python developers, and therefore uses the Python standard as a model for its API. We want PyDASH to be *pythonic*. Therefore, the name *PyDASH* is an analogy to many other Python libraries using the "Py-" syllable, like NumPy. PyDASH is publicly available as part of the DASH source distribution[1].

We illustrate the usage of PyDASH with a short Python program. Using the program, we also exemplify the execution of a PyDASH program.

Being one key difficulty, we explain ownership of objects that are passed between Python and C++ and present a solution how to solve ownership with pybind11.

4.1 Reference Implementation via pybind11

In this section we show a short reference implementation of a Python-binding of a simple function written in C++ [51]. Consider the add-function written in C++ in List. 4.1:

```
1 int add(int i, int j) { return i + j; }
```

Listing 4.1: C++-function to add two integer values

Binding this function to Python with pybind11, we get the code as in List. 4.2: When calling a module from the Python interpreter with `import`, a function is called which is automatically created by the `PYBIND11_PLUGIN()` macro. The expression `py::module m("example", "pybind11 example plugin")` defines and creates a module. In this case, we name the module `example` and provide the *docstring* which is describing the module and is required by Python. To actually bind the `add()` function to Python, we use the expression `m.def("add", &add, "A function which adds two numbers")`, which creates the wrapping code. `return m.ptr()` returns a pointer to the internal Python object `m` to Python.

Assuming having stored the code of List. 4.2 in a file `example.cpp`, we can compile and invoke it. Listing 4.3 shows the compilation of the file, the import

[1] `https://github.com/dash-project/pydash`

```
#include <pybind11/pybind11.h>

int add(int i, int j) { return i + j; }

namespace py = pybind11;

PYBIND11_PLUGIN(example)
{
   py::module m("example", "pybind11 example plugin");
   m.def("add", &add, "A function which adds two numbers");
   return m.ptr();
}
```

Listing 4.2: pybind11 binding of the add-function in List. 4.1

of the created module into an interactive Python runtime and the invocation of the function.

```
$ c++ -O3 -shared -std=c++11 -I ./pybind11/include `python-config --
    cflags --ldflags` example.cpp -o example.so
$ python
Python 3.5.0 (default, Jun 07 2017, 15:39:39)
>>> import example
>>> example.add(1, 2)
3L
```

Listing 4.3: Python Session to compile, load and execute the example.cpp file

4.2 Hello World!

Every DASH program is framed by initialisation and finalisation of the DASH runtime. This invariant stays true for PyDASH. To write a Python program using PyDASH, no special efforts are required; only the initialisation and finalisation of the DASH runtime are obligatory. Language constructs of PyDASH follow the semantics of Python. The module hierarchy of PyDASH is corresponding to the class and namespace hierarchy of DASH. For example, the DASH Barrier `dash::barrier()` can be called in PyDASH with `pydash.barrier()`.

Listing 4.4 outlines of a simple PyDASH program. In line 1, the PyDASH module is imported. Similar to a DASH program, PyDASH is initialising DASH in line 3 with `pydash.initialize(0, "")`. The ID of the active unit can be obtained with `pydash.myid().id()`, the number of units with `pydash.nunits()` (line 5 and 6). Finally, the programmer can release resources and end the DASH runtime with `pydash.finalize()` as in line 10.

PyDASH is not introducing any exotic language constructs but keeps to the Python logic. The programmer can focus on writing his (parallel) program, while the backends of parallel execution are well hidden from him with no complex

```python
import pydash

# Initialize the DASH Runtime: dash::init(&argc, char *argv[])
pydash.initialize(0, "")

# Get the ID of the active unit: dash::myid();
myid    = pydash.myid().id()

# Get the number of units: dash::size();
nunits = pydash.nunits()

print("Hello World! My unit id:{}team size:{}".format(myid, nunits))

# Finalize the DASH Runtime: dash::finalize()
pydash.finalize()
```

Listing 4.4: Hello World in PyDASH

language constructs visible to him that could be source of errors. Therefore we believe that rapid prototyping with PyDASH is no problem.

4.3 Program Execution

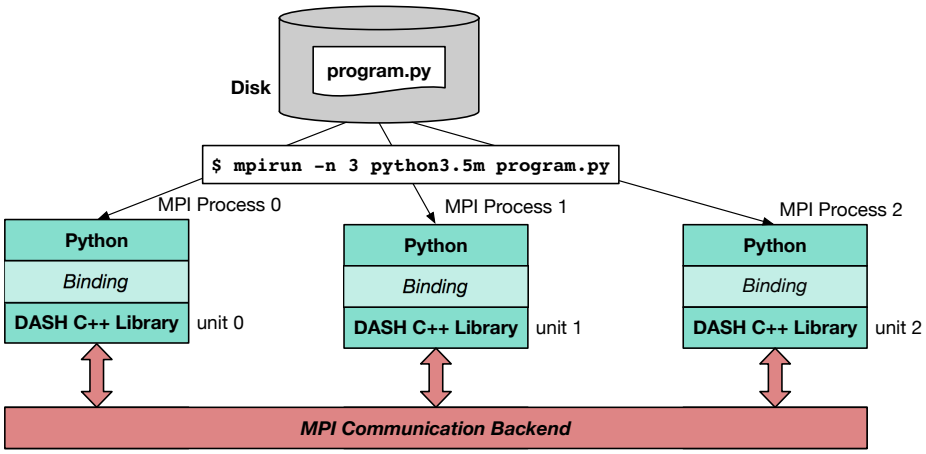

Figure 4.1: Execution of a PyDASH program.

A PyDASH program can either be executed in an interactive Python session or executed on a MPI-basis with `mpirun`. As the Python runtime is single threaded, any PyDASH program that is executed interactively in a Python session is single threaded with no possibility to exploit parallelism. This execution environment is supported by PyDASH as an environment with one unit with ID `0` and team size `1`.

In the following we concentrate on an environment that is targeted by DASH, an MPI-based execution environment that supports an arbitrary number of units and is able to exploit true parallelism. Like an execution of an DASH-MPI program written in C++, we can execute any PyDASH program written in Python with the simple terminal command `mpirun -n X python program.py`, where `X` is an integer number defining the number of MPI processes. The command will run X copies of the Python runtime with each of them executing the program that is written in `program.py`.

