Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Python for High Performance I/O

vorgelegt von

Johannes Coym

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Wirtschaftsinformatik
Matrikelnummer:	6693524
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Prof. Dr. Thomas Ludwig
Betreuer:	Dr. Michael Kuhn

Hamburg, 2019-07-30

Abstract

Python is gaining importance in High Performance Computing (HPC) due to the faster and easier development than in more performance-oriented programming languages like C. This makes Python very attractive for prototyping new ideas, but also in performance aspects Python is getting better with modules like NumPy, SciPy and even optimized versions specific for Intel CPUs. Python even has support to access libraries written in C, to utilize the performance of C, while having the object orientation and ease-of-use of Python.

But although Python is making progress in High Performance Computing, Python has little direct support for High Performance I/O (HPIO) systems. Since the calculation power in HPC systems is growing much faster than the speed of I/O devices, this is an area which is quickly gaining importance.

The goal of this thesis is to evaluate the possibilities of HPIO in Python. Therfore an interface to an existing HPIO system will be implemented using Python's support to access C libraries. The I/O system used for this implementation will be JULEA, which provides a modular design and a object-oriented like implementation in C, which makes the conversion to a real object-oriented language like Python easier. For this implementation, different approaches for the interface between Python and C will be evaluated to ensure a good maintainability and performance.

Contents

1. Introduction **7**
1.1. Motivation . . . 7
1.2. Thesis Goals . . . 8
1.3. Thesis Outline . . . 9

2. Background **10**
2.1. Python . . . 10
2.2. ctypes . . . 10
2.3. CFFI . . . 11
2.4. JULEA . . . 11

3. Related Work **13**
3.1. HPC with Python . . . 13
3.2. Data storage with Python . . . 14

4. Design **15**
4.1. Python interfaces . . . 15
4.1.1. CFFI . . . 15
4.1.2. ctypes . . . 17
4.2. JULEA . . . 18
4.2.1. Key-Value Client . . . 18
4.2.2. Object Client . . . 18
4.2.3. Distributed Object Client . . . 19
4.2.4. Item Client . . . 19
4.2.5. Batches . . . 19

5. Implementation **20**
5.1. Key-Value . . . 20
5.2. Object . . . 21
5.3. Distributed Object . . . 23
5.4. Item . . . 24
5.5. Tests . . . 25
5.6. Benchmarks . . . 26
5.7. Example Application . . . 26

6. Evaluation **28**
6.1. Python Libraries . . . 28
6.2. Key-Value . . . 31

6.3. Object 33
6.4. Distributed Object 35
6.5. Profiling 38

7. Summary, Conclusion and Future Work **42**
7.1. Summary 42
7.2. Conclusion 42
7.3. Future Work 43

Appendices **46**

A. Abbreviations **47**

B. CFFI Code **48**

C. Python Library Tests **53**

List of Figures **57**

List of Listings **58**

List of Tables **59**

1. Introduction

In this chapter, a brief introduction will be provided with a small discussion of the motivation and goals of this thesis. Afterwards there will follow an outline what the remainder of the thesis will consist of.

1.1. Motivation

In High Performance Computing (HPC) there are many programming languages used, since there are other useful libraries for every research field or usecase of an application. These libraries are often in different programming languages, which gives all of these languages their place in HPC. These programming languages also all have different advantages and disadvantages, mostly in the area of complexity, code maintainability and performance. Python is one of these programming languages and is currently gaining more interest since it is easy to use and has several libraries useful for scientific applications.

Python also is currently, according to the Tiobe Index [TIO19], the fastest growing programming language and is not only getting generally more important, since even in HPC Python is getting used more and more. It is an mostly interpreter-based and object-oriented language which helps for quick testing and prototyping since the code does not have to be compiled, which can take time, especially for larger projects.

I/O also plays an important role in HPC systems as most programs need I/O to write results or even checkpoints to permanent storage, which allows the programs to resume jobs at a later time when they are halted or have crashed. Depending on the program this data can consist of only a few bytes or kilobytes for small results, but they can also exceed into the range of terabytes for huge simulation results or checkpoints that need the whole RAM persisted to restore the program run. This is especially important in HPC, since HPC programs usually compute with large amounts of data, which needs to be persisted to access it after the program finishes.

Since there are often so huge amounts of data that have to be written and each type of data has different requirements to be stored effectively, you need different High Performance I/O (HPIO) storage solutions for HPC and the I/O performance is very critical. The I/O performance gets even more important over time, since the computing power is growing significantly faster than the I/O speed, as can be seen in figure 1.1.

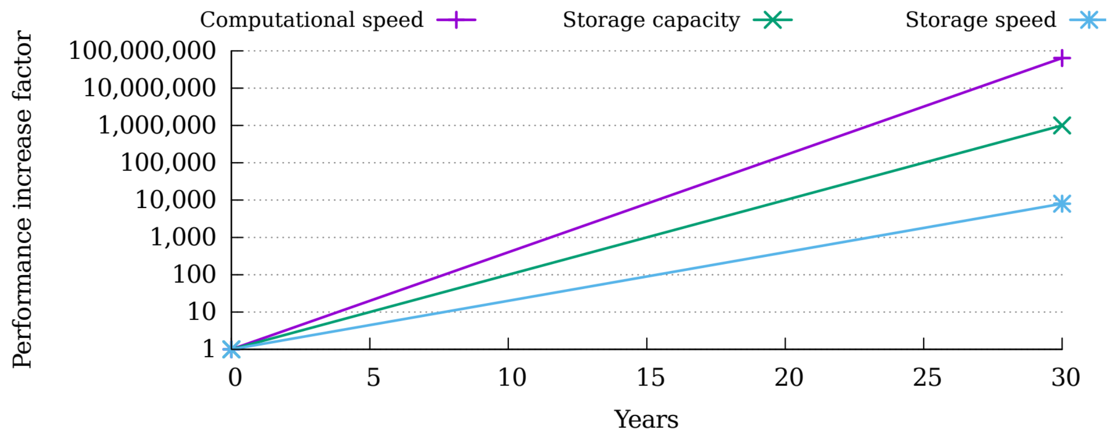

Figure 1.1.: Development of computational and storage speed over 30 years [KKL16, p. 75]

As can be seen in this figure, the storage speed and capacity diverges quickly from the computational speed, which causes the programs to be able to create much more data in the same time, but in this same time frame it gets harder to store all of this data. This can cause huge bottlenecks, which can hold back the speed of these programs, and thats why optimized data storage solutions are needed even more in the future, but there are very little HPIO solutions for Python.

1.2. Thesis Goals

Due to the before mentioned lack of HPIO solutions for Python, the purpose of this thesis is to evaluate the adoption of existing HPIO solutions for usage in Python. The system developed will access the C code of an existing HPIO solution over an Foreign Function Interface (FFI) library. Beforehand will be evaluated which library is suited best for this usecase.

The HPIO solution used will be JULEA [Kuh17], which is well suited for Python, with a modular and already object-oriented like implementation. The implementation will inlude the most important clients to cover most usecases, as well as fitting tests and benchmarks to evaluate the performance compared to the native implementation.

1.3. Thesis Outline

The following chapters will first give some insight to the software and libraries used in chapter 2. Afterwards there will be some focus on the related work in chapter 3, which is followed by chapter 4, which shows the Design of the Python library that is to be built, as well as some insight why this specific FFI implementation was chosen.

Chapter 5 will focus on the implementation of the library, with chapter 6 providing information on the evaluation of the library, as well as the FFI implementations. Chapter 7 then concludes this thesis.

2. Background

In this chapter, some background will be provided on the used programming language with the used libraries. It also gives some information on the storage framework JULEA used for the implementation.

2.1. Python

As mentioned earlier, Python is a rapidly growing language, which is quickly gaining importance in multiple areas, including HPC. Coding in Python is also much simpler than for example in C, but this comes at the cost of performance, since C code can be more than 1000 times faster than plain Python Code, depending on the test [Pug16].

To mitigate these performance issues there are several libraries for Python, which promise improved performance most of the time by passing the operations to better performing C code. Some examples are NumPy, which is a part of SciPy and optimized for scientific calculations, and Cython, which provides a superset language to Python with access to some C functionality, that will then be compiled to increase performance. Two other examples are ctypes and C Foreign Function Interface (CFFI), which provide interfaces to access C code and libraries directly, but they will be in the focus in the next parts.

Due to these performance enhancing libraries with remaining the easy implementations in Python, it is commonly found in HPC by now, which also increases the need for well performing high performance I/O solutions in Python.

2.2. ctypes

Ctypes is a library included in Python, which adds support for accessing C datatypes and function from within Python. To access the functions, ctypes can diretly load the C libraries and is able to cast Python datatypes into C datatypes.

To access the C code, ctypes is using the library libffi, which provides an interface to call functions from other programming language. Therfore ctypes is essentially only a wrapper around libffi to make the access on the C code easier.

It has two main usecases, the first of those is to accelerate your exisiting Python code, by porting the compute expensive code to C and accessing the C funtions from Python. The second one, which will also be used in the implementation later on, is the ability to extend existing C code by an own Python interface for easier access.

2.3. CFFI

CFFI is another library for accessing C datatypes and functions and it is based off ctypes, but has a different approach on several aspects of the implementation. It is also not included with the default Python installation, but it can be easily installed from Python's own module management system pip.

The first of the different approaches is, that CFFI does not use ctypes' `argtypes` and `restype` parameters, which define the datatypes of arguments and return values. Instead it uses a single `cdef` call for a whole library, which has a similar structure as a C include file. Opposed to the `argtypes` and `restype` parameters, this `cdef` call also is not optional and requires definition of any function and non-standard datatype used within all of the functions ans structs. One restriction `cdef` has, is that it allows no kind of directives, which would be very handy for directly importing the header files.

The second difference is that CFFI also brings a so called API mode, which accesses the C code directly at API level, as this mode generates C code based off the `cdef`, which has to be compiled afterwards and can then be imported to Python. CFFI's other mode is the ABI mode, which accesses the library at binary level, therfore it is very similar to the default operation of ctypes, with the required `cdef` being the main difference.

2.4. JULEA

JULEA is a flexible storage framework, which has the goal of simplifying storage systems in HPC with providing client interfaces, that can be directly used within the HPC applications. A key feature of JULEA is the flexibility as it provides several different clients for data and metadata, as well as several backends for storing this data or metadata on different servers. JULEA also provides interfaces for easy addition of new clients or backends, which can directly integrate and provide new features without larger code changes.

This flexibility of the C implementation also makes it suitable for this test of an inclusion of an Python interface, as JULEA currently lacks a bit of language support, with only C and C++ supported, and provides easy access to all functions, that the clients provide. For this implementation all included clients, with the exemption of the HDF5 client, as it works over the HDF5 VOL interface instead of directly over JULEA, will get their own Python interfaces, as well as all internal JULEA functionality needed to run these clients.

