pyCubexR

A Python library to parse Cube files

David Marlon Gengenbach
July 8, 2020

Contents

1 Background **1**
1.1 Performance modeling . . . 1
1.2 Cube file format . . . 2
1.3 Related software . . . 4

2 The pyCubexR file reader **4**
2.1 Architecture . . . 5
2.2 Implementation . . . 5

3 Future work **5**

1 Background

An important task in the development life-cycle, especially in high-performance computing, is getting an insight into the performance characteristics of a given program. Often, knowing how a system will scale with the input size, varying computing resources or other, program-specific parameters is of great importance as it determines the usefulness in many applications. While the performance of relatively simple applications can be modeled intuitively with hand-crafted benchmarks, such an approach becomes more and more difficult with increasing complexity introduced by more distributed systems, for example. Another noticeable disadvantage of hand-crafted benchmarks is that measured data must be evaluated manually as well, often resulting in custom file formats to save measurements and programs to analyze them. An alternative to hand-crafted performance measurements is more standardized solutions, such as Score-P or Extra-P, which provide a working foundation for sustainable performance measurement **and** analysis in one. Since such performance modeling tools often use common file formats, such as the `Cube` file format [5, 3], they also achieve interoperability with similar software.

While there is an active ecosystem of software related to performance modeling and file formats it uses, this work aims to provide a library to parse the aforementioned `Cube` file format. The goal is to contribute a `Cube` parser with minimal dependencies which is both easy-to-use and achieves great performance.

Before presenting our contribution, the `pyCubexR` library, and our requirements, we provide more background on the motivation for the `Cube` file format and software which create or support it.

1.1 Performance modeling

There are several approaches and differing granularity in benchmarking. While a simple run-time measurement provides a coarse insight into the performance of a program, it often fails to deliver actionable information for improvement. More fine-grained metrics, such as timings for single function calls in an application run, on the other hand, can result in an insurmountable amount of data. Existing performance measurement software tries to minimize the impact of measurement and still provide a fine granularity - all while aiming to keep the size of the measurement minimal. In the light of these efforts, a file format called `Cube` was introduced.

1.2 Cube file format

The Cube file format[1], not to be confused with the Gaussian CUBE file format[2], is a container for performance measurements. Initially, the Cube file format was defined as part of a greater framework, also called Cube. However, since then, several tools started to integrate the file format without using the framework.

Apart from providing a format for structuring measured metrics, Cube files also allow for space-efficient storage by compressing data by default using the `*.tar.gz` archive scheme. While there are multiple versions of the Cube file format, we will focus on version 4.4 [5].

Data stored in Cube files The main contents of a Cube file are metrics and their measurements. A metric consists of a name, such as `Executing time` or `Visits`, and associated measured values.

1.2.1 Archive structure

As mentioned before, Cube files are `tar.gz` archives with a pre-defined structure. The archive always contains an `anchor.xml` file describing the performance measurements and providing an index to parse the rest of the Cube file. Figure 1 shows an example of a Cube file.

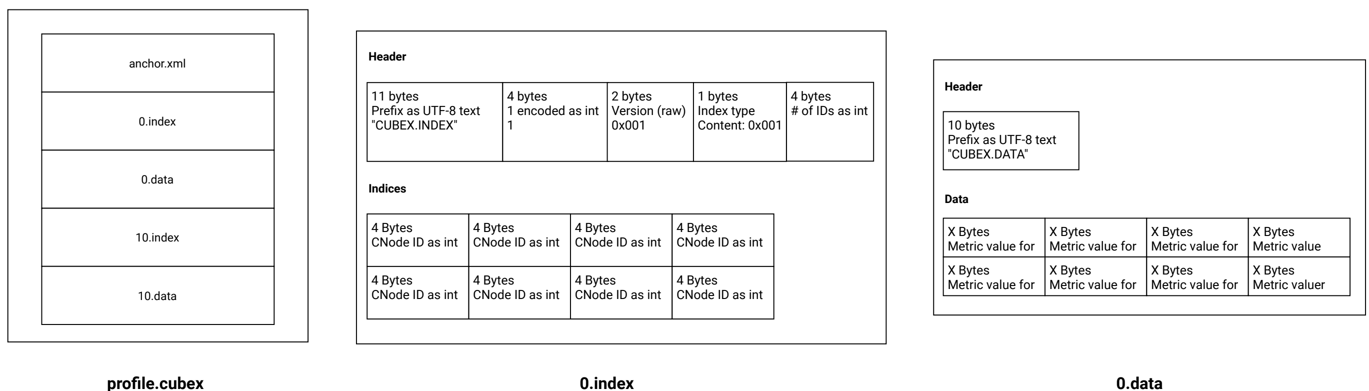

Figure 1: Cube file structure. The left-most diagram shows the Cube archive structure whereas the other two show the structure of metric index and data files, respectively

As we can see in Figure 2, the `anchor.xml` contains meta-data for the measurements and provide context for analysis. Here, we also see that each entity in the XML has a ID which is used to reference the entity in other sections of the XML.