In List. 4.5 we see the execution and its result of the program of List. 4.4. The program prints out a message with its unit ID and team size to the terminal standard output.

```
1 $ mpirun -n 3 python3.5m program.py
2 Hello World! My unit id: 0 team size: 3
3 Hello World! My unit id: 2 team size: 3
4 Hello World! My unit id: 1 team size: 3
```

Listing 4.5: Execution of a PyDASH program in a Bash Shell

In Fig. 4.1 we see the execution details of a PyDASH program. The program `program.py` is stored on the disk and executed with the command `mpirun -n 3 python3.5m program.py`. The command creates 3 copies of a Python runtime (version 3.5m in this example) and assigns each to one of the three MPI processes. The Python runtimes will execute their identical copies of the program. Each program will interact with the DASH C++ library, which itself is communicating with the MPI-backend. The MPI-backend assigns to each node a number (0, 1 and 2). This ID of a MPI process is identical to the ID of a DASH unit. The ID can be accessed by the PyDASH program through the DASH C++ library.

We conclude that the execution of a PyDASH program doesn't impose any heavy efforts on the programmer. It does not require any changes to the MPI backend of the system where it is executed (provided that the system offers a MPI backend). The communication between units is handled by the DASH C++ library itself with no manual action through the programmer required. All in all, we believe it is simple and easy to use.

4.4 Lifetime Management of PyDASH Objects

We have seen that Python and C++ use different mechanisms of resource reclamation and subsequently lifetime of objects. Figure 4.2 shows us the control flow of construction and destruction of an object between Python as a *user* and C++ as a *source*. We call objects that have been created by PyDASH, *PyDASH objects*, e.g. a data type of PyDASH is called PyDASH data type.

Ownership of PyDASH Objects When passing an object between Python and C++, the ownership of the object must be clear. The owner and only the owner

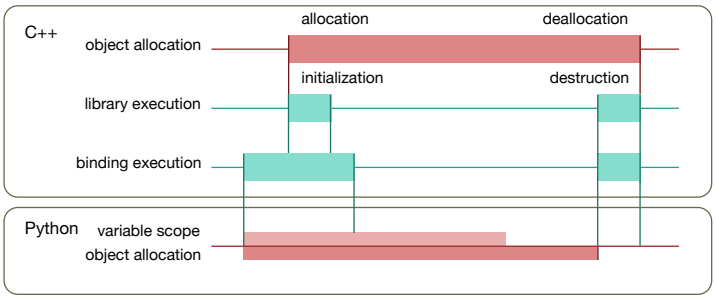

Figure 4.2: Construction and destruction of an object between Python and C++/DASH

of the object is responsible for the lifetime management of the object. If the ownership of an object is unclear, the program may run into fragmentation of memory or undefined behaviour causing probable program failures. Implementations of Python-bindings have to take this into account. To highlight this, we consider the example in List. 4.6:

```
1 /* Method returning a pointer to a static data structure: */
2 Data * get_data() { return _data; }
3 /* ... */
4 /* Binding code */
5 m.def("get_data", &get_data);
```

Listing 4.6: Function returning object without defining its ownership [52]

For this example, we assume that the return object, which has a native C++ type, is wrapped into a usable Python type, when `get_data()` is called from the Python interpreter. In this case, it is a pointer object to a static data structure. It is not immediately clear who is owning the static data structure object after the method call. This can imply several possible problems:

First, it may be assumed that Python takes over the ownership of the object. Python uses reference counting. If the reference counter of the object reaches 0, the object is deallocated by the Python Garbage Collector. It can be possible that there are still references to this object on the C++ side of which the Python side is not aware of because it has no information about it. This would lead to a definite failure of the program.

Second, we assume Python does not take over the ownership of the object. Consequently, Python is not be able to free the resources of this objects as it has no rights to do so. But there is no instruction for the C++ side to call the deconstructor. This means that the object might stay alive without being referenced to by any side, neither from Python nor from C++. This would result in fragmented and wasted memory.

If the ownership is not clearly defined, we conclude that initialised memory or free data structures may be accessed leading to hard-to-debug non-determinism

Table 4.1: Return value policies offered by pybind11.

Return Value Policy	Description	Owner of Return Object	Responsible for Destruction	Safety
`take_ownership`	Reference existing object and shift ownership (C++→Python).	Python	Python	No, undefined behavior if C++ calls destructor or Data not dynamically allocated.
`copy`	Create new copy.	Python	Python (C++ for source object)	Yes, because objects are decoupled.
`move`	`std::move` to move value contents into new instance.	Python	Python (C++ for source object)	Yes, because objects are decoupled.
`reference`	Reference existing object.	C++	C++	Undefined behavior if C++ deletes object which is still used by Python.
`reference_internal`	Reference existing object.	C++	C++	Prevents the parent object to be garbage collected if return value is still referenced by Python. Lifetime tied to calling this / self argument.
`automatic`	if the return value is a ... **pointer** → `take_ownership`; **rvalue** → `move`; **lvalue** → `copy`			
`automatic_reference`	This is the default conversion policy for function arguments when calling Python functions manually from C++ code (i.e. via handle::operator()). Use reference, if return value is pointer			

and segmentation faults [52].

Return Value Policies pybind11 offers a solution for managing the ownership of a passed object. The ownership of a passed object can be defined by an annotation in the binding code. These annotations are called *return value policies* (RVP). Table 4.1 gives an overview of all return value polices that are offered by pybind11. The usage of RVP annotations requires knowledge of the *value category* (cf. chapter 2.2.2) of the passed object. For example, it is not possible to move an object if it has no *rvalue reference*, therefore annotating an object with a RVP `move` may not work as expected. Another example, if the RVP annotation `automatic` is used, the handling of the passed object depends on its category. Cf. the row entry of `automatic` in Tab. 4.1.

We apply RVPs to the binding of List. 4.6 and get List. 4.7. The ownership of the passed object of the function is shifted from C++ to Python as an effect of this annotation. Python as the owner of the passed object is then responsible for the deconstruction of the object.

```
/* Binding code */
m.def("get_data",
      &get_data,
      py::return_value_policy::take_ownership);
```

Listing 4.7: Function Passing an Object with a move of the ownership from C++ to Python [52].

5 Validation of Implementation

In the prevoius sections 2.2.3 and 4.4 we discussed conceptual differences of Python and C++/DASH. In our implementation of DASH, PyDASH, we had to overcome those differences and find a mapping to bring the different concepts together. We believe that our implementation PyDASH has solved the conceptual differences and is able to map the concepts from C++/DASH to Python using pybind11.

In this chapter, we validate our implementation PyDASH in respect to the mapping. We are not formally verifying PyDASH, because it is not feasible in the scope of this work. We derive the validity of the implementation from use cases with respect to the conceptual differences, namely the management of lifetime of PyDASH objects, the mapping of DASH iterators and the usage of DASH algorithms.

In the evaluation, we proceed with the following steps:

1. Describing the *Evaluation Aspect*, the subject of consideration.
2. Setting the prerequisites in *Initial Situation*
3. Presenting our methodology in *Validation Method*.
4. Outlining our anticipation in *Expected Behavior*.
5. Validating the Evaluation Aspect in *Evaluation*
6. Presenting our conclusion *Result*.