This implementation in Python will provide even more flexibility to JULEA, as only two programming languages are supported currently with C/C++. The Python implementation may also benefit more from the flexibility as the backends do not require any new code and all new backends will be supported by the Python implementation without any code changes.

Furthermore this will also provide benefits to new projects using JULEA, as new clients for testing, which are often based on the existing basic clients like key-value (KV) and Object, will be much easier implemented within Python, compared to the C interface.

3. Related Work

In this chapter, some related work in the area of HPC in Python, as well as data storage solutions for Python will be discussed.

3.1. HPC with Python

For High Performance Computing in Python there have already been several trainings with one of them being from Jussi Enkovaara from the Finnish IT Center for Science [Enk13]. This slides mostly give a simple introduction into Python and NumPy, but they also provide a performance comparison for matrix multiplication between Python, C and NumPy, with NumPy being much faster even than C, which is caused to NumPy using optimized math libraries. Furthermore these slides also provide a look into MPI4py, which provides the same bindings to Python as MPI provides to C or C++.

Another training on Python for HPC is by Monte Lunacek by the University of Colorado Boulder [Lun]. This focuses more on parallelization and multiprocessing within Python, comparing builtin libraries like multiprocessing to external libraries like scoop or MPI4py.

There are also research papers for implementations of HPC, two of those will be covered here. The first of those is about Bohrium, which is a framework to easily paralellize NumPy code [Kri+13]. Bohrium allows to run NumPy functions without any code changes parallel on CPUs, GPUs or clusters. As this paper shows, for some applications they reached up to 181 times the performance when they were using a GPU or up to 13.2 times the performance when using a cluster, compared to the native NumPy execution.

The second paper is covering a parallelized Python implementation of the Lattice Boltzman method, which is often used to study fluid dynamics [PG19]. The implementation is done with NumPy and MPI4py and, as the paper shows, it provides less performance than a native C++ implementation, but when scaling with more processes, the gap between the implementations gets smaller until they reach their peak performance. They also highlight the simplicity of coding in Python as their whole implementation only consists of 160 lines of Python code [PG19, p. 131].

Unfortunately they do not provide comparable code in C++ for this specific model, but, in the GitHub repository to the paper [Pas], they also provide a serial variant without MPI. This variant only consists of 32 pure lines of code for the unoptimized Python variant, compared to 98 pure lines of code for the unoptimized C++ variant.

3.2. Data storage with Python

There are also already some data stroage systems, with one of them being H5py [Col]. H5py is also a wrapper around the C functionality of HDF5 and is using Cython to do so. It only supports NumPy at the moment as the write function for datasets just takes NumPy arrays as an input. As HDF5 is completely standardized, the created files in Python can also be read using every other programming language with HDF5 support and vice versa.

Another data storage system for Python is PyMongo [Mon], which gives access to MongoDB, which is also supported as a key-value backend in JULEA. PyMongo uses a native implementation in Python, which removes the necessity of any dependencies, but has some performance flaws. Therfore PyMongo also has an included Python extension module written in C, which can accelerate more compute-bound tasks like the encoding of BSON files.

Both of these systems are a bit limited, as PyMongo only gives you access to a database system, which is mostly usable for metadata, therefore in many cases you would need another system for data blobs. H5py on the other hand only supports NumPy arrays for datasets, which can still be enough in many cases, and it also lacks support for its own VOL plugin interface. This interface is fairly new within release versions of HDF5 and since JULEA also has its own VOL plugin, it would be a nice addition, to have support for this from Python over H5py. Since JULEA also has support for MongoDB, the JULEA Python interface should be a good alternative for PyMongo if you only need MongoDB to store metadata and need a storage solution for data as well.

4. Design

In this chapter, the design decisions of the interface implementation will be discussed. Also this will evaluate which FFI library is best suited for this implementation.

For the implementation in Python it is important to stay close to the Python guidelines, as Python is already providing guidelines for most style and design decisions which are the described in Python Enhanced Proposal (PEP) [War+00] articles in Python's developer's guide. One of the best known PEP articles is PEP 8 [RWC01], which is providing a complete code style guide for Python. This style guide for example sets restrictions on the maximum line length with 79 characters and clarifies that indentations should be made with 4 spaces instead of tabs. There are also tools to check existing code for PEP 8 conformity, which was also used for the implementation of this thesis.

The second important guideline for the design is PEP 328 [Aah03], which defines how imports should be made and, more importantly, how module packages should be designed and accessed. This is important because all the clients and also other internal stuff shall get their own classes, which shall be in a similar structure as the C implementation of JULEA, while the clients itself shall also be easily accessible with `import julea.object` for example. Therfore the __init__.py files, which are always accessed, when the specific package is imported, within each subpackage will manage all the necessary imports, that you can import all necessary classes, with just importing the specific client. Each folder also recieves its own __init__.py, so each client, the lib folder, as well as tests and benchmarks get their own subpackages within the JULEA package.

4.1. Python interfaces

4.1.1. CFFI

The first mode, that was tested, was the API mode. In the first implementation this mode was tested with the manual definition of all datatypes and functions, since CFFI only supports the default datatypes of C. The issue with cdef is, that CFFI does not allow any directives, like `#include`, or opaque datatypes, like GMutex from GLib. Due to the inablity to use includes, every datatype and struct, which is used in any function or struct, has to be defined in the cdef, which causes huge cdefs.

In the following listing there is an excerpt of the cdef used for testing with the key-value pairs from JULEA. This shows the normally opaque union GMutex as well as the definition of `JKV` and `j_kv_new` which then required type definitions for `guint32`, `gint` and `gchar`. The full code can be found in appendix B.

```
1  typedef union _GMutex GMutex;
2  union _GMutex
3  {
4      gpointer p;
5      guint i[2];
6  };
7  typedef struct JKV
8  {
9      guint32 index;
10     gchar* namespace;
11     gchar* key;
12     gint ref_count;
13     ...;
14 } JKV;
15 JKV* j_kv_new(gchar const* namespace, gchar const* key);
```

Listing 4.1: CFFI API Mode

This complexity of the cdef makes not only the implementation far more difficult, but even the maintainability since every new type or struct needs to be defined. The solution to this problem would be to directly load the full header files for the cdefs.

This was also the next variant, which was tested, but it also has the issue that, as already mentioned, no directives are supported within the cdef. Therefore every header file, which has **`#include`**-statements, has either to be preprocessed with the C preprocessor or custom preprocessed within Python [Rig12].

This is already a better variant than the first one, but it also requires to send the code to the preprocessor after every code change in C. The code for this CFFI variant can also be viewed in appedix B, but it would not be functional this way since the files would need to be replaced with the preprocessed ones.

Both of the previous variants also have the issue, that they generate C Code which needs to be compiled before it can even be used. To prevent this, there is the "ABI" mode, which accesses the binaries directly and was the third CFFI test. Although this mode does not require additional compilation, it also suffers from the issue of requiring definition of all datatypes, structs and functions used.

Therefore the cdef, which is also found in appedix B, but is just for the `j_kv_new` function, looks the same as for the first try. The only difference is the loading with `ffi.dlopen`, which is also shown in the following listing.

```
1 from cffi import FFI
2 ffi = FFI()
3
4 lib = ffi.dlopen("libjulea-kv.so")
```

Listing 4.2: CFFI ABI Mode

4.1.2. ctypes

As CFFI is derived from ctypes, its ABI mode is very similar to the usage of ctypes, therfore not much was changed for ctypes. The main difference in the code is that ctypes does not have a `cdef` attribute, but instead using attributes to each C function, which are called `argtypes` and `restype`. These arguments describe the C datatypes of the arguments and the return value of each function, but are not necessarily required as most datatypes get correctly recognized by ctypes.

For JULEA the description of the specific datatypes is not necessary in most cases, but ctypes has the issue, that every time a pointer is returned, it gets automatically converted to a Python int or long, which makes it more difficult to use this pointer as an argument later. Since the pointer is alwas needed on an access of any object from julea, i.e. a JKV, there would be the issue, that this pointer would be converted to a C int, which would break the function call. A solution would be to set the argtypes for every single function so it would convert it correctly to a pointer, but there is an even better solution, as it is possible to create a subclass of the ctypes datatype `c_void_p`, which does not change the class itself but disables the automatic conversion to an integer value.

As a result a similar implementation for `j_kv_new` like in Listing 4.1 would be as following with ctypes, where the pointer to the JKV would be stored in the variable kv.

```
1 import ctypes
2
3 class my_void_p(ctypes.c_void_p):
4     pass
5
6 JULEA_KV = ctypes.CDLL('libjulea-kv.so')
7 JULEA_KV.j_kv_new.restype = my_void_p
8 kv = JULEA_KV.j_kv_new("test", "test")
```

Listing 4.3: Ctypes JKV New Call

Despite the flaws ctypes has, especially with the handling of pointers, it is still much easier to implement with the builtin datatype detection of attributes, which is an advantage against both CFFI modes. Additionally it does not need any additional compilation like CFFI's API mode, which also helps to maintain the code when there are changes in the C code or especially the header files, which need to be in the cdef. Due to the lack of header code in the Python code, ctypes code is also much easier to expand when new functionality needs to be implemented.

As will be shown in chapter 6 ctypes also has a very similar performance to CFFI, which is not surprising as they use a similar approach, and therfore it is better suited for this implementation and will therfore be the approach used for the full implementation later on.

4.2. JULEA

4.2.1. Key-Value Client

The first of the clients to implement is the key-value client, as most of the testing between the Python libraries was also done with this client. This client is mostly meant for metadata and therfore it currently only has several databases as possible backends.

To design this client within Python, The focus is on Python's object-oriented part, as JULEA already has an implementation very similar to object-oriented classes. Therefore this client should be handled as an own class named `JKV` with the constructor replacing the `j_kv_new` function from C. Therefore each key-value pair used, shall be an object of the type JKV.

Since the data also needs to be handled already within Python, there is also a need for buffers which can handle any type of data, bytestrings are very fitting, since they can handle all types of data in binary form.

4.2.2. Object Client

The second client is the object client, which is the basic client for storing larger data. The object side of the backends has more classic data storage backends like POSIX, but also modern object stores, like RADOS for example.

The design of the implementation is very similar to the one of the key-value client, with the main difference being a seperate create and write function, while the key-value client has a put function, which serves the functionality of both.

4.2.3. Distributed Object Client

As the object client writes the corresponding data to the object only to a single server within the JULEA data storage servers, there is also the distributed object client. It distributes each data object between several data servers to utilize the combined I/O speed of all of them.

The design of the client itself is nearly identical to the one of the object client, but it also requires a distribution, therfore there has to be another class for the distribution. This class also requires an enum in C, so the enum also has to be converted to be usable in Python.

4.2.4. Item Client

The item client is a bit more complex as it combines the distributed object client and the key-value client. It uses the key-value pair to store metadata to the object and since it uses an distributed object, the before mentioned class for the distribution is also needed for this client.