`<metrics>` section Cube uses the metric IDs to name the metric data and index files, as seen in Figure 2. The metric with ID `0`, for example, results in metric filenames as `0.data` and `0.index`. Another important thing to note is the `<dtype>`, or data type, of a metric: these data types are needed to parse the actual measurements data files, `0.data` for example.

`<program>` section The program section in the XML contains data about the actual program callpaths. Here, the `<regions>` provide a mapping of source code regions, such as functions, to an internal callpath tree structure, the `<cnodes>`.

A `<cnode>`, on the other hand, signifies an actual call-path node. The `calleeId` references the corresponding `<region>`. The distinction between `cnode` and `region` is needed since a function can be called from different functions, resulting in different call-trees. The regions provide de-duplication here - meaning that a given function must only be described once in a `region` instead of for each `cnode`. As the cnodes signify the call-tree of a program, they are nested. While a cnode with a given `calleeId` can appear multiple times, the cnode `id` must be unique.

`<system>` section The system section contains information about the machines, processes, and threads that actually executed the code. Here, a hierarchy can be defined, starting with `systemtreenodes` in the root and ending with `locations` in the leaves.

Figure 3 shows the hierarchy of the `<system>` entries. Please note measurements are only done at `locations` and that the rest of the `<system>` hierarchy simply provides a mechanism to further structure measurement data.

[1] `https://www.scalasca.org/software/cube-4.x/download.html`, accessed on 28th June, 2020
[2] `http://paulbourke.net/dataformats/cube/`, accessed on 28th June, 2020

```xml
<metrics>
   <metric
      id="0"
      type="EXCLUSIVE">
      <disp_name>Visits</disp_name>
      <uniq_name>visits</uniq_name>
      <dtype>UINT64</dtype>
   </metric>
   <metric
      id="10"
      type="INCLUSIVE">
      <disp_name>Usage</disp_name>
      <uniq_name>Usage</uniq_name>
      <dtype>UINT32</dtype>
   </metric>
   <!-- ... more metrics -->
</metrics>
```

```xml
<program>
   <region
      id="267"
      mod="/path-to-the-program-source-code/some_file.cpp"
      begin="-1"
      end="-1">
      <name>MAIN__</name>
      <mangled_name>MAIN__</mangled_name>
      <paradigm>compiler</paradigm>
      <role>function</role>
      <url />
      <descr />
   </region>
   <!-- more regions -->
   <cnode
      id="0"
      calleeId="267">
      <cnode
         id="1"
         calleeId="268">
         <!-- more nested cnodes -->
      </cnode>
   </cnode>
</program>
```

```xml
<system>
   <systemtreenode
      Id="0">
      <name>machine A</name>
      <class>machine</class>
      <systemtreenode
         Id="1">
         <name>node A</name>
         <class>node</class>
         <locationgroup
            Id="0">
            <name>MPI Rank 0</name>
            <rank>0</rank>
            <type>process</type>
            <location
               Id="0">
               <name>Master thread</name>
               <rank>0</rank>
               <type>thread</type>
            </location>
            <!-- more locations -->
         </locationgroup>
      </systemtreenode>
   </systemtreenode>
</system>
```

Figure 2: Example `anchor.xml` with two metrics (*Visits* and *Usage*) and one location (*Master thread*)

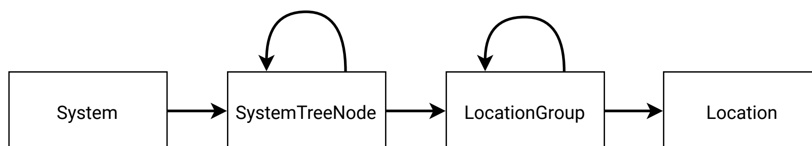

Figure 3: `anchor.xml` `<system>` hierarchy

`*.index` metric file Apart from the `anchor.xml`, the archive also contains the actual metric measurements in a binary format. In our example in Figure 1, the archive contains measurements for two metrics with IDs 0 and 10. Through the `anchor.xml` we can find out the actual names of the metrics with these IDs. Please note that not all metrics defined in the `anchor.xml` have corresponding measurements.