5.1 Automatic Deallocating Resources of Objects in PyDASH

One of the conceptual differences between Python and C++/DASH that we examined were the methods of Python and C++ of deallocating resources that are out-of-scope. In this section, we evaluate that it is possible to map the different concepts of deallocation with pybind11.

Evaluation Aspect We validate that pybind11 takes care of correct reference counting on objects and evaluate that PyDASH objects are deallocated deterministically as it can be expected by reference counting.

Initial Situation We create PyDASH objects inside a local scope. After creation, their reference counter should be set to a positive, non-zero number while they are in scope.

Validation Method We write a program that (first) creates one/several PyDASH object/-s inside a local scope that (second) assigns the object/-s to a variable/-s and that (third) runs out of the local scope of the variable/-s, leaving the object/-s for destruction. The PyDASH object is implemented by a custom, logged value type. The logged value type is characterised that it logs the address of the object and its creation/copy/movement/destruction. When the PyDASH program is out-of-scope of the variables, we consider the moment and order of deallocating PyDASH objects.

Depending on the moment of the destruction of the object, we can interpret that reference counting is working as anticipated. We differentiate the following possible outcomes:

(First) If the object is deallocated after it is out-of-scope, the RC must have invoked its deallocation, because the RC is deterministic and should have reduced the reference count to 0, and also because the C++ side is not aware of the scope of an object on the Python side.

(Second) If the objects are destroyed after the end of the Python runtime, C++ has called deconstructor. In this case, the reference counting was either not working properly or it was not involved in the deallocation of the objects. We use the log that is produced by the logged value type to trace the flow of the program.

Expected Behavior We expect that objects are deallocated after the object is out-of-scope because reference counting is deterministic and handled by pybind11. The deallocation of several objects should be in inverse order of their construction.

Evaluation In the following evaluation, we consider several scenarios. We are starting with a basic scenario, where only one PyDASH object is examined, advancing to more complicated scenarios, where we consider multiple objects that are interlaced in nested loops and/or stored in containers.

Deallocation of Single PyDASH Object We start with the most basic scenario. We use a program that creates one PyDASH object of a logged value type inside an if-statement-block, assigns the object to a variable inside that if-statement-block and leaves the if-statement-block. The program is written as stated in List. 5.1.

In the program, the if-statement-block is used to create a local scope for variable a. If the program leaves the if-statement-block, a is out-of-scope and is destructed by the RC. For other test scenarios, we also use if-statement-block (and for-loop-blocks) to create local scopes for objects.

Executing the program produces the log as in List. 5.2.

```
def test_deallocation(x):
  if x:
    print("Entering Scope")
    a = pydash.LV(1,"A")
    print("Leaving Scope")

  print("Left Scope")
```

Listing 5.1: Deallocation of Single PyDASH Object

```
>>> test_deallocation(True) #Invocation of program.
Entering Scope
[ LOG | logged_val(n,s) | @:0x...020 --- create A
Leaving Scope
Left Scope
[ LOG |   ~logged_val() | @:0x...020 --- destroy A and free data
```

Listing 5.2: Deallocation of Single PyDASH Object Log

As we can see in the log, the program created the PyDASH object after it entered the local scope. We expected that the reference counter invokes the deallocation of the PyDASH object after it is no longer in scope. In the log, we can reproduce that the object is deallocated after it is out-of-scope. This indicates that reference counting handled by pybind11 works as expected.

Deallocation of Multiple PyDASH Objects Now we consider a scenario involving multiple PyDASH objects. Comparable to the previous scenario but with multiple objects, we write a program that enters a local scope realized by a for-loop, creates each time it enters the scope one PyDASH object separately and leaves the local scope again. We expect that every time one object runs out-of-scope it is deallocated.

We can see the code in List. 5.3.

```
def test_deallocation():
  for x in range(0, 10):
    print("Entering For-Scope, Time: " + str(x))
    a = pydash.LV(1, str(x))
    print("Leaving Scope")
  print("Left Scope")
```

Listing 5.3: Deallocation of Multiple PyDASH Objects

We can see in the log (List. 5.4) that the program is working as described above. After leaving the local scope and reentering it, it deallocates the previously created object which is now out-of-scope. This indicates that the reference counting is working because the deallocation is deterministic and occurring when expected.

```
>>> test_deallocation(True)
Entering For-Scope, Time: 0
[ LOG | logged_val(n,s) | @:0x...de0 --- create 0
Leaving Scope
Entering For-Scope, Time: 1
[ LOG | logged_val(n,s) | @:0x...f50 --- create 1
[ LOG |   ~logged_val() | @:0x...de0 --- destroy 0 and free data
Leaving Scope
...
Entering For-Scope, Time: 9
[ LOG | logged_val(n,s) | @:0x...f50 --- create 9
[ LOG |   ~logged_val() | @:0x...de0 --- destroy 8 and free data
Leaving Scope
Left Scope
[ LOG |   ~logged_val() | @:0x...f50 --- destroy 9 and free data
```

Listing 5.4: Deallocation of Multiple PyDASH Objects Log

Deallocation of Multiple PyDASH Objects in Nested Scopes In this scenario, we consider the creation of an PyDASH object inside a local if-statement which is nested inside a for-loop. We realized the described approach in a program as in List. 5.5. As in the previous scenario, we expect that the objects are deallocated after the program left their individual scope.

```
def test_deallocation(y):
  for x in range(0, 10):
    print("Entering For-Scope, Time: " + str(x))
    if y:
      print("Entering If-Scope")
      a = pydash.LV(1, str(x))
      print("Leaving If-Scope")
    print("Leaving For-Scope")
  print("Left Scope")
```

Listing 5.5: Deallocation of Multiple PyDASH Object in Interlaced Scopes

As we can see in the log (List. 5.6), an object is deallocated after the program run out of it's local if-scope. Therefore the deallocation is as expected. Again, the reference counting was handled as anticipated by pybind11. This indicates that reference counting is handled correctly.

Deallocation of Multiple PyDASH Object in Container In this scenario, we use a program that creates a container in a local if-statement-block. While the program is inside the if-statement-block, the container is filled with PyDASH objects. The program is implemented in List. 5.7. We expect that the objects are deallocated after the program left the scope of the container in reverse order.