While the item client also has a similar syntax to both of the object clients, it also needs another additional class, the collection. Collections are a little similar as folders, as each item is in a collection and a collection can contain any amount of items, which can be sorted using the collections.

With the item implementation, there is also another difference, since there is no directly accessible new function, you have to either directly create a new item or get an existing item. Therfore the constructor has to differentiate between create and get for the initialization.

4.2.5. Batches

As JULEA also has batches for all operations, that require to be executed on the hard drive and not only in the program, these are required for the Python implementation as well. These batches can contain an arbitrary amount of operations and the current set of operations can be executed at any time.

The design for this is a very simple class as this is mostly a simple wrapper, but it needs a function providing the direct C pointer to the batch, as the functions of the other classes need this to add their operations. Also, like with the distribution, it needs an enum of the different semantics, for example if the data has to be directly persisted to the hard drive or if it can be cached first.

5. Implementation

In this chapter, the implementation of the different clients and necessary additional classes is explained. Additionally the implementation of the according tests and benchmarks will also be described and an example application will be provided to compare the programming languages.

5.1. Key-Value

The key-vlaue client gets its own class named JKV. Within the constructor only a simple call to the function `j_kv_new` of the C interface is needed, where the returned pointer is required to be stored in a variable of the class JKV. As discussed earlier in chapter 4, to correctly store this pointer the restype of the function needs to be set to the type of our previously created subclass `my_void_p`. As can be seen in the following listing, the strings that are required as namespace and key are taken as attributes from the constructor, but they have to be encoded into bytestrings, so they are correctly recognized by ctypes as strings, otherwise the C interface only receives the first character.

```
1 def __init__(self, namespace, key):
2     enc_namespace = namespace.encode('utf-8')
3     enc_key = key.encode('utf-8')
4     self.kv = JULEA_KV.j_kv_new(enc_namespace, enc_key)
```

Listing 5.1: JKV Python Constructor

While the destructor is the simplest possible wrapper function for `j_kv_unref`, which is needed to also release the memory in C, to avoid memory leaks, the `delete` function also looks like a simple wrapper, as can be seen in listing 5.2, but it has one additional function call. Since the C code asks for the pointer to the batch, the batch can not just directly be passed, as an instance of the Python class JBatch is passed, so the function `get_pointer` is needed, that returns the pointer, which is actually usable in C.

```
1 def delete(self, batch):
2     JULEA_KV.j_kv_delete(self.kv, batch.get_pointer())
```

Listing 5.2: JKV Python Delete Function

More interesting are the put and get functions, as these need C objects, which have to be created from within Python. For the put function in the first place, as can be seen in listing 5.3, an unsigned long has to be created, where the length of the value, that should be written, is stored, and also add one to the length, since the bytestrings used also need a null terminator. Afterwards a string buffer from ctypes is created and the bytestring is passed as the raw data, so it behaves like a normal pointer to a buffer in C.

```
1 def put(self, value, batch):
2     length = ctypes.c_ulong(len(value))
3     value2 = ctypes.create_string_buffer(length.value)
4     value2.raw = value
5     JULEA_KV.j_kv_put(self.kv, ctypes.byref(value2),
       ↪ length, None, batch.get_pointer())
```

Listing 5.3: JKV Python Put Function

The get function on the other hand only needs an empty char pointer to recieve the bytestring, as well as an unsigned long with the default value of 0. After the function is called, these values are also both returned as a tuple, so both can be accessed afterwards. The important part with the returned values is, that both have still to be the ctypes datatypes themselves and not just the values in them, since the values in them only get changed after the batch is executed. Therefore to access the data after the batch is executed, it is also important to access with value.value[:length.value] or length.value to access the stored data. The length parameter is required for the value, since the data is accessed as a char array and the data is not null terminated, so the length of the read data is one byte longer than the length of the data itself.

```
1 def get(self, batch):
2     value = ctypes.c_char_p()
3     length = ctypes.c_ulong(0)
4     JULEA_KV.j_kv_get(self.kv, ctypes.byref(value),
       ↪ ctypes.byref(length), batch.get_pointer())
5     return value, length
```

Listing 5.4: JKV Python Get Function

5.2. Object

The object client is also implemented in an own class named JObject. While the constructor, destructor and delete function are exactly the same as for the key-value client, there are also several new functions, that are different to the key-value client. The first one of those functions is the create function, which is a simple wrapper function just as the delete function.

It gets more interesting going on to the `write` function. As already with the quite similar `put` function from JKV, we first have to create a C variable for the length, only this time it is an unsigned 64-bit integer, as we can work with bigger data for the object client. The buffer created is the same as with the key-value client, but new is another unsigned 64-bit integer, which is returned afterwards and is providing the number of bytes written. As with the return types we had before, the number has to be accessed after the batch is executed with `bytes_written.value` to get the correct value.

```
1 def write(self, value, offset, batch):
2     length = ctypes.c_ulonglong(len(value))
3     value2 = ctypes.create_string_buffer(length.value)
4     value2.raw = value
5     bytes_written = ctypes.c_ulonglong(0)
6     JULEA_OBJECT.j_object_write(self.object,
       ↪ ctypes.byref(value2), length, offset,
       ↪ ctypes.byref(bytes_written), batch.get_pointer())
7     return bytes_written
```

Listing 5.5: JObject Python Write Function

The `read` function does not differentiate itself much from the `get` function from JKV, as that already had a second return value for the length, this simply gets replaced by `bytes_read`, as the length to be read has to be provided in the function arguments. The only real difference is again, as with the `write` function, that the C variable has to be a 64-bit integer instead of a 32-bit one.

The last remaining function of JObject, `status`, is also interesting, as it only returns two numbers, the modification time and the size of the object. Those values also require creation of the fitting C variables beforehand, therefore the modification time needs a signed 64-bit integer and the size needs an unsigned 64-bit integer. Both of them are then returned directly, as like read and write, the `status` function also requires, that the batch is executed first, before the values are accessed.

```
1 def status(self, batch):
2     modification_time = ctypes.c_longlong(0)
3     size = ctypes.c_ulonglong(0)
4     JULEA_OBJECT.j_object_status(
5         self.object, ctypes.byref(modification_time),
6         ctypes.byref(size), batch.get_pointer())
7     return modification_time, size
```

Listing 5.6: JObject Python Status Function

5.3. Distributed Object

The distributed object client is very similar to the object client. The only bigger difference is the addition of a distribution, which gets its own class JDistribution and needs to be provided in the constructor as it has to be passed to the C interface to determine how the data is distributed across the servers.

As the distribution is created using one of three options stored in an enum in C, this enum has to be mapped to the according integer values within Python. The constructor then only takes a single input, but, as shown in listing 5.7 checks if the input is an integer, if it is, then it creates a new distribution from that, otherwise it assumes the second possible way to get a distribution, which is from a BSON datatype. To get the according BSON, which has to be a C pointer to a BSON datatype, the only way is with the `serialize` function of the distribution, which is therfore also implemented and only a wrapper for the serialize C function. The destructor is also a simple wrapper like for the other classes and as the distribution has to be passed to the distributed object, it also has a `get_pointer` function, just like the before mentioned one of JBatch.

```
1 def __init__(self, input):
2     if isinstance(input, int):
3         self.distribution = JULEA.j_distribution_new(input)
4     else:
5         self.distribution =
            ↪ JULEA.j_distribution_new_from_bson(input)
```

Listing 5.7: JDistribution Python Constructor

5.4. Item

As the item client builds directly on top of the distributed object client, its implementation is also very similar to the distributed object, and therefore also the object client. It differentiates itself from the before mentioned clients only in the constructor, which is far more complex than for the other clients, and in the need for a collection, which is exclusive to this client.

The collection is very similar to the distribution, except there is no enum and instead of being able to generate it again from a BSON datatype, it has its own `get` function. Therefore the constructor also needs more inputs as it needs a name, a batch and a boolean variable, which defines if the constructor shall create a new collection or if it should just get an existing collection, as shown in listing 5.8.

```
1 def __init__(self, name, batch, get):
2         enc_name = name.encode('utf-8')
3         if get:
4             self.collection = my_void_p
5             JULEA_ITEM.j_collection_get(ctypes.byref(
                ↪ self.collection), enc_name,
                ↪ batch.get_pointer())
6         else:
7             self.collection =
                ↪ JULEA_ITEM.j_collection_create(enc_name,
                ↪ batch.get_pointer())
```

Listing 5.8: JCollection Python Constructor

This collection is then used in the constuctor of the JItem class, which also features a boolean variable as input, since the item client also has a `get` function from the C interface which is used in the constructor as an alternative to creating a new item. The distribution within the input variables is also special, as it has `None` as a default value. This is due to the fact, that it is possible to just give the C interface `NULL` as a distribution, which causes it to just use the round robin distribution. Therfore it also has to be checked before the function call if the `distribution` is `None`, as then `None` just has to be passed, which is automatically converted into `NULL`. Otherwise the pointer of the distribution has to be provided to the C interface.

In the case no new item has to be created and instead an existing item shall be retrieved, then we also do not need the distribution, but instead we first have to generate an empty pointer as the get function has no return value because it only gives a pointer after the batch is executed. Therfore in that case, instead of the distribution, a pointer to the pointer where the item is stored is passed onto the `get` function.

```
1  def __init__(self, collection, name, batch, get,
     ↪ distribution=None):
2      enc_name = name.encode('utf-8')
3      if get:
4          self.item = my_void_p(0)
5          JULEA_ITEM.j_item_get(collection.get_pointer(),
             ↪ ctypes.byref(self.item), enc_name,
             ↪ batch.get_pointer())
6      else:
7          if distribution is None:
8              self.item = JULEA_ITEM.j_item_create(
                 ↪ collection.get_pointer(), enc_name,
                 ↪ distribution, batch.get_pointer())
9          else:
10             self.item = JULEA_ITEM.j_item_create(
                 ↪ collection.get_pointer(), enc_name,
                 ↪ distribution.get_pointer(),
                 ↪ batch.get_pointer())
```

Listing 5.9: JItem Python Constructor

5.5. Tests

The tests for the Python interface are mostly designed to test correct executions of the calls that are passed to the C interface. Therfore they are not designed to test as thoroughly for all edge cases, as the C tests already do that part, although they are designed in close relation to their C counterparts. For the formatting of the output an extra function was created which formats the output exactly like the C tests.

The tests also are only for the clients itself, as running them already requires all other classes, which are implemented within the Python interface. The key-value client gets two tests with the first one just testing a put and a delete on the object and the second one also testing a get call between the put and delete. In the get test it also asserts that the correct value and length are returned to assure correct execution.