As we can see in Figure 1, the `*.index` file consists of a header and indices. The header contains a predefined string (`CUBEX.INDEX`) to identify the file and provide a sanity check. Next, it contains the number 1 encoded as a signed integer. This 1 is useful to check the endianess[3] of the data: when unpacking the binary 1 into a signed integer, it should equal 1. When it contains a different value, the endianess with which the value was encoded in the first place is not the same as the endianess of the machine unpacking it - resulting in wrong values. All subsequent parsed numbers, including the `*.data` files, **MUST** be unpacked using the right endianess. With a "wrong" endianess, not only the index file will be parsed wrongly but even when the unpacking is successful, the unpacked measurement data will be wrong in all cases. Next, the `*.index` file contains other control fields, the version, and index type. The version should correspond to the version defined in `anchor.xml`. The index type, on the other hand, defines how the index is defined, densely, or sparse. In all our `Cube` files, even when all indices were defined, only the sparse index type was used. The last part of the header is the number of elements in the following list.

After the header, the `*.index` file then continues with a list of cnode IDs.

`*.data` metric file As we can see in Figure 1, the `*.data` file also starts with a predefined string (`CUBEX.DATA`). The rest of the file bytes contain the actual measurement values for the metric. To parse the values, the data type of metric is taken from the `anchor.xml`. The number of individual values depends on two factors: (1) the number of cnodes (c) as defined in the corresponding `*.index` file, and (2) the number of locations (l) as defined in the `anchor.xml`. The `*.data` file should contain exactly $c * l$ values, so one value for each cnode and location.

When introducing our library in the next sections, we will provide a more high-level overview of `Cube` file parsing.

[3] `https://en.wikipedia.org/wiki/Endianness`, accessed on 02.07.2020

1.3 Related software

As mentioned before, the ecosystem of performance modeling is rather active and integrated. In this section, we introduce some of the available software with the focus of its ability to parse and use the `Cube` file format. In the next section, we will get into our motivation to create a yet another parsing library instead of reusing existing ones.

Score-P is software to instrument source code to enable measurements. Here, instrumentation entails changing the source code to add measurement capabilities, often automatically by using special compilers. When running an instrumented program, the instrumentation run-time then gathers performance data, such as the number of calls to a specific function or the time spent on a given function. These measurements are then saved into a predefined file format, such as the `Cube` or OTF2 format[4]. For an explanation of the approach of Score-P, see [4].

Score-P can be found online[5].

Extra-P Given measurements produced by Score-P, for example in the `Cube` file format, Extra-P can generate performance models. Here, instead of looking at a single measurement, the impact of a varying parameter, such as the number of used processors, on the performance is modeled. So, the relationship between a parameter and the resulting performance is analyzed. Score-P also allows the automatic discovery of performance bugs or critical places in the program code that get affected by a parameter change. While the current Extra-P version is written in a mix of C++ and Python, a newer version will be ported to Python only. For more information on the approach of Extra-P, see [1, 2, 6].

Extra-P can be found online[6].

CubeLib is a high-performance C++ library for parsing and processing `Cube` files. While CubeLib provides excellent installation instructions, it relies on several dependencies because of its additional functionalities apart from parsing.

CubeLib can be found online[7].

jCubeR is a Java library for parsing `Cube` files. It is rather easy to install and has great compatibility with the different `Cube` file format versions.

jCubeR can be found online[8]

cubex is a Python library for parsing `Cube` files. While cubex also provides almost all desired functionality, unfortunately, it does not handle the endianess of `Cube` data files correctly. Another drawback regarding development is the rather complex way in which parsing is done: parsing is performed across multiple classes which therefor act both as data containers and parsers. That said, Cubex provided a good source for looking up how the binary `Cube` file format is structured since the structure is not clearly defined via a specification.

cubex can be found online [9].

2 The pyCubexR file reader

Our goal for this work is to provide an easy-to-use and light-weight Python library to parse `Cube` files. The need for such a library arose from the decision by Extra-P to move to Python to have a more light-weight and easier-to-maintain code-base. In particular, the requirements for the library were as follows:

- use modern Python
- parse `Cube` 4.4 files
- retrieve metrics on demand
- little to no dependencies
- easy to use and extend

[4] For further information, see `https://www.vi-hps.org/projects/score-p/`
[5] `https://www.vi-hps.org/projects/score-p/`
[6] `https://www.scalasca.org/software/extra-p`
[7] `https://www.scalasca.org/software/cube-4.x/download.html`
[8] `https://www.scalasca.org/scalasca/software/cube-4.x/download.html`
[9] `https://github.com/marshallward/cubex`

The resulting product is our `Cube` parser library named `pyCubexR`. While we might have extended previous libraries, for example `cubex`, we decided on creating a new library instead to cater to our special requirements. Also, as we have seen in previous sections, existing libraries are either implemented in differing languages or, in the case of `cubex`, work only with specific `Cube` files.

