Controlling overhead in large Python programs

Master’s thesis

Øyvind Ingebrigtsen Øvergaard

August 4, 2011
Abstract

Computational overhead is parts of a program’s run time not directly attributable to the job the program is intended to do. In the context of high-level dynamic languages, the term overhead is often used to describe performance in relation to C.

In this thesis I use The Genomic Hyperbrowser, a large and complex Python program, as the case in a study of controlling computational overhead.

Controlling computational overhead can have at least three different meanings, all of which are considered in this thesis.

First, it can mean to assess the amount of overhead in a given program, i.e. estimate how much faster the program could run according to some notion of optimal execution. Several ways of defining overhead is covered in this thesis, together with ways to estimate overhead based on these definitions. Furthermore, I propose ways to manually inspect the run time distribution in programs to determine what parts of your code are slow.

Second, it can mean to limit excess overhead in a program by using specialized tools and techniques to find parts of the code that are particularly good candidates for code optimization to reduce overall running time. Here, I show step by step a real case of finding, analyzing and resolving a performance issue caused by excessive overhead.

A third meaning can be to design a program in such a way as to informally bound the overhead in the final program, i.e. to apply simple design principles that can help avoid the overhead ending up at an inappropriate level. In this thesis I show how central design decisions in the HyperBrowser ended up affecting computational overhead.
7 Future work

7.1 Dynamic CompBin size
7.2 Further analysis of granularity on the file level
7.3 Chunking data sets in HDF5 files

Appendices

A Determining factors for weighting operations
A.1 Functions for timing C++ operations
A.2 C++ function call
A.3 Python function call
A.4 C++ class instantiation
A.5 Python class instantiation
A.6 C++ string concatenation
A.7 Python string concatenation
A.8 C++ list access
A.9 Python list access
A.10 numpy operations

B Plots for binning Runs with IndexBin size 1000

C Call graph subsections for binning Runs
List of Figures

2.1 Description of the internal data structure in the pstats.Stats class. ........................................ 16

3.1 Flow of control in a descriptive statistic Run, calculating a mean. ............................................. 23
3.2 UML Sequence diagram of the HyperBrowser’s data handling subsystem. ..................................... 25
3.3 Illustration of lost space between files in a block-based file system. ............................................. 28
3.4 An illustration of how the different types of Binning operate together on a Track. .............................. 30

4.1 The output of a Run, showing how overhead calculated based on each model is presented. .................. 36

5.1 An example of a bottleneck plot. .......................... 47
5.2 Subsection of a visualized call graph. .................... 49
5.3 Bottleneck plot of the Run described in Section 5.2.1 on page 50. ............................................. 51
5.4 Section of the call graph of the Run described in Section 5.2.1 on page 50 showing the hasattr and VirtualNumpyArray::__getattr__ calls. ............................................. 53
5.5 Section of the call graph of the Run described in Section 5.2.1 on page 50 showing the VirtualNumpyArray::__sub__ calls. ............................................. 54
5.6 Section of the call graph of the Run described in Section 5.2.1 on page 50 showing the VirtualNumpyArray::__getattr__ calls. ............................................. 56
5.7 Bottleneck plot of the Run described in Section 5.2.1 on page 50, after removing a bottleneck. .......... 59

6.1 A Track, split into four CompBins, showing how these proxy numpy objects ............................... 61
6.2 Measuring the run time of summing numpy and Python lists in number of Python function calls. ............ 63
6.3 Measuring the run time of summing \texttt{numpy} and Python lists in number of Python class instantiations. \\
6.4 Plots of how UserBin size and CompBin size affects run time and atom overhead of MeanStat on Meltmap at 100k IndexBin size. \\
6.5 Plots of how UserBin size and CompBin size affects run time and atom overhead of CountPointStat on Genes at 100k IndexBin size. \\
6.6 Plots of how UserBin size and CompBin size affects run time and atom overhead of CountPointStat on Sequence at 100k IndexBin size. \\
B.1 Plots of how UserBin size and CompBin size affects run time and atom overhead of MeanStat on Meltmap at 1k IndexBin size. \\
B.2 Plots of how UserBin size and CompBin size affects run time and atom overhead of CountPointStat on Genes at 1k IndexBin size. \\
B.3 Plots of how UserBin size and CompBin size affects run time and atom overhead of CountPointStat on Sequence at 1k IndexBin size. \\
C.1 Cutouts from the call graphs of MeanStat on Meltmap with UserBin size set to 100k and 1M. \\
C.2 Cutout from the call graph of MeanStat on Meltmap with UserBin size and CompBin size set to 1M. \\
C.3 Cutouts from the call graphs of CountPointStat on Genes with UserBin size set to 100k and 1M. \\
C.4 Cutout from the call graph of CountPointStat on Genes with UserBin size and CompBin size set to 1M. \\
C.5 Cutouts from the call graphs of CountPointStat on Sequence with UserBin size set to 100k and 1M. \\
C.6 Cutout from the call graph of CountPointStat on Sequence, Repeating elements with UserBin size and CompBin size set to 1M.


**List of Tables**

4.1 Factors by which operations are slower in Python than C++. 34
4.2 Overhead measurements for MeanStat on Meltmap. . . . . . . 39
4.3 Overhead measurements for CountPointStat on Sequence, Repeating elements as Segments. 41
4.4 Overhead measurements for CountPointStat on Genes as Segments. 42

5.1 Run times of the Run in Section 5.2.1 on page 50 in various states of the code. 58

6.1 Runtime overview in data set 1. . . . . . . . . . . . . . . . . . . 74
6.2 Runtime overview in data set 2. . . . . . . . . . . . . . . . . . . 75
6.3 Runtime overview in data set 3. . . . . . . . . . . . . . . . . . . 75
6.4 Runtime overview in data set 4. . . . . . . . . . . . . . . . . . . 76
Listings

2.1 Example of weak typing in PHP. .................................. 13
2.2 Example of strong typing in Python. ................................. 13
5.1 The gold.track.VirtualNumpyArray module in its original state. ......................................................... 54
5.2 The gold.track.VirtualNumpyArray module after first try at resolving the performance issue. .......................... 55
5.3 The gold.track.VirtualNumpyArray module after resolving the performance issue. ................................. 57
A.1 A C++ function for timing C++ operations. ......................... 78
A.2 A C++ function for getting the time delta between two clock_t instances. ........................................... 78
A.3 Setup code for timing C++ function calls. ......................... 79
A.4 Timed operation for C++ function call. ............................ 79
A.5 Setup code for timing Python function calls. ....................... 79
A.6 Timed operation for Python function call. ........................ 79
A.7 Setup code for C++ class instantiations. .......................... 79
A.8 Timed operation for C++ class instantiation. ....................... 80
A.9 Setup code for Python class instantiations. ......................... 80
A.10 Timed operation for Python class instantiation. ................... 80
A.11 Setup code for C++ string concatenation. ......................... 81
A.12 Timed operation for C++ string concatenation. .................... 81
A.13 Setup code for Python string concatenation ....................... 81
A.14 Timed operation for Python string concatenation .................. 81
A.15 Setup code for C++ list accesses. ................................. 81
A.16 Timed operation for C++ list access. ............................. 81