For both of the object clients and the item client, there are three tests each and they are very similar between all of those clients, with the only exception being collections and distributions. The first test for those clients is only creating and deleting an object or item. The second one gets more advanced as it then writes to the object, reads from it and checks that the read value and length is correct. In the last test the status function also gets tested, that it returns a real modification time and the correct size of the written data.

5.6. Benchmarks

The benchmarks are also built in a very close relation to their C counterparts, as they need to provide comparable results. Therefore all clients got most of the benchmarks of their C counterparts, also with the different batch usages. The formatting of those benchmarks is then again handled by an own function only responsible for formatting the output just like the C benchmarks.

The time measurement is done using Python's own time function, which gives us the current time in millisecond precision, so we have the same precision as the output of the C benchmarks. Also all batches in the benchmarks are used over with-statements, that automatically execute the batch at the end of the statement, therefore saving an extra execute call, but have no performance impact. Each benchmark also has a single function for the different batch variants and the read and write benchmarks also support different block sizes, just like the C variants. The number of iterations is also the same, so the execution times are also comparable, not only the iterations per second.

The benchmarks then have an own function, which calls all the benchmarks in the correct order and prints the formatted output. These functions are then imported by the benchmark script, which is also a Python file but it lays outside of the package itself. This script then first starts the JULEA server, executes the benchmarks afterwards and stops the server at the end.

5.7. Example Application

Since JULEA already includes an example application, consisting of a simple write and read to an object, this application was also implemented in Python to have a comparison between those interfaces. As can be seen in the listings below, the first difference is already in the import statement. With the Python interface you can directly import all classes needed from the client, while in C the header files for JULEA and the object client are loaded.

Another difference is the usage of batches, as we are using with-statements here as well(e.g. line 8 in listing 5.11), to save an extra execute call. Furthermore the Python implementation also does not need an own buffer for the read call, as the read call creates its own buffer, which causes again the need to give the bytes written as a length opposed to just reading 128 bytes.

```
1  #include <julea.h>
2  #include <julea-object.h>
3
4  #include <stdio.h>
5
6  int
7  main (int argc, char** argv)
8  {
9    (void)argc;
10   (void)argv;
11
12   g_autoptr(JBatch) batch = NULL;
13   g_autoptr(JObject) object = NULL;
14
15   gchar buffer[128];
16   gchar const* hello_world =
       ↪ "Hello World!";
17   guint64 nbytes;
18
19   batch =
       ↪ j_batch_new_for_template(
       ↪ J_SEMANTICS_TEMPLATE_DEFAULT);
20   object = j_object_new( "hello",
       ↪ "world");
21
22   j_object_create( object, batch);
23   j_object_write( object,
       ↪ hello_world,
       ↪ strlen(hello_world), 0,
       ↪ &nbytes, batch);
24   j_object_read(object, buffer,
       ↪ 128, 0, &nbytes, batch);
25   j_batch_execute(batch);
26
27   printf("Object contains: %s (%lu
       ↪ bytes)\n", buffer, nbytes);
28
29   return 0;
30 }
```

Listing 5.10: JULEA C Hello World Application [Kuh19]

```
1  from julea.object import JObject,
     ↪ JBatch,
     ↪ J_SEMANTICS_TEMPLATE_DEFAULT
2
3
4  def main():
5    object = JObject("hello",
       ↪ "world")
6    hello_world = "Hello World!"
7
8    with JBatch(
       ↪ J_SEMANTICS_TEMPLATE_DEFAULT)
       ↪ as batch:
9      object.create(batch)
10     bw = object.write(
         ↪ hello_world.encode('utf-8'),
         ↪ 0, batch)
11
12   with JBatch(
       ↪ J_SEMANTICS_TEMPLATE_DEFAULT)
       ↪ as batch:
13     out = object.read(bw.value, 0,
         ↪ batch)
14
15   print("Object contains: {buffer}
       ↪ ({length} bytes)".format(
       ↪ buffer =
       ↪ out[0].raw.decode('utf-8'),
       ↪ length = out[1].value))
16
17
18 if __name__ == "__main__":
19   main()
```

Listing 5.11: Python Hello World Application

6. Evaluation

In this chapter, the performance of the Python libraries will be compared and the overhead of the Python implementation will be compared to the C implementation. Additionally some functions will be profiled to see where they lose performance against the C implementation.

6.1. Python Libraries

To test the Python libraries a C program was created, which consists of 2 test functions and a `main` function to run the code in native C. Thefore the code was also compiled in 2 variants, one as a C program and one as a shared library, so CFFI and ctypes could access the functions. As one of the functions also uses plain function calls, the code was also compiled without any compiler optimizations to prevent the compiler from optimizing it away. The full code can also be seen in appendix C.

These tests were run on a personal computer with an Intel Core i5-8250U and 8 GB DDR4 memory running on a 5.1.18 Linux Kernel with GCC 9.1.1 and Python 3.7.3. All tests were run 10 times, except the native Python and NumPy variants, which were only run 3 times as they had very long runtimes and are clearly at their own levels of performance. For each library or language tested a standard deviation was calculated additionally to the average runtime, as some runtimes are quite close together.

The first test function is `calc`, which is just making a basic mathematic calculation on an integer using multiplication, addition and division in a loop. This function is then called 10,000 times with an iterated integer passed to emulate small data transfers to the function.

```
1 int calc(int j) {
2     unsigned long num = j;
3     for (int i = 1; i < 50000; i++) {
4         num *= i;
5         num += 1;
6         num = num / i;
7     }
8     return num;
9 }
```

Listing 6.1: calc function

As table 6.1 shows, ctypes and the API mode of CFFI provide very comparable performance to the C code, as they are only 1.5 and 4 hundreds of a second slower for the 10000 iterations. Interesting hereby is that ctypes has less overhead than CFFI, which is also supported by the standard deviation, that is much smaller than the difference between ctypes and CFFI. Additionally the implementation of the function in plain Python is more than 10 times slower than the libraries, with NumPy even being much slower than Python itself.

This might be surprising as NumPy should be accelerating calculations in Python, but as shown in [Fow16], NumPy is slow for small numbers of elements, because it is optimized for large datasets. Therfore with the number of elements growing, Python's performance is quickly beaten by NumPy, but this case is not considered in this test, as we only work with single elements.

Language/Library	Average runtime	Standard deviation
C	7.0612 s	0.0065
ctypes	7.0751 s	0.0057
CFFI API Mode	7.1012 s	0.0063
Python	100.19 s	2.60
NumPy	2233.1 s	45.2

Table 6.1.: Performance results using calc function

The second test function is `test_call`, which is a function that just returns 1 to emulate the pure performance impact by a function call itself. To get better measurable execution times, this function is then called 1,000,000 times and this time only in C, ctypes and CFFI.

```
1 int test_call() {
2   return 1;
3 }
```

Listing 6.2: test_call function

This test obviously provides a huge performance advantage, as it should primarily show the overhead of plain function calls from Python. When comparing the results to the first test function, the results in table 6.2 show a different order of ctypes and CFFI, with CFFI this time being faster.

Language/Library	Average runtime	Standard deviation
C	0.0023 s	0.0000172
ctypes	0.2702 s	0.0064
CFFI API Mode	0.2389 s	0.0064

Table 6.2.: Performance results using test_call function

Therfore these tests are showing that ctypes and CFFI provide a huge performance advantage over pure Python, depending on the function, the performance even is not far off the performance of pure C code. Also ctypes and CFFI perform slightly different depending on the test case, but in both cases the differences were not significant enough to show that one of them clearly performs better, in the end it is all dependent on the usecase.

6.2. Key-Value

As JULEA is targeted for server usage in an HPC system, all of the client benchmarks were run on a dual-socket server with two Intel Xeon X5650, 12GB of DDR3 memory and a single HDD running on an 4.15.0 Kernel with Python 3.6.8. All benchmarks were run three times and an average was taken for creation of the diagrams.

The first benchmark is the null backend of the key-value client, which was run on the client directly, therefore it is suited well to measure the pure overhead of the Python interface. As figure 6.1 shows, the C benchmarks all have quite similar results, while the put benchmark has the largest overhead in the Python interface. The least overhead is with the delete functions, while the get functions sit right betweent delete and put.

There are two things standing out with this benchmark. The first is that every batch variant ob the benchmarks has significantly less overhead than the normal variants, since there are far fewer function calls because the batch is only executed once. Secondly it seems to be relevant if data is transferred between Python and C over function attributes and in which direction they are transferred. Since the put function creates its own C buffer which is passed to the C interface and the get function only creates a pointer from which the data is later accessible, it seems like the data transfer from C to Python creates less overhead.

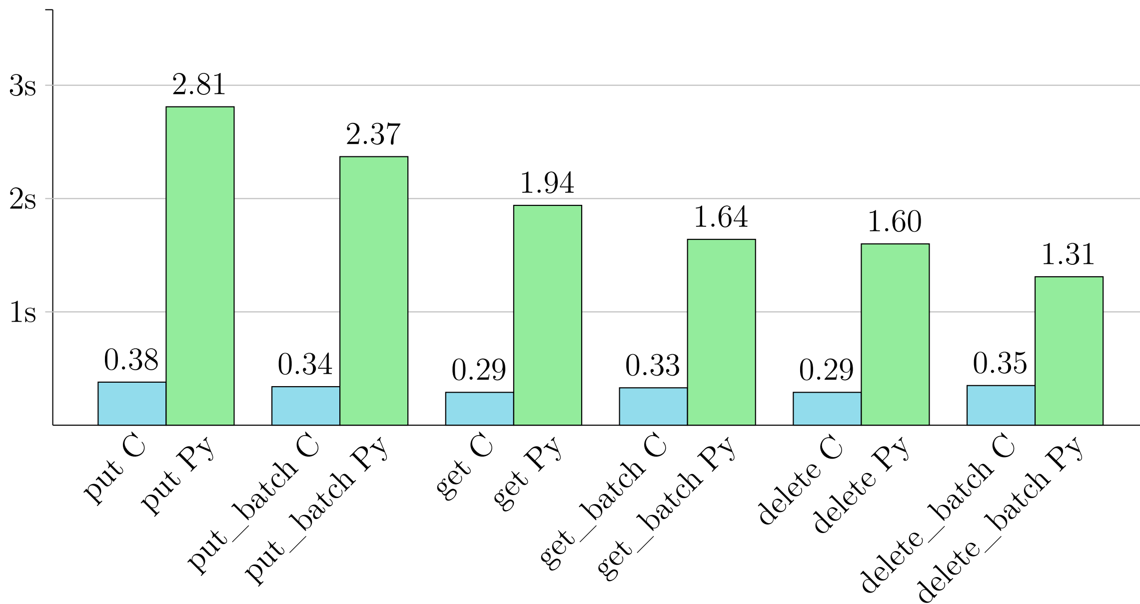

Figure 6.1.: Key-Value Null Client Performance

The second key-value benchmark is using the leveldb backend in server mode, which should replicate a more common usage mode as it also sends the data to the server and persists the data in contrast to the null backend that just discards the data. Again figure 6.2 shows a huge difference of the overhead between the normal and batch variants, with the batch variants having an overhead between 7 and 17 percent, while the normal variants have an overhead of 37 to 50 percent due to the higher amount of function calls.