As we can see in the log (List. 5.8), PyDASH objects are created each time the

```
>>> test_deallocation(True)
Entering For-Scope, Time: 0
Entering If-Scope
[ LOG | logged_val(n,s) | @:0x...060 --- create 0
Leaving If-Scope
Leaving For-Scope
Entering For-Scope, Time: 1
Entering If-Scope
[ LOG | logged_val(n,s) | @:0x...200 --- create 1
[ LOG |   ~logged_val() | @:0x...060 --- destroy 0  and free data
Leaving If-Scope
Leaving For-Scope
...
Entering For-Scope, Time: 9
Entering If-Scope
[ LOG | logged_val(n,s) | @:0x...200 --- create 9
[ LOG |   ~logged_val() | @:0x...060 --- destroy 8  and free data
Leaving If-Scope
Leaving For-Scope
Left Scope
[ LOG |   ~logged_val() | @:0x...200 --- destroy 9  and free data
```

Listing 5.6: Deallocation of Multiple PyDASH Object in Interlaced Scopes Log

```
def test_deallocation(y):
  if y:
    print("Entering If-Scope")
    l = []
    for x in range(0, 10):
      print("Entering For-Scope, Time: " + str(x))
      l.append(pydash.LV(1, str(x)))
      print("Leaving For-Scope")

    print("Leaving If-Scope")
  print("Left Scope")
```

Listing 5.7: Creation and Destruction of Multiple PyDASH Object in Container

program enters the for-loop. After the program has left the last for-scope and also the if-scope, the container that is storing the PyDASH objects is out-of-scope. As anticipated, the PyDASH objects are destroyed in reverse order of their creation. Again, the deterministic behavior of deallocation shows us that reference counting is working as expected.

Result We evaluated the mapping between Python and C++/DASH using pybind11 concerning the deallocation of objects and build test scenarios to validate this. We were able to use the log created by the logged value type to trace the flow of the program. We have expected that objects are deallocated after the pro-

```
>>> test_deallocation(True)
Entering If-Scope
Entering For-Scope, Time: 0
[ LOG | logged_val(n,s) | @:0x...510 --- create 0
Leaving For-Scope
Entering For-Scope, Time: 1
[ LOG | logged_val(n,s) | @:0x...130 --- create 1
Leaving For-Scope
...
Entering For-Scope, Time: 9
[ LOG | logged_val(n,s) | @:0x...5c0 --- create 9
Leaving For-Scope
Leaving If-Scope
Left Scope
[ LOG |   ~logged_val() | @:0x...5c0 --- destroy 9 and free data
...
[ LOG |   ~logged_val() | @:0x...130 --- destroy 1 and free data
[ LOG |   ~logged_val() | @:0x...510 --- destroy 0 and free data
```

Listing 5.8: Creation and Destruction of Multiple PyDASH Objects in Container Log

gram runs out-of-scope of the individual object. Our test scenarios show that this is the case. This indicates that the reference counter is responsible for deallocation of the objects and/or containers holding those objects. Therefore we assume that reference counting is working as expected and the mapping of Python and C++/DASH is implemented correctly.

5.2 Passing a Global Object from C++ to Python

Another important point of a Python binding is the handling of ownership of objects that are passed between C++ and Python. In this section we evaluate the ownership of a global object that was created by PyDASH and is passed from C++ to Python using return value policies. We differentiate three scenarios and treat every scenario individually.

Evaluation Aspect A global PyDASH object in C++ is passed over to Python. We consider the ownership of the object before and after the passing.

Initial Situation A global PyDASH object is created by C++. After the creation of the object, the ownership of the object is taken by C++ because the C++ constructor obtained the ownership through the allocation of the memory of the object.

Validation Method We implement a method that passes a global object from C++ to Python by returning it and bind this function to Python with pybind11,

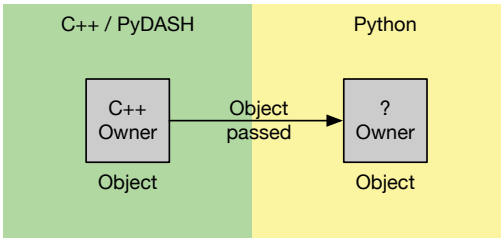

Figure 5.1: Global Object is passed from C++/PyDASH to Python. The owner after the object is passed is unknown.

annotaing the ownership of the returned object using return value policies (RVP) and/or shared pointer constructs offered by pybind11. The global PyDASH object is implemented by a custom, logged value type. The logged value type is characterised that it logs the address of the object and its creation/copy/movement/destruction.

We write a program that (first) calls the binding of the before mentioned method from Python inside a local scopethat (second) assigns the return object of the function to a variable and that (third) runs out of the local scope of the variable, leaving the object for destruction.

For the destruction, we differentiate two options: (First) If the object is deallocated immediately after it is out-of-scope, the reference counter (RC) must have invoked it's deallocation, because the RC is deterministic and because the C++ side is not aware of the scope of an object on the Python side. (Second) If the objects are destroyed after the end of the Python runtime, C++ has called deconstructor at the end of the runtime. In this case, the RC was not working properly or was not involved.

Depending on the invoker of the destruction of the object, we can interpret the ownership, because only the owner of an object is subject to destroying it. We use the log that is produced by the value type to trace the flow of the program.

Expected Behavior We expect that the object is passed from C++ to Python with the ownership as stated by the RVP attribute or the shared pointer, respectively.

Evaluation For the evaluation, we differentiate 3 scenarios:

1. Passing a copy from C++ to Python
2. Passing a shared pointer from C++ to Python
3. Moving the object from C++ to Python

Scenarios

We investigate each scenario individually.

Passing a copy from C++ to Python

We have modified PyDASH to instantiate a global object of value type logged_val at start up and and implemented a function `return_global_object_copy()` to pass the global object with the RVP attribute `return_value_policy::copy` as in List. 5.9

```
/* ... */
pydash::logged_val g_object = pydash::logged_val();
pydash::logged_val return_global_object() { return g_object; }
/* ... */
// Binding:
m.def("return_global_object_copy", &return_global_object,
          "Create and return logged_val object copy",
          py::return_value_policy::copy);
```

Listing 5.9: Passing a Copy of a Global Object from C++ to Python

We expect that the function passes a copy of the global object to Python and that Python obtains the ownership of the copied object. We use the program in List 5.10 to test this assumption.

```
def test_return_global_object_copy(x):
  if x:
    print("Entering Scope")
    a = pydash.return_global_object_copy()
    print("Leaving Scope")
  print("Left Scope")

test_return_global_object_copy(True)
```

Listing 5.10: Python Test of Passing a Copy of a Global Object from C++ to Python

In the program, the if-statement is used to create a local scope for variable a. If the program leaves the if-statement-block, a is out-of-scope and is destructed by the RC. Executing the program gives us the log of List. 5.11:

As we can see from the log, the global object is created by importing of PyDASH. When the program is invoked with a true condition, the program enters the if-statement-block and calls the function `return_global_object_copy` that creates a copy of the global object, as documented by the log. Still in the local scope, the copied object is moved, because the assignment of the copied object to the variable a in the local scope induces a move operation, see section 2.2.1 for explanation.

```
>>> import pydash
[ LOG |                logged_val | @:0x...a28 ooo --- default
    construct X
.....
>>> test_return_global_object_copy(True)
Entering Scope
[ LOG | logged_val(const self &) | @:0x...a60 --- create copy of @0x
    ...a28: X
[ LOG | logged_val(const self &) | @:0x...a60 --- copied value
[ LOG |      logged_val(self &&) | @:0x...fe0 --- move  * <-- @0x...
    a60: X
[ LOG |            ~logged_val() | @:0x...a60 --- destroy X
Leaving Scope
Left Scope
[ LOG |            ~logged_val() | @:0x...fe0 --- destroy X
>>> exit()
[ LOG |            ~logged_val() | @:0x...a28 --- destroy X
```

Listing 5.11: Python Test of Passing a Copy of a Global Object from C++ to Python Log

The copied object is destroyed after it was moved for the assigment. After the program has left the if-statement-block, the local variable is out-of-scope. If a object is out-of-scope, it is destroyed by the reference counter. As expected, the object is destroyed. The end of the Python runtime invokes the destruction of the *source* global object. This shows that our statement in the *Initial Situation* paragraph was correct that C++ has ownership of the global object. We can exclude that the GC of Python destroyed the global object because it has no awareness of the global object, because the global object is on the 'C++ side'. Concluding we see that the handling of the global object and its copy has remained within each's environment and the lifetime of both objects are not tied together, because both objects are destroyed independent of each other, and therefore is no interference of their ownership. The log files have showed that the RVP is working as expected.

Passing a shared pointer from C++ to Python

In this scenario we pass a global PyDASH object of shared pointer logged value type and pass a shared pointer of the object to Python. We expect that Python will have shared ownership of the object and the RC works as expected. We implement a function that returns a shared pointer of that object to Python as in the following List. 5.12:

Calling the binding of this function from Python, we get the log List. 5.13.

According to the log, we get a "double free" error in Python. That means that Python tries to free the object two times. This indicates that reference counting on a shared pointer is not working correctly. This might be an implementation

```
/* ... */
static std::shared_ptr<pydash::logged_val> g_object = std::
    make_shared<pydash::logged_val>(1234, 'Y');
std::shared_ptr<pydash::logged_val> give_shared_lv() { return
    g_object; }
/* ... */
/* Binding: */
  m.def("give_shared_lv", &give_shared_lv, "Create and return
    logged_val object");
```

Listing 5.12: Shared Pointer of a Global Object from C++ to Python

```
...
*** Error in python: double free or corruption (out): 0x...0d0 ***
===== Backtrace: =====
....
```

Listing 5.13: Shared Pointer of a Global Object from C++ to Python Log

problem of our binding or of pybind11 itself. As we do not use shared pointers currently for our binding, we do not need to discuss this in detail.

Moving the object from C++ to Python

We have modified PyDASH to instantiate a global object of value type logged_val at start up and and implemented a function `return_global_object_move()` to pass the global object with the RVP attribute `return_value_policy::move`. The implementation is as in List. 5.9 with the RVP attribute `return_value_policy::move` instead of `return_value_policy::copy`.

```
def test_return_global_object_move(x):
  if x:
    print("Entering Scope")
    a = pydash.return_global_object_move()
    print("Leaving Scope")
  print("Left Scope")

test_return_global_object_move(True)
```

Listing 5.14: Python Test of Move of a Global Object from C++ to Python

Using the program in List. 5.14, we get the log (List. 5.15)

As we can see, the object has not been moved from Python to C++, but a copy as been created. This is in contrast to the attribute `return_value_policy::move`. We assume that pybind11 might not recognise the value type logged_val as a moveable type. We can not judge if this due to the compiler or due to pybind11 itself.

```
>>> import pydash
[ LOG |                logged_val | @:0x...a28 -- default construct X
>>> test_return_global_object_move(True)
Entering Scope
[ LOG | logged_val(const self &) | @:0x...c20 -- create copy of @0x
    ...a28: X
[ LOG | logged_val(const self &) | @:0x...c20 -- copied value *-*'
[ LOG |       logged_val(self &&) | @:0x...100 -- move  * <- @0x...c20
    : X
[ LOG |             ~logged_val() | @:0x...c20 -- destroy X and free
    data
Leaving Scope
Left Scope
[ LOG |             ~logged_val() | @:0x...100 -- destroy X and free
    data
>>> exit()
[ LOG |             ~logged_val() | @:0x...a28 -- destroy X and free
    data
```

Listing 5.15: Python Test of Move of a Global Object from C++ to Python Log

At least, the 'source' and the copied object have been decoupled and their destruction was undertaken by Python and C++ respectively. Therefore, we have no interference with ownership and no problematic behaviour, although copying data can decrease the performance.

Results We have seen that only the first of the three scenarios behave as expected. We see that RVP are a useful addition to a Python-binding, but their functionality needs further investigation.

5.3 Passing a Temporary Object from C++ to Python

In this section we evaluate the ownership of a temporary object that was created by a C++ function in PyDASH and is passed from C++ to Python. Our approach of handling this evaluation including the methodology we use is very similar to the previous section, section 5.2.

For the evaluation, we differentiate three scenarios and treat every scenario individually.

Evaluation Aspect We create a temporary PyDASH object as a return object in C++ and pass it over to Python. We consider the ownership of the temporary object before and after the passing.

Initial Situation A temporary PyDASH object is created by a C++ function. After the creation of the object, the ownership of the object is taken by C++

because the C++ constructor obtained the ownership through the allocation of the memory of the object. C++ keeps the ownership at least until the object is passed to Python.

Validation Method We implement a function that (first) creates a temporary PyDASH object with the C++ constructor and that (second) returns the object. We bind this function to Python with pybind11 and annotate the ownership of the returned object using return value policies (RVP) and shared pointer constructs offered by pybind11. As in section 5.2, the PyDASH object that is created by the function is implemented by a custom, logged value type. We write a program that (first) calls the binding of the function from Python that (second) assigns the return object of the function to a local variable and that (third) runs out of the scope of the variable, leaving the object for destruction.

For the destruction, we differentiate two options: (First) If the object is deallocated immediately after it is out-of-scope, the reference counter (RC) must have invoked it's deallocation, because the RC is deterministic and because the C++ side is not aware of the scope of an object on the Python side. (Second) If the objects are destroyed at the end of the Python runtime, C++ has invoked the deconstructor at the end of the runtime. In this case, the RC has not worked properly. Depending on the invoker of the destruction of the object, we can interpret the ownership, because only the owner of an object is subject to destroying it. We use the log that is produced by the value type to trace the flow of the program.