Apart from our `pyCubexR` library, another major contribution is the specification of the `Cube` file format in this document. To our knowledge, the file format was only defined ad-hoc in parsing libraries and no official specification existed until now.

2.1 Architecture

One of the main differences to `cubex` is that our library separates the parsing from the data structures: instead of parsing `anchor.xml` inside data container class, there is a separate parser function that creates simple data container with minimal knowledge of the overall structure. In contrast, the data classes in `cubex` have access to almost the whole `Cube` file, while our library aims to remove these dependencies and move most of the logic into a separate class - thereby separating the concerns of parsing and using the measurements.

Steps In particular, parsing happens in the following steps:

1. Open *.cubex file using the built-in `tarfile` library
2. Open `anchor.xml` from the archive
3. Parse the contents of the `anchor.xml` into internal data classes
4. When measurement data is requested by the user, read in both the binary `METRIC_ID.index` and `METRIC_ID.data` file

One thing to note is that we read in the metric index and data only on demand. This results in little to no overhead during initialization and prevents reading in huge amounts of data that are not used. Additionally, the library still allows the user to explore the contents of the `anchor.xml` via an easy interface. However, when measurements are requested and read, we save the resulting measurements in an internal cache, so a subsequent call will not read the metric files again.

2.2 Implementation

To increase the development ease and prevent common type-based errors before running the program, we use type hints as defined in PEP 484[10] which necessitate using a Python version greater or equal to 3.5 which was released September 13th, 2015, thus most likely having good adoption. We structured the library according to common practices using a `setup.py` file which enables us to use wide-spread package managers, such as `pip`, to distribute and install our work easily. During development, we checked the results of our parsing against the results of both the `CubeGUI` and the `Cube` CLI[11].

To parse the binary metric files, `*.index` and `*.data`, we used the built-in Python library (`struct`). When parsing, we added additional sanity checks to ensure that the parsed data is sensible - especially when dealing with endianess (see previous sections).

The implementation with relevant information on how to install and use our library is published on GitHub[12] under the BSD 3-Clause ”New” or ”Revised” license[13].

3 Future work

One major feature of `Cube` files is the differentiation between inclusive and exclusive metrics. While not in our scope for this project, adding functionality to process the raw metrics before returning them to the user might prove to be useful.

Another crucial feature would be extending the automatic tests. On the same page, another interesting extension would be to allow the creation of `Cube` files. This feature could also simplify the creation of automated tests since one could just create a reference `Cube` file using the library and parse it again - checking whether the encoding/decoding pipeline works as expected.

References

[1] Alexandru Calotoiu et al. “Fast multi-parameter performance modeling”. In: *Proceedings - IEEE International Conference on Cluster Computing, ICCC* (2016), pp. 172–181. ISSN: 15525244. DOI: 10.1109/CLUSTER.2016.57.

[10] https://www.python.org/dev/peps/pep-0484/
[11] We also created a Docker image to simplify using the CubeGUI to prevent installing all dependencies on the host machine
[12] https://github.com/extra-p/pycubexr
[13] https://choosealicense.com/licenses/bsd-3-clause/

[2] Alexandru Calotoiu et al. “Using automated performance modeling to find scalability bugs in complex codes”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* (2013). ISSN: 21674337. DOI: 10.1145/2503210.2503277.

[3] Markus Geimer et al. “Scalable collation and presentation of call-path profile data with CUBE”. In: *Advances in Parallel Computing* 15 (2008), pp. 645–652. ISSN: 09275452.

[4] Dieter Mey et al. “Score-P: A Unified Performance Measurement System for Petascale Applications”. In: *Competence in High Performance Computing 2010* (2012), pp. 85–97. DOI: 10.1007/978-3-642-24025-6.

[5] Pavel Saviankou et al. “Cube v4: From performance report explorer to performance analysis tool”. In: *Procedia Computer Science* 51.1 (2015), pp. 1343–1352. ISSN: 18770509. DOI: 10.1016/j.procs.2015.05.320. URL: http://dx.doi.org/10.1016/j.procs.2015.05.320.

[6] Sergei Shudler et al. “Exascaling your library: Will your implementation meet your expectations?” In: *Proceedings of the International Conference on Supercomputing* 2015-June (2015), pp. 165–175. DOI: 10.1145/2751205.2751216.