Also the trend, that the overhead on the C variant is the largest for the put function and smallest for the delete function, continues, but with only a small difference between the put and get overhead. Due to the very good performance of the batch variants in Python, they seem very well suited for real implementations without a big performance hit. This also shows that implementations using this interface should also try to keep their batch executions as low as possible since each function call to C increases the overhead.

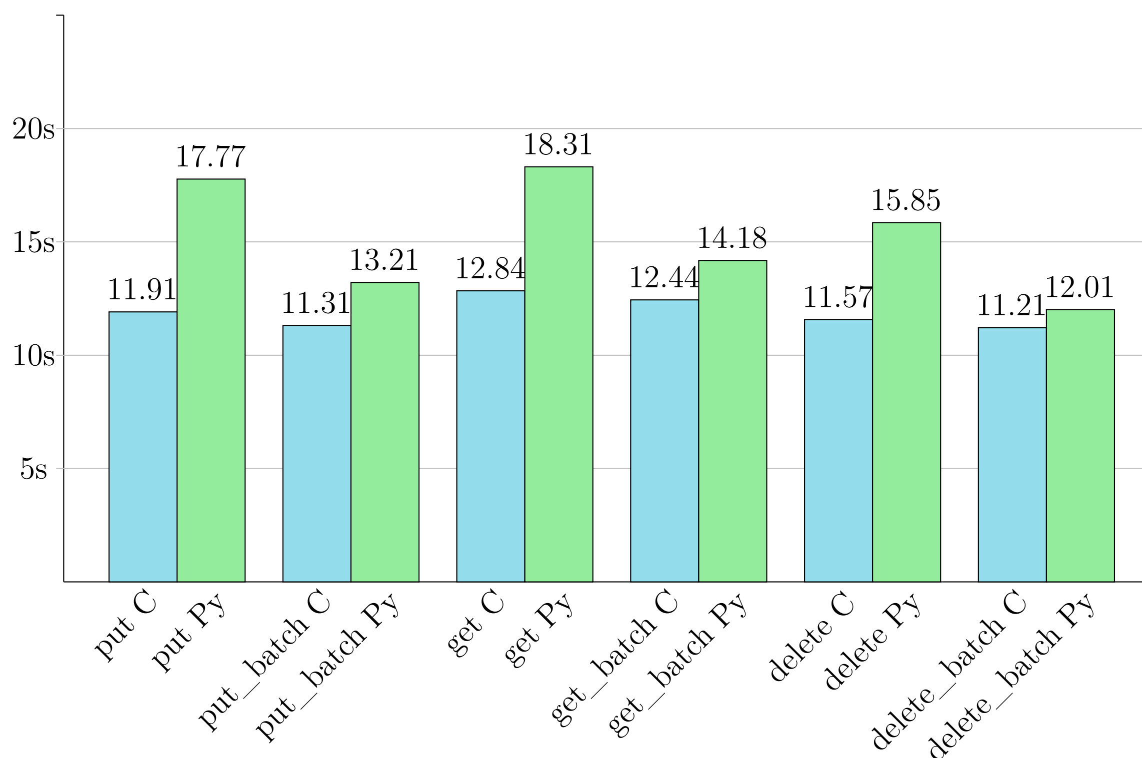

Figure 6.2.: Key-Value leveldb Server Performance

6.3. Object

For the object client the first test again is with the null backend in client mode to measure the pure function overhead. The results in figure 6.3 show very similar results as for the key-value client, with the functions with very little or no data transfers being considerably faster than the read and write functions. Also just like before, all batch variants are faster than their normal counterparts, because of the lower number of function calls.

For the C implementation the benchmarks again all ran for a very similar time, with the batch variants of read and write being the exception, as they are a bit faster, which isn't as clear for the other benchmarks, as for delete the batch variant is even slower than the normal one. The least overhead of the Python benchmarks has the batch variant of the status benchmark with 0.42 seconds overhead, which is still nearly triple the runtime of the C counterpart.

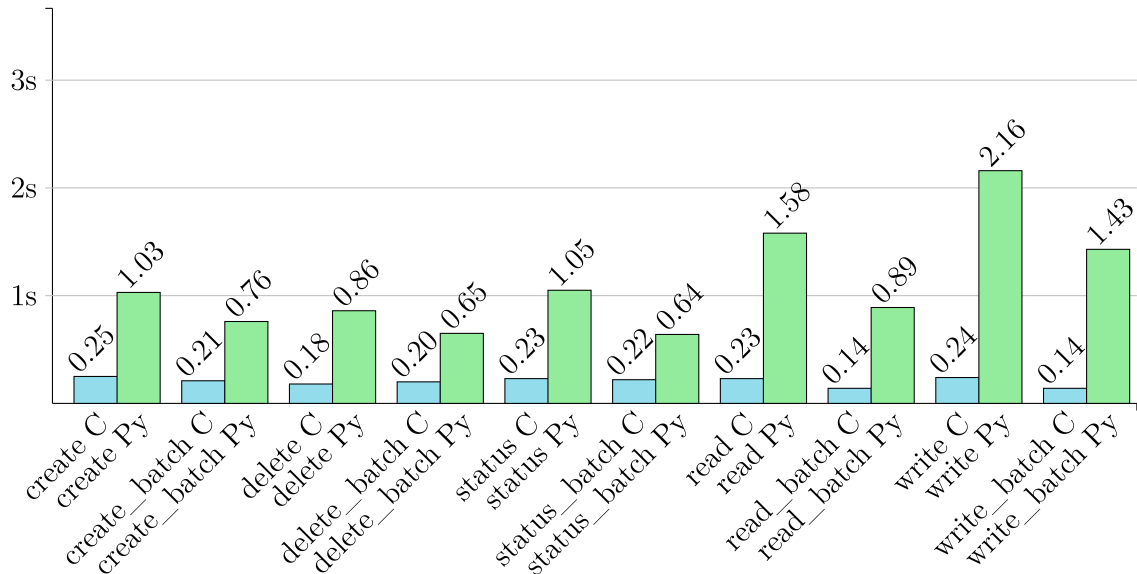

Figure 6.3.: Object Null Client Performance

The next run was using the POSIX backend in client mode, which already really writes data, but does not communicate with a server. This benchmark run showed very mixed results for the different functions, with some going pretty well and others even getting worse. For create, delete and status the results already look pretty good with just 14 to 35 percent overhead for their batch variants and 54 to 105 percent overhead for the normal variants.

When going to read and write then, the relative overhead quickly increased to up to 218 percent for the batch variant of the read variant. These show even larger absolute performance gaps for these functions than the null backend, with only the batch variants having a similar amount of overhead.

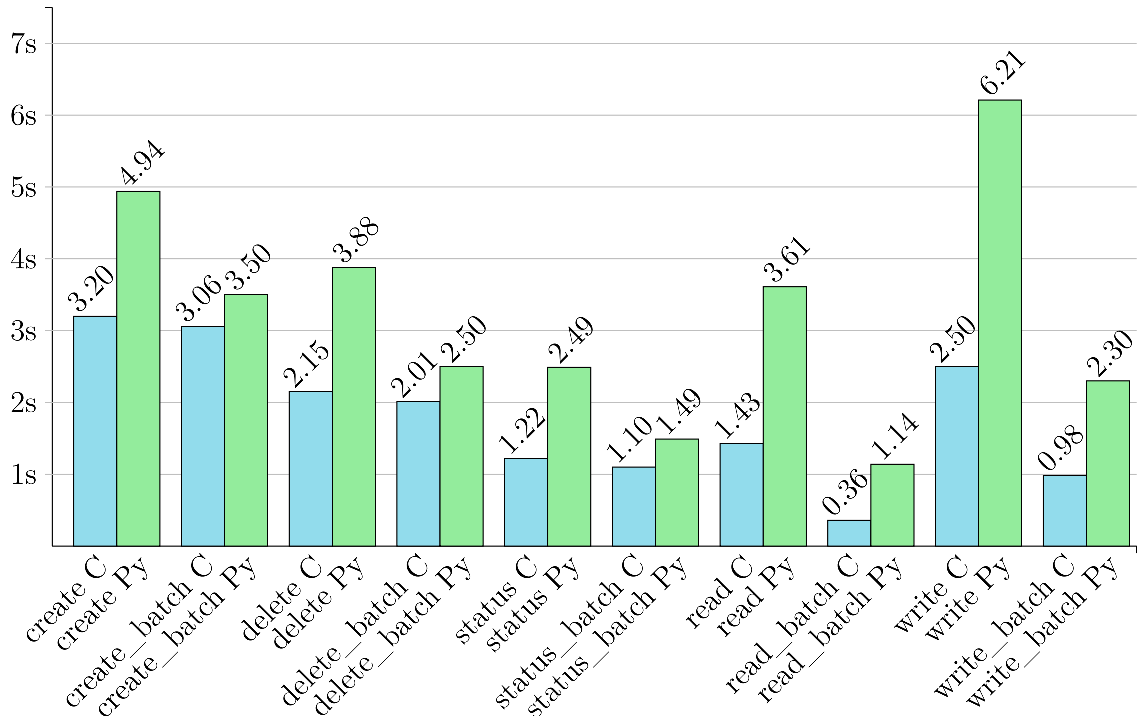

Figure 6.4.: Object POSIX Client Performance

As the client mode of the POSIX backend also measures more of an overhead, as it works directly within the C interface, the same banchmark was also run in server mode. Because in server mode the data is sent to the server process, this reduces the performance of all functions, while the overhead of the Python interface stays the same and therefore the relative overhead gets reduced significantly.

While this is true for all normal variants with 8 to 32 percent overhead, the Python variant even reduces the overhead, compared to the client mode, for all batch variants. It also seems that the server mode of the POSIX backend really suits the Python backend, as three of the batch variants are even faster with the server mode than with the client mode. These small amounts of overhead of 1 to 16 percent also show again with how little overhead this can be used, if it is well optimized.