Expected Behavior We expect that the object is passed from C++ to Python with the ownership as stated by the RVP attribute or the shared pointer, respectively.

Evaluation For the evaluation, we differentiate 3 scenarios for passing an object from C++ to Python.

1. Passing a copy from C++ to Python
2. Passing a shared pointer from C++ to Python
3. Moving the object from C++ to Python

Scenarios

We investigate each scenario individually.

Passing a copy from C++ to Python

We implement the before mentioned function to create a logged value type object and bind it to Python with the RVP attribute `return_value_policy::copy` to bind the function as in List. 5.16.

```
/* ... */
pydash::logged_val return_object() { return pydash::logged_val(); }
/* ... */
// Binding:
m.def("return_object",
      &return_object,
      "Create and return logged_val object copy",
      py::return_value_policy::copy);
```

Listing 5.16: Code Snipped of Function returning Global Object with RVP copy attribute

```
def test_return_object(x):
  if x:
    print("Entering Scope")
    a = pydash.return_object()
    print("Leaving Scope")
  print("Left Scope")

test_return_object_copy(True)
```

Listing 5.17: Python Test Program Passing a copy from C++ to Python

We expect that the function passes a copy of the global object to Python and that Python obtains the ownership of the copied object. We use the program in List 5.3 to test this assumption.

As we can see in the log (List. 5.3), no copy was made but the temporary object directly passed to Python. The reference counter deallocated the object immediately after it was out-of-scope.

Passing a shared pointer from C++ to Python

As in section 5.2, we end up in an error message. For details please see this section.

```
>>> test_return_object_copy(True)
Entering Scope
[ LOG |  logged_val | @:0x...540 --- default construct X
[ LOG | logged_val(self &&) | @:0x...3e0 --- move  * <-- @0x...540: X
[ LOG |       ~logged_val() | @:0x...540 --- destroy X and free data
Leaving Scope
Left Scope
[ LOG |    ~logged_val() | @:0x...3e0 --- destroy X and free data
```

Listing 5.18: Log of Python Test Program Passing a copy from C++ to Python

```
>>> test_return_object_move(True)
Entering Scope
[ LOG |           logged_val | @:0x...c70 ooo --- default construct X
[ LOG | logged_val(self &&) | @:0x...350 ((( --- move  * <-- @0x...
    c70: X
[ LOG |       ~logged_val() | @:0x...c70 xxx --- destroy X and free
    data
Leaving Scope
Left Scope
[ LOG |       ~logged_val() | @:0x...350 xxx --- destroy X and free
    data
```

Listing 5.19: Log of Moving the object from C++ to Python

Moving the object from C++ to Python

We implement the function as in List. 5.16, while we annotate the binding with the attribute `return_value_policy::move`. We expect that the function moves the temporary to Python and that Python obtains the ownership of the moved object. We use the program in List 5.10 to test this assumption.

As we can see in the log (List. 5.17), the object was directly assigned (move constructor) after its creation. After it is out-of-scope, the object is destructed. This shows that the object is handled by Python, therefore Python must have gained the ownership.

Results Our observations are similar to the results in the previous section.

5.4 Iterators

Another conceptual difference of Python and C++ are ranges and iterators. Iterators are an important concept of C++ and fundamental for DASH containers. While iterators are available in C++, Python offers ranges. Section 2.2.3 offers a description of the differences. To map those different concepts to each other, pybind11 offers a solution that is easy to implement. In the following code snipped (List. 5.18), we use this solution and implement a binding of a DASH iterator, the DASH array.

For this evaluation, we use the DASH array as an exemplification of an iterator because it is the most important container of DASH. Remember: In DASH, if one is accessing data of value type `T` that is stored in a container, one is returned a proxy reference `GlobRef<T>`. This is displayed in Fig. 2.5 and described in section 2.1.4. To access the value of the global reference, it must be dereferenced. The Python-binding of the DASH array takes this into account. Therefore it is important to emphasize that iterating over a PyDASH array, which is the DASH array that is bound to Python, returns a PyDASH *global reference object* and *not*

```
/* ... */
 py_array.def("__iter__",
        [](dash_array_t & arr) {
          return py::make_iterator<
                    py::return_value_policy::reference_internal,
                    iterator_t, iterator_t, dash::GlobRef<T>
                  >(arr.begin(), arr.end());
        },
        /* ... */
);
/* ... */
```

Listing 5.20: Example Code Snipped of Python-Binding of DASH array as an Iterator using pybind11

the value type that is contained by the array. To access and manipulate the value of the element of the container, one must use special setter and getter methods.

Evaluation Aspect In this section we want to evaluate the mapping of iterators to ranges of the array container of DASH using pybind11 and examine whether it allows us to use PyDASH containers as ranges.

Initial Situation We bind DASH arrays with a specific pybind11 annotation to make them accessible as a range from Python. Cf. List. 5.18.

Validation Method We investigate the evaluation aspect using a Python program. The program is written as in List. 5.19. The program, embed inside the frame of initializing and finalizing the PyDASH environment, collectively instantiates a PyDASH array. Then the array is filled with values depending on the executing unit. After that, we iterate 3 times over the array: (first) to print out the values of the array to compare whether the array is filled with the values stated in the program, (second) to change all values to the same, arbitrary value (in this case, 99), and (third) again to print out the values to examine, whether every element of the array is filled with the same value. Please note that the elements of the array are of type global reference. Therefore, we are using `.get()` and `set(`*value*`)` to get and set, respectively, values to the elements of the array.

Expected Behavior We expect that it is possible to iterate over a PyDASH array, to get and to set its values.

Evaluation Executing the program with 4 units gives us the log (List. 5.22).

The log indicates that the program has set the values of the elements of the container. Unit 0 has iterated over the array using the Python range syntax and

```
import pydash

pydash.initialize(0, "")
myid = pydash.myid().id()
nunits = pydash.nunits()

# Collectively instantiate
array = pydash.ArrayInt(3 * nunits)

# Initialize array:
array[myid * 3 + 0] = 100 * (1 + myid) + 0
array[myid * 3 + 1] = 100 * (1 + myid) + 1
array[myid * 3 + 2] = 100 * (1 + myid) + 2

#Wait for all units:
pydash.barrier()

#Unit 0 prints out, changes and again prints out values of array
if myid == 0:
  for val in array:
    print(val.get())

  for val in array:
    val.set(99)

  for val in array:
    print(val.get())

pydash.finalize()
```

Listing 5.21: Python Program to Test Iteration over a PyDASH array

```
$ mpirun -n 4 python3.5m test_iterator.py
100
101
...
402
99
...
99
```

Listing 5.22: Test of Python Program to Test Iteration over a PyDASH array

printed out the values, changed the values and again printed out the values as described by the program.