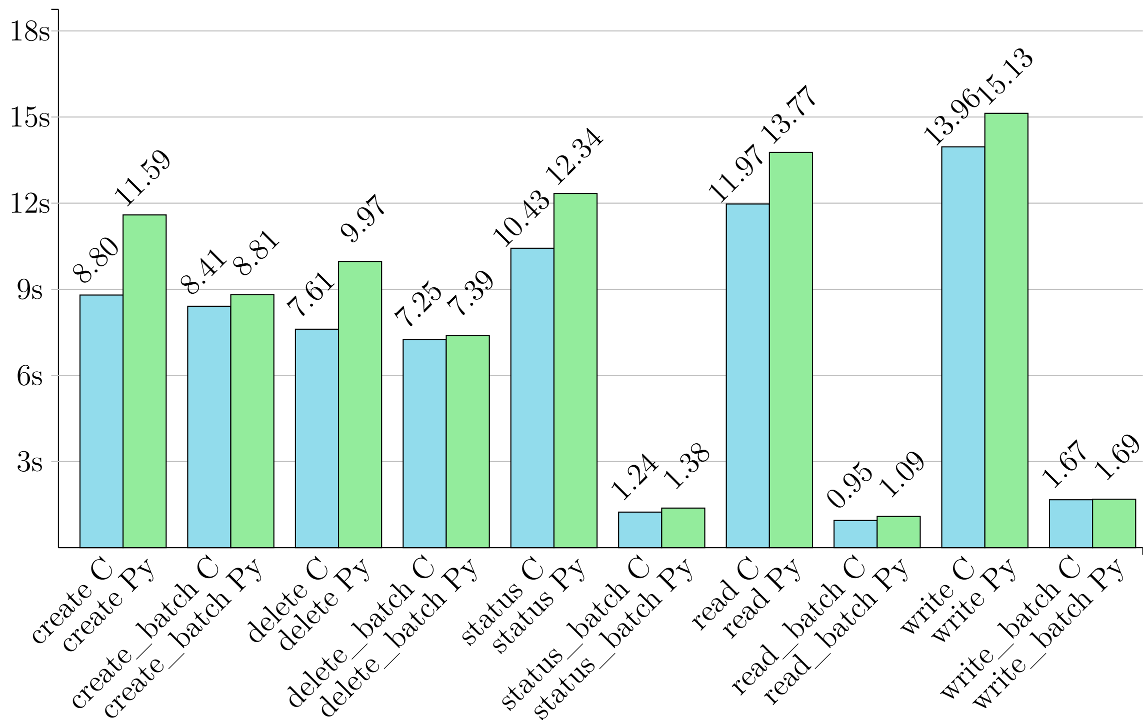

Figure 6.5.: Object POSIX Server Performance

6.4. Distributed Object

For the distibuted object client the same benchmarks were run as for the object client, but, due to the different number of iterations, the results were quite different. In figure 6.6 For the null backend in client mode, there is a quite noticable peak for the batch variant of the create function, which is caused by using 100,000 instead of 1,000 iterations just for the batch variant.

Despite this anomaly due to the higher number of iterations, again read and write have the highest overhead due to the data transfers, with write again taking longer than read in Python while they perform exactly the same in C. Additionally, like before, most of the batch variants perform better than their normal variants in Python, with the only exception being the status function, which is not really representative considering the very low execution time with the relatively low accuracy of only three decimal places.

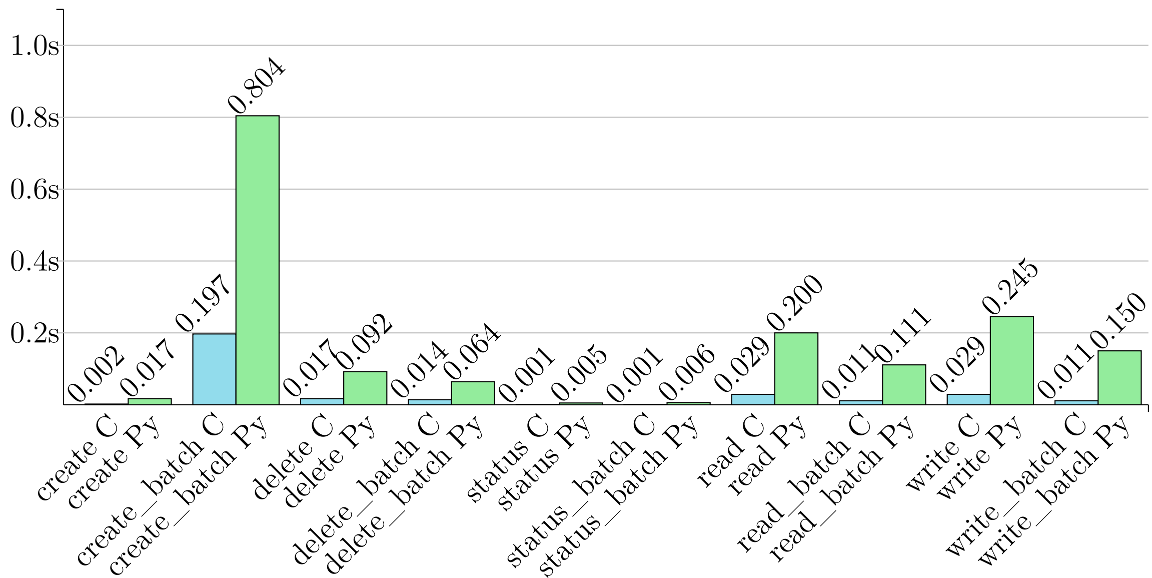

Figure 6.6.: Distributed Object Null Client Performance

The client mode for the object backend in figure 6.7 shows quite similar results as the null backend previously with most of the absolute overheads more or less the same. The batch variants also remained faster, except for the status benchmark, which still stays a little bit slower, and the create benchmark, which takes longer but is in fact faster if you divde its time by 100 to get the same number of iterations.

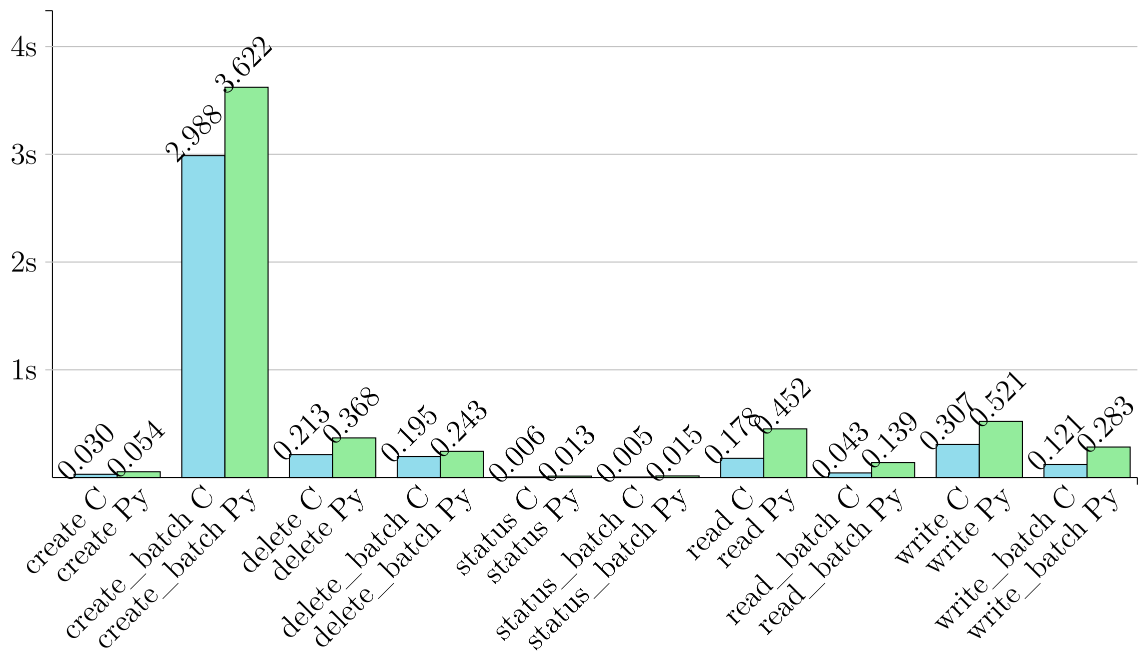

Figure 6.7.: Distributed Object POSIX Client Performance

To also provide results of a more realistic usecase, this benchmark was also run with the POSIX backend in server mode. The results of this were very similar to the ones of the object client, as the overhead was very little compared to the C interface, although it got a bit unusual, as some benchmarks ran faster in Python than in C.

The performance gap was very small in the cases when the Python interface was faster, with at most 21 ms lead for the Python interface, but it is still an unusual result. Additionally, just like with the object client, the distributed object client also performs the batch variant of the write benchmark faster in server mode than in client mode. All in all, just like the other benchmarks in server mode, this shows again that with good optimization it is possible to get similar data storage performance using Python, when compared to C.

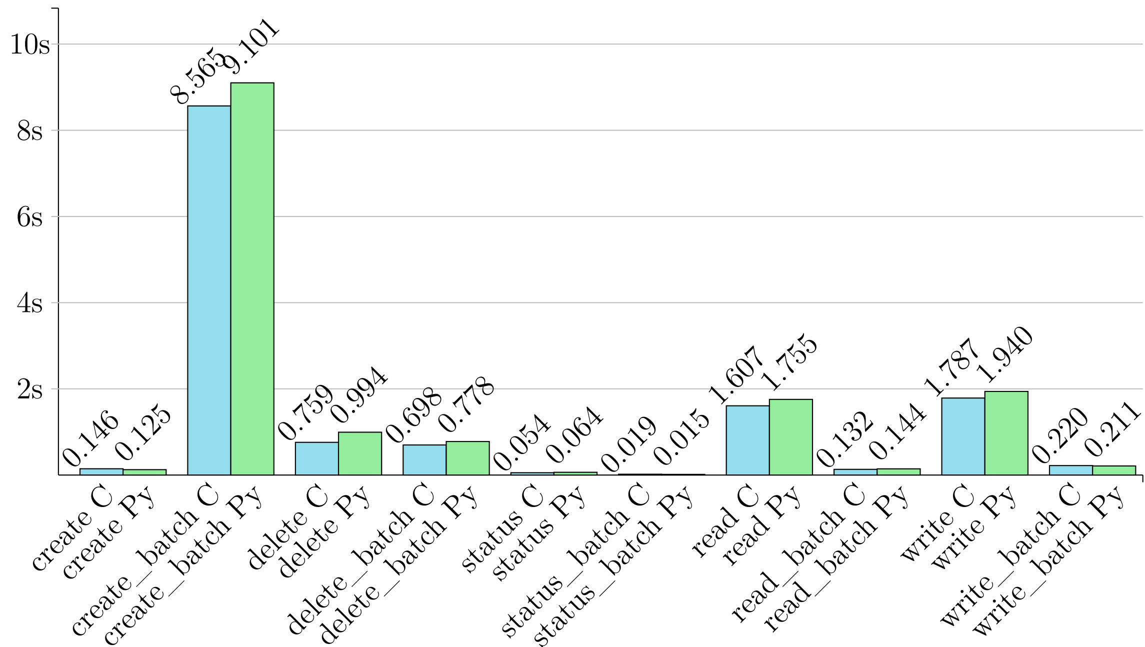

Figure 6.8.: Distributed Object POSIX Server Performance

6.5. Profiling

Profiling was done on the same machine as the Python library tests, and therfore on an other machine than the other benchmarks, but due to the profiling overhead, these results are not meant to be comparable to the benchmarks anyways. For profiling the `cProfile` module of Python was used together with the `pstats` module to get readable numbers out of it. The indentation of the function names within the tables shows which function is called within which other function and all function calls with execution times close to or under 1 millisecond were ignored for the tables, since they are not really relevant for the full execution time.

The first table is table 6.3 and was created by profiling the object create benchmark using the default settings and with one batch for each create call.

Function	Calls	Total time	Percentage
bench_object.py(bench_create)	1	9.604s	100%
- batch.py(execute)	100000	5.625s	58.6%
- batch.py(___exit___)	2	3.132s	32.6%
- batch.py(execute)	2	3.132s	32.6%
- object.py(___init___)	100000	0.275s	2.86%
- 'encode' of 'str'	200000	0.058s	0.60%
- object.py(create)	100000	0.115s	1,20%
- batch.py(get_pointer)	100000	0.012s	0.12%
- object.py(delete)	100000	0.091s	0.95%
- batch.py(get_pointer)	100000	0.012s	0.12%

Table 6.3.: Profiling result of Object Create Benchmark

This benchmark shows that for this benchmark there is very little room for optimization, since the batch execute calls make combined more than 91 percent of the whole call time and these are just basic wrapper functions, that simply pass the task to the C interface. The execute calls are seperated since two of them were called from the `__exit__` function of the class JBatch, so these were the calls that cleaned up afterwards with the actual calls from the benchmark only taking about 5.6 seconds.

Within JObject's constructor were another 0.275 seconds spent, which were mostly by calling the C interface, but more than 20 percent of this time were spent again while encoding the strings that were given for the name and namespace. This encoding is necessary for C to detect the Python strings correctly and they could be already encoded before the constructor gets them, but since the overhead is very minimal with 0.058 seconds for 100000 objects, this is a tradeoff that makes using the interface simpler with very little additional overhead. Also this overhead could only be reduced, when changing this, when either the namespace or name would be the same everytime in the program, since it would need to be encoded everytime otherwise as well.

The next profiled benchmark is the object write benchmark in table 6.4, since it has quite a different call structure, because only one objectis used within the benchmark and the write call also does more work internally.

Function	Calls	Total time	Percentage
bench_object.py(bench_write)	1	8.737s	100%
- batch.py(execute)	200000	7.300s	83.6%
- object.py(write)	200000	1.189s	13.6%
- ctypes(create_string_buffer)	200000	0.314s	3.59%
- ctypes(byref)	400000	0.053s	0.61%
- builtins(len)	200000	0.041s	0.47%
- batch.py(get_pointer)	200000	0.026s	0.30%

Table 6.4.: Profiling result of Object Write Benchmark

This time the batches only give one set of execute calls, since the delete call only needs to delete one object this time for cleanup and that call falls under the minimal measured millisecond. Another difference is the percentage of time spent in the execute calls, since this makes only 83.6 percent of the time for this benchmark. This is due to the more complex write calls compared to the create calls last time, as the benchmark stays in the write function for 13.6 percent of the time.

Most of this time is again spent with the C calls themselves as these are also more complex than for the create calls. But with nearly 3.6 percent of the total time, a significant time span is also spent within the `create_string_buffer` function from ctypes, which is necessary to create a buffer that can be provided to the C interface as a data input. With `byref` comes another necessary ctypes call next, pointers are needed to be provided to the C interface and byref provides pointers for every ctypes element it recieves. More time is also spent with the **`len`** function, which is builtin in Python. This is like the string encodes also only provided for comfort reasons, as this prevents us from having to provide the length of the input data by ourselves each time and with under 0.5% it is again a tradeoff with little overhead. The last mentioned function was also present in the table before and the function `get_pointer` is necessary for every JULEA call that needs to be executed, as a C pointer for the batch is needed in this cases and not only the Python JBatch object.

Next is a profile of the KV put benchmark in table 6.5, which is also interesting since the key-value pairs have a very different approach than the objects.

Function	Calls	Total time	Percentage
bench_kv.py(bench_put)	1	14.629s	100%
- batch.py(execute)	200000	6.630s	45.3%
- batch.py(__exit__)	2	5.589s	38.2%
- batch.py(execute)	2	5.589s	38.2%
- kv.py(put)	200000	0.877s	5.99%
- ctypes(create_string_buffer)	200000	0.248s	1.70%
- ctypes(byref)	400000	0.039s	0.27%
- builtins(len)	200000	0.028s	0.19%
- batch.py(get_pointer)	200000	0.023s	0.16%
- kv.py(__init__)	200000	0.508s	3.47%
- 'encode' of 'str'	400000	0.094s	0.64%
- kv.py(delete)	200000	0.193s	1.32%
- batch.py(get_pointer)	200000	0.023s	0.16%
- kv.py(__del__)	200000	0.122s	0.83%

Table 6.5.: Profiling result of KV Put Benchmark

Since a put call is like a combination of a create and a write call, the results of the profiling also look like a mix from the previous two tables. The batch executions are split up again like in the first table, but since the function has a similar complexity as the write, the batch executions only make about 83.5 percent of the total time again.

The remaining time is mostly split up between the put calls and the constructor calls. The former has a very similar structure as the write call in the last table with `create_string_buffer`, `byref`, **`len`** and `get_pointer` calls, while the latter is very similar to the constructor in the first table, consisting of only string encoding and the C call itself. Another small chunk of time is spent with the delete function and new compared to the other profiling results is the destructor, which simply calls the `unref` function from the C interface to free the memory, which takes significantly longer with the key-value pairs compared to the objects.

The last table 6.6 provides some profiling for the item create benchmark, which is also interesting, as it differentiates itself from the object create benchmark, since it has the create integrated into the constructor.

Function	Calls	Total time	Percentage
bench_item.py(bench_create)	1	0.512s	100%
- batch.py(execute)	1000	0.378s	73,8%
- batch.py(___exit___)	4	0.090s	17,6%
- batch.py(execute)	4	0.090s	17.6%
- item.py(___init___)	1000	0.025s	4,88%
- 'encode' of 'str'	1000	0.002s	0,39%
- item.py(delete)	1000	0.004s	0,78%
- item.py(___del___)	1000	0.003s	0,59%

Table 6.6.: Profiling result of Item Create Benchmark

This benchmark is much shorter, since it only has 1000 iterations, but it shows a very similar image as the object create benchmark in table 6.3 as the combined batch executions also make more than 91 percent of the execution time of the item create benchmark. The constructor in this benchmark also takes nearly 5 percent of the execution time, which is more than the constructor and the create function of the object create benchmark combined, which is due to the extra complexity with also having a collection and the option to provide a distribution. The collection was also called, but since it only needed to be called once at the start, it was not even considered, because it took less than 1 millisecond. As with the last benchmark as well, the profiling table is rounded off with the delete function and the destructor, which both only consume less than one percent of the execution time.

7. Summary, Conclusion and Future Work

In this chapter, the thesis will be summarized and a conclusion of the results gained from this thesis will be taken. Additionally some possible future work will be discussed to build upon the results of this thesis.

7.1. Summary

This thesis showed an approach to bring HPC storage solutions to Python, which, as mentioned in chapter 1, is a quickly growing progrmming language with increasing importance in HPC. Chapter 2 then provided more information to Python, the used storage solution JULEA and the libraries in consideration for the implementation. After having a short excursion to the usages and current storage solutions for Python in chapter 3, chapter 4 went more into depth how this should be implemented. As the first design decision, the above-mentioned libraries were discussed and compared to select one which would be most suited for the implementation. The chapter also discussed the design approaches taken for the different clients of JULEA and which parts need to featured within the Python interface.

Chapter 5 then discussed what was needed to do in the implementation with focus on the functions that required more work to be accessible from Python. In chapter 6 this implementation was then taken and benchmarked against the C implementation. Additionally there were also more benchmarks done to show the differences of the Python libraries and the implementation was also profiled to see where the overhead comes from.

7.2. Conclusion

As shown in the first part of chapter 6, Python code can be hugely accelerated by using C code, with very little performance impact compared to the native C implementation. This also translated to the implementation of the storage solution in Python, as using JULEA within Python can be nearly as fast as the C implementation in some cases. This is also hugely dependent of the usage itself, as with optimization of the batches and how they are executed, much performance can be gained.

Another big part is also the much simpler usability as the JULEA code has been transformed to a completely functioning object-oriented interface in Python. As shown with the example application in chapter 5, this also provides the chance of much shorter code, which can be quickly implemented for use in a Python program or even testing of new possible features for JULEA.

This also shows that Python is well suited for HPIO systems in general, which comes on top on Python's already well functioning support for HPC applications, with several libraries available to enhance the performance.

7.3. Future Work

With the basic clients now implemented, any upcoming clients like the structured metadata (SMD) client [Str19] will need their own implementation with their own class with in Python as well. This however will only be necessary for the clients and some of their dependencies, like e.g. the collection for the item client, not for the backends.

Another possible addition for the Python interface would be some more internal functions, as the Python interface for example currently does not have any access to the configuration. Also, to make the batches more configurable, a Python implementation of the semantics would be possible, as the batches currently only support the default semantics templates, but this would take some work as the semantics utilize many different enums.

Additionally the usage of the Python interface in a real world application or some application with more mixed accesses of key-value and object datatypes with different amounts of data would also help evaluate the performance further.

Bibliography

[Aah03] Aahz. *PEP 328 – Imports: Multi-Line and Absolute/Relative*. Dec. 21, 2003. URL: https://www.python.org/dev/peps/pep-0328/ (visited on 04/29/2019).

[Col] Andrew Collette. *HDF5 for Python*. URL: https://www.h5py.org/index.html (visited on 07/18/2019).

[Enk13] Jussi Enkovaara. *Python in High performance computing*. Apr. 16, 2013. URL: http://www.training.prace-ri.eu/uploads/tx_pracetmo/pythonHPC.pdf (visited on 06/17/2019).

[Fow16] Matt Fowler. *Speeding up Python and NumPy: C++ing the Way*. Mar. 20, 2016. URL: https://medium.com/coding-with-clarity/speeding-up-python-and-numpy-c-ing-the-way-3b9658ed78f4 (visited on 06/07/2019).

[KKL16] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. "Data Compression for Climate Data". In: *Supercomput. Front. Innov.: Int. J.* 3.1 (Jan. 2016), pp. 75–94. ISSN: 2409-6008. DOI: 10.14529/jsfi160105. URL: https://doi.org/10.14529/jsfi160105.

[Kri+13] Mads Kristensen et al. "Bohrium: Unmodified NumPy Code on CPU, GPU and Cluster". In: Nov. 2013.

[Kuh17] Michael Kuhn. "JULEA: A Flexible Storage Framework for HPC". In: *High Performance Computing*. Ed. by Julian M. Kunkel et al. Cham: Springer International Publishing, 2017, pp. 712–723. ISBN: 978-3-319-67630-2.