Result We evaluated the mapping of an iterator to a range between Python and C++/DASH using pybind11 and build a test case to validate this. We were able to use the print-out created by the program to trace the flow of the program. We have expected that it is possible to iterate over a PyDASH array using range syntax. Our test case show that this is the case. This indicates that pybind11 has mapped the iterator correctly to a range. Therefore we assume that in general it is possible to map iterators from C++/DASH to Python using pybind11.

5.5 Algorithm - Minimum Container Element

We have implemented the DASH algorithm `dash::min_element` as a PyDASH algorithm on an array. The `dash::min_element` algorithm finds the iterator to the global minimal element of a container. In DASH, the interface of the algorithm is `dash::min_element(Array.begin(), Array.end())`. This interface follows the C++ STL definitions. PyDASH is not supposed to imitate the C++ STL, but is aimed at Python developers, and therefore uses the Python standard as a model for the API. This means that algorithms are called from a container, in our case, `Array.min`, because this is *pythonic*.

Our Python-implementation of the algorithm returns the value of the minimal element of the array

Evaluation Aspect We evaluate whether the `dash::min_element` algorithm called from Python finds the element of an array with the minimum value.

Initial Situation An array is created and filled with values. Manually we set the value of an element to be to a minimal value.

Validation Method We investigate the evaluation aspect using a Python program. The program is written as in List. 5.21.

The program, embed inside the frame of initializing and finalizing the PyDASH environment, collectively instantiates a PyDASH array. Then the array is filled with values depending on the executing unit. After the array is initialized, the unit with ID 0 sets the value of the element at the position 2 to 12. After that, the program calls the algorithm on the array. If it returns the same value for the minimal element that was set manually, we assume the implementation works.

```
import pydash

pydash.initialize(0, "")

myid   = pydash.myid().id()
nunits = pydash.nunits()
# Collectively instantiate array:
array  = pydash.ArrayInt(3 * nunits)
# Initialize array:
array[myid * 3 + 0] = 100 * (1 + myid) + 0
array[myid * 3 + 1] = 100 * (1 + myid) + 1
array[myid * 3 + 2] = 100 * (1 + myid) + 2

# Set minimum to be found:
if (myid == 0):
    print("Array size: {}".format(array.size()))
    array[2] = 12

# Wait for all units:
pydash.barrier()

# dash::min_element(a.begin(), a.end())
min_val = array.min(-1)
min_idx = array.argmin()

print("Unit {}: got minimum element {} at index {}"
        .format(myid, min_val, min_idx))

pydash.finalize()
```

Listing 5.23: Python Program to Test Iteration over a PyDASH array

Expected Behavior We expect the the value and position of the minimal element that is found by the algorithm is identical to what was set.

Evaluation Executing the program with 4 units delivers the log (List. 5.22). The algorithm has found the minimal element at its set position as we can see.

Result We evaluated the implementation of a Python-binding of a DASH algorithm using pybind11 and build a test program to validate this. We were able to use the print-out created by the program to trace the flow of the program. We have expected that the algorithm called from Python finds the minimal element.

```
$ mpirun -n 4 python3.5m test_min_algorithm.py
Array size: 12
Unit 0: got minimum element 12 at index 2
Unit 1: got minimum element 12 at index 2
Unit 2: got minimum element 12 at index 2
Unit 3: got minimum element 12 at index 2
```

Listing 5.24: Test of Python Program to Test Iteration over a PyDASH array.

Our test program shows that this is the case. This indicates that the binding works as expected and the implementation is correct. This shows in general that it is possible to bind DASH algorithms to Python and call those algorithms from there.

5.6 Conclusion of Evaluation

We have evaluated in the previous sections of this chapter our Python-binding of DASH. We were able to see that PyDASH provided the results we expected in most cases. Our results show that ownership of objects was clearly decoupled between Python and DASH/C++. Ownership was never connected and therefore there have been no problems of managing the lifetime of a PyDASH object. We have seen, though, that not in all cases an object was moved, but copied. This might lead to a decrease of performance, but it does not cause errors. Only shared ownership is not working as expected, but as it is not required by our binding and being a more "theoretical" case, we doubt that this is a serious problem. The mapping of C++ iterators to Python ranges is working and it is possible to access DASH algorithms from Python. Therefore, we can conclude that it is possible to map conceptual differences of two programming languages and bind a complex C++ Library like DASH to Python.

6 Conclusion and Outlook

In this work, we begun with an characterisation of high performance computing and introduced MPI and PGAS. We gave an overview of the features of the C++ Template Library DASH. Thereafter we presented concepts of the Python and C++ programming languages, showed current usages of Python in the scientific and HPC community and compared the two languages on a conceptual level, especially we looked at iterators and ranges and the lifetime management of an object. Based on the requirements of DASH, we have defined criteria which we used to evaluate and compare tool alternatives that can be used to bind C++ code to Python. We have determined which binding tool satisfies our requirements, pybind11. Using the binding tool pybind11, we showed a reference implementation of C++ code in Python. We presented PyDASH, our official Python Binding of the DASH C++ Library. It offers an easy-to-use toolset for writing HPC applications, bringing together the intuitive syntax of Python with the complexity of the DASH C++ library. Using an example, we showed how to execute a PyDASH program and what happens in the backend of a PyDASH program execution. We treated one difficulty, the lifetime management of PyDASH objects, and presented a solution for it. We validated our implementation PyDASH while we derived the validity of the implementation from use cases with respect to the conceptual differences that have been presented before. The validation showed that it was possible to overcome the conceptual differences between two very different programming languages.

In our thesis, we have shown that it is possible to bring together the best of two worlds: The easiness of use of Python and the expressiveness and efficiency of C++.

We have seen that there is a clear demand for a powerful HPC Python library. Therefore, we believe that our module PyDASH has large potential.

We will focus to provide even more features of DASH in PyDASH and aim to fully implement DASH in the near future.

List of Figures

Bibliography

[1] A. Tate, A. Kamil, A. Dubey, A. Größlinger, B. Chamberlain, B. Goglin, C. Edwards, C. J. Newburn, D. Padua, D. Unat, *et al.*, "Programming abstractions for data locality," research report, PADAL Workshop 2014, April 28–29, Swiss National Supercomputing Center (CSCS), Lugano, Switzerland, Nov. 2014.

[2] H. Zhou, Y. Mhedheb, K. Idrees, C. Glass, J. Gracia, K. Fürlinger, and J. Tao, "DART-MPI: An MPI-based implementation of a PGAS runtime system," in *The 8th International Conference on Partitioned Global Address Space Programming Models (PGAS)*, Oct. 2014.

[3] T. Stitt, *An introduction to the Partitioned Global Address Space (PGAS) programming model*. Connexions, Rice University, 2009.