[Kuh19] Michael Kuhn. *JULEA Example Application*. Mar. 19, 2019. URL: https://github.com/wr-hamburg/julea/blob/master/example/hello-world.c (visited on 07/18/2019).

[Lun] Monte Lunacek. *Python for High Performance Computing*. URL: https://sea.ucar.edu/sites/default/files/python_hpc.pdf (visited on 07/18/2019).

[Mon] MongoDB. *PyMongo 3.8.0 Documentation*. URL: https://api.mongodb.com/python/current/ (visited on 07/18/2019).

[Pas] Lars Pastewka. *MPI-parallel Lattice Boltzmann with Python*. URL: https://github.com/IMTEK-Simulation/LBWithPython (visited on 07/30/2019).

[PG19] Lars Pastewka and Andreas Greiner. "HPC with Python: An MPI-parallel implementation of the Lattice Boltzmann Method". In: Universität Tübingen, Apr. 2019, pp. 119–133. URL: http://dx.doi.org/10.15496/publikation-29049.

[Pug16] Jean Francois Puget. *A Speed Comparison Of C, Julia, Python, Numba, and Cython on LU Factorization*. Jan. 15, 2016. URL: https://www.ibm.com/developerworks/community/blogs/jfp/entry/A_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization?lang=en (visited on 06/18/2019).

[Rig12] Armin Rigo. *Support for # include in cdef*. Aug. 30, 2012. URL: https://groups.google.com/forum/#!topic/python-cffi/vDAw37NHRSg (visited on 06/20/2019).

[RWC01] Guido van Rossum, Barry Warsaw, and Nick Coghlan. *PEP 8 – Style Guide for Python Code*. July 5, 2001. URL: https://www.python.org/dev/peps/pep-0008/ (visited on 04/30/2019).

[Str19] Michael Straßberger. *Structured metadata for the JULEA storage framework*. Online https://wr.informatik.uni-hamburg.de/_media/research:theses:michael_strassberger_structured_metadata_for_the_julea_storage_framework.pdf. Bachelor's Thesis. May 2019.

[TIO19] TIOBE. *TIOBE Index*. June 9, 2019. URL: https://www.tiobe.com/tiobe-index/ (visited on 06/18/2019).

[War+00] Barry Warsaw et al. *PEP 1 – PEP Purpose and Guidelines*. June 14, 2000. URL: https://www.python.org/dev/peps/pep-0001/ (visited on 07/18/2019).

Appendices

A. Abbreviations

Abbreviations

CFFI C Foreign Function Interface

CPU Central Processing Unit

FFI Foreign Function Interface

GPU Graphics Processing Unit

HPC High Performance Computing

HPIO High Performance I/O

KV key-value

MPI Message Passing Interface

PEP Python Enhanced Proposal

POSIX Portable Operating System Interface

SMD structured metadata

B. CFFI Code

CFFI API Mode

```
from cffi import FFI
ffibuilder = FFI()

ffibuilder.set_source("_j_kv_cffi",
"""
    #include "julea-kv.h"
    #include "julea.h"
""",
    libraries=['julea-kv', 'julea'],
    include_dirs=["../../include"])

ffibuilder.cdef("""
    typedef int... guint32;
    typedef int... gint;
    typedef int... guint;
    typedef int... uint32_t;
    typedef int... uint8_t;
    typedef char... gchar;
    typedef gint gboolean;
    typedef void* gpointer;
    typedef struct bson_t bson_t;
    typedef struct JList JList;
    typedef struct JListElement JListElement;
    typedef void (*JListFreeFunc) (gpointer);
    typedef gpointer (*JBackgroundOperationFunc) (gpointer);
    typedef struct JSemantics JSemantics;
    typedef union _GMutex GMutex;
    typedef struct _GCond GCond;

    union _GMutex
    {
        gpointer p;
        guint i[2];
    };

```

```
36      struct _GCond
37      {
38          gpointer p;
39          guint i[2];
40      };
41
42      typedef struct JKV
43      {
44        guint32 index;
45        gchar* namespace;
46        gchar* key;
47        gint ref_count;
48          ...;
49      } JKV;
50
51      struct bson_t {
52          uint32_t flags;
53          uint32_t len;
54          uint8_t padding[120];
55      };
56
57
58      typedef struct JBackgroundOperation
59      {
60          JBackgroundOperationFunc func;
61          gpointer data;
62          gpointer result;
63          gboolean completed;
64          GMutex mutex[1];
65          GCond cond[1];
66          gint ref_count;
67          ...;
68      } JBackgroundOperation;
69
70      typedef struct JBatch
71      {
72        JList* list;
73          JSemantics* semantics;
74          JBackgroundOperation* background_operation;
75          gint ref_count;
76          ...;
77      } JBatch;
78
79      struct JList
```

```
80      {
81          JListElement* head;
82          JListElement* tail;
83          guint length;
84          JListFreeFunc free_func;
85          gint ref_count;
86      };
87
88      struct JListElement
89      {
90          JListElement* next;
91          gpointer data;
92      };
93
94      struct JSemantics
95      {
96          gint atomicity;
97          gint consistency;
98          gint persistency;
99          gint concurrency;
100         gint safety;
101         gint security;
102         gint ordering;
103         gboolean immutable;
104         gint ref_count;
105     };
106
107     JKV* j_kv_new(gchar const* namespace, gchar const* key);
108
109     void j_kv_put (JKV* kv, bson_t* value, JBatch* batch);
110
111     void j_kv_get (JKV* kv, bson_t* value, JBatch* batch);
112
113     void j_kv_delete (JKV* kv, JBatch* batch);
114 """)
115
116 if __name__ == "__main__":
117     ffibuilder.compile(verbose=True)
```

CFFI API Mode with C Header

```
1  from cffi import FFI
2  ffibuilder = FFI()
3
4  includes = open('../../include/kv/jkv.h').read()
5  source = open('../../lib/kv/jkv.c').read()
6
7  ffibuilder.cdef(includes)
8  ffibuilder.set_source("_j_kv_cffi", source,
9      libraries=['julea-kv', 'julea'],
10     include_dirs=["../../include"])
11
12 if __name__ == "__main__":
13     ffibuilder.compile(verbose=True)
```

CFFI ABI Mode

```
from cffi import FFI
ffi = FFI()

lib = ffi.dlopen("libjulea-kv.so")

ffi.cdef("""
    typedef int... guint32;
    typedef int... gint;
    typedef char... gchar;

    typedef struct JKV
    {
      guint32 index;
      gchar* namespace;
      gchar* key;
      gint ref_count;
        ...;
    } JKV;

    JKV* j_kv_new (gchar const*, gchar const*);
""")

lib.j_kv_new("python", "test")
```

C. Python Library Tests

C Code

```
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

int calc(int j) {
    unsigned long num = j;

    for (int i = 1; i < 50000; i++) {
        num *= i;
        num += 1;
        num = num / i;
    }

    return num;
}

int test_call() {
    return 1;
}

int main(int argc, char* argv[]) {
    clock_t begin = clock();

    for (int i = 0; i < 10000; i++) {
            calc(i);
        }

    clock_t end = clock();
    double time_spent = (double)(end - begin) /
       ↪ CLOCKS_PER_SEC;
    printf("%lf \n", time_spent);

    begin = clock();

    for (int i = 0; i < 1000000; i++) {
```

```
35                test_call();
36            }
37
38        end = clock();
39        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
40        printf("%lf \n", time_spent);
41    }
```

Python Code

```
import ctypes
import time
import numpy
from cffi import FFI

start = time.time()
LIB = ctypes.CDLL("./libsimple-calc.so")
for i in range(0, 10000):
    LIB.calc(i)
end = time.time()
print(end - start)


start = time.time()
ffi = FFI()
ffilib = ffi.dlopen("./libsimple-calc.so")
ffi.cdef("""
int calc(int);
""")
for i in range(0, 10000):
    ffilib.calc(i)
end = time.time()
print(end - start)


start = time.time()
for _ in range(0, 10000):
    num = 1
    for i in range(1, 50000):
        num = num * i
        num = num + 1
        num = num / i
end = time.time()
print(end - start)


start = time.time()
for _ in range(0, 10000):
    num = 1
    for i in range(1, 50000):
        num = numpy.multiply(num, i)
        num = numpy.add(num, 1)
```

```
43          num = numpy.divide(num, i)
44  end = time.time()
45  print(end - start)
46
47
48  start = time.time()
49  LIB = ctypes.CDLL("./libsimple-calc.so")
50  for i in range(0, 1000000):
51      LIB.test_call()
52  end = time.time()
53  print(end - start)
54
55
56  start = time.time()
57  ffi = FFI()
58  ffilib = ffi.dlopen("./libsimple-calc.so")
59  ffi.cdef("""
60  int test_call();
61  """)
62  for i in range(0, 1000000):
63      ffilib.test_call()
64  end = time.time()
65  print(end - start)
```

List of Figures

1.1. Development of computational and storage speed over 30 years [KKL16, p. 75] . . . 8

6.1. Key-Value Null Client Performance . . . 31
6.2. Key-Value leveldb Server Performance . . . 32
6.3. Object Null Client Performance . . . 33
6.4. Object POSIX Client Performance . . . 34
6.5. Object POSIX Server Performance . . . 35
6.6. Distributed Object Null Client Performance . . . 36
6.7. Distributed Object POSIX Client Performance . . . 36
6.8. Distributed Object POSIX Server Performance . . . 37

List of Listings

4.1. CFFI API Mode . . . 16
4.2. CFFI ABI Mode . . . 17
4.3. Ctypes JKV New Call . . . 17

5.1. JKV Python Constructor . . . 20
5.2. JKV Python Delete Function . . . 20
5.3. JKV Python Put Function . . . 21
5.4. JKV Python Get Function . . . 21
5.5. JObject Python Write Function . . . 22
5.6. JObject Python Status Function . . . 22
5.7. JDistribution Python Constructor . . . 23
5.8. JCollection Python Constructor . . . 24
5.9. JItem Python Constructor . . . 25
5.10. JULEA C Hello World Application [Kuh19] . . . 27
5.11. Python Hello World Application . . . 27

6.1. calc function . . . 28
6.2. test_call function . . . 29

kv_cffi_build_api.py . . . 48
kv_cffi_build_api_header.py . . . 51
kv_cffi_build_abi.py . . . 52

perf_test/simple_calc.c . . . 53
perf_test/simple_calc.py . . . 55

List of Tables

6.1. Performance results using calc function 29
6.2. Performance results using test_call function 30
6.3. Profiling result of Object Create Benchmark 38
6.4. Profiling result of Object Write Benchmark 39
6.5. Profiling result of KV Put Benchmark 40
6.6. Profiling result of Item Create Benchmark 41

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Wirtschaftsinformatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum Unterschrift