[4] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: a PGAS extension for C++," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1105–1114, IEEE, 2014.

[5] A. Kamil, Y. Zheng, and K. Yelick, "A local-view array library for partitioned global address space C++ programs," in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, p. 26, ACM, 2014.

[6] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in Chapel: Philosophy and framework," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pp. 12–12, USENIX Association, 2010.

[7] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, D. Iten, and V. Litvinov, "Authoring user-defined domain maps in Chapel," in *CUG 2011*, 2011.

[8] K. Fürlinger, "Exploiting Hierarchical Exascale Hardware using a PGAS Approach," in *Proceedings of the 3rd International Conference on Exascale Applications and Software*, pp. 48–52, University of Edinburgh, 2015.

[9] K. Fürlinger, C. Glass, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedeb, and H. Zhou, "DASH: Data structures and algorithms with support for hierarchical locality," in *Euro-Par 2014 Workshops (Porto, Portugal)*, 2014.

[10] T. Fuchs and K. Fürlinger, "Expressing and exploiting multidimensional locality in DASH," in *Proceedings of the SPPEXA Symposium 2016*, Lecture Notes in Computational Science and Engineering, (Garching, Germany), Jan. 2016. to appear.

[11] K. Fürlinger, T. Fuchs, and R. Kowalewski, "DASH: A C++ PGAS library for distributed data structures and parallel algorithms," *CoRR*, vol. abs/1610.01482, 2016.

[12] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp. 180–186, IEEE, 2010.

[13] T. Fuchs and K. Fürlinger, "A multi-dimensional distributed array abstraction for PGAS," in *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016)*, (Sydney, Australia), pp. 1061–1068, Dec. 2016.

[14] "cppreference.com," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[15] J. Meulemans, T. Ward, and D. Knights, "Syntax and semantics of coding in python," *Hydrocarbon and Lipid Microbiology Protocols: Statistics, Data Analysis, Bioinformatics and Modelling*, pp. 135–154, 2016.

[16] "The python standard library - python 3.6.1 documentation - index," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[17] "Python 2.7.13 - documentation - extending and embedding the python interpreter," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[18] "Python 2.7.13 - documentation - the python tutorial - modules," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[19] W. T. Lavrijsen and A. Dutta, "High-performance python-c++ bindings with pypy and cling," in *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*, pp. 27–35, IEEE Press, 2016.

[20] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi, "Python: the full monty," in *ACM SIGPLAN Notices*, vol. 48, pp. 217–232, ACM, 2013.

[21] "Python 3.6.1 - documentation - garbage collector interface," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[22] G. V. Stefan Van Der Walt, S. Chris Colbert, "The numpy array: a structure for efficient numerical computation," *Computing in Science and Engineering,*

Institute of Electrical and Electronics Engineers,, vol. 13 (2), pp. pp.22–30., 2011.

[23] T. E. Oliphant, *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[24] T. Evans, A. Gómez-Iglesias, and C. Proctor, "Pytacc: Hpc python at the texas advanced computing center," in *Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing*, PyHPC '15, (New York, NY, USA), pp. 4:1–4:7, ACM, 2015.

[25] L. Dalcín, R. Paz, and M. Storti, "Mpi for python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, 2005.

[26] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, "Parallel distributed computing using python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124–1139, 2011.

[27] M. Driscoll, A. Kamil, S. Kamil, Y. Zheng, and K. Yelick, "Pygas: A partitioned global address space extension for python," in *Poster in the PGAS Conference, Citeseer*, 2012.

[28] *ISO/IEC 14882:2014*. 2014.

[29] "C++ standard library header files - cppreference.com," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[30] *ISO/IEC 14882:2003(E) Programming Languages ? C++ §17-27*.

[31] E. D. G. William M. Miller, *A Taxonomy of Expression Value Categories, PL22.16/10-0045 = WG21 N3055*. 2010.

[32] "Value categories - cppreference.com," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[33] B. Stroustrup, *The C++ Programming Language (3rd Edition)*. Addison-Wesley Professional, 1997.

[34] "Move constructors - cppreference.com," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[35] "Reference declaration - cppreference.com," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[36] A. Alexandrescu and H. Sutter, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ In-Depth Series)*. Addison-Wesley Professional, 2004.

[37] "Python 3.6.1 - documentation - 4. built-in types," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[38] "boost - documentation - chapter 1. range 2.0," 2010. [Online; accessed on 01. June 2017, 08:00 a.m.].

[39] E. Niebler and C. Carter, "Working draft, c++ extensions for ranges," *ISO/IEC JTC1/SC22/WG21 document N*, vol. 4620, 2015.

[40] W. Lavrijsen, M. Marino, P. Mato, and J. Generowicz, "Reflection-based python-c++ bindings," 2005.

[41] "Python 3.6.1 - documentation - introduction," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[42] D. Abrahams and R. W. Grosse-Kunstleve, "Building hybrid systems with boost. python," *CC Plus Plus Users Journal*, vol. 21, no. 7, pp. 29–36, 2003.

[43] "http://www.swig.org/doc3.0/swigdocumentation.pdf." [Online; accessed on 01. June 2017, 08:00 a.m.].

[44] "Documentation - 6 swig and c++." [Online; accessed on 01. June 2017, 08:00 a.m.].

[45] T. Jenness, J. Bosch, R. Owen, J. Parejko, J. Sick, J. Swinbank, M. de Val-Borro, G. Dubois-Felsmann, K.-T. Lim, R. H. Lupton, *et al.*, "Investigating interoperability of the lsst data management software stack with astropy," in *SPIE Astronomical Telescopes+ Instrumentation*, pp. 99130G–99130G, International Society for Optics and Photonics, 2016.

[46] "Boost.python - 1.44.0," 2010. [Online; accessed on 01. June 2017, 08:00 a.m.].

[47] P. D. Adams, R. W. Grosse-Kunstleve, L.-W. Hung, T. R. Ioerger, A. J. McCoy, N. W. Moriarty, R. J. Read, J. C. Sacchettini, N. K. Sauter, and T. C. Terwilliger, "Phenix: building new software for automated crystallographic structure determination," *Acta Crystallographica Section D*, vol. 58, pp. 1948–1954, Nov 2002.

[48] "Cffi documentation - cffi 1.10.0 documentation," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[49] J. L. C. Rodríguez, H. Eichhorn, and F. McLean, "poliastro: An astrodynamics library written in python with fortran performance,"

[50] "cppyy: C++ bindings for pypy - pypy documentation," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[51] "First steps - pybind11 2.1.1 documentation," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].

[52] "Functions - pybind11 2.1.1 documentation," 2017. [Online; accessed on 01. June 2017, 08:00 a.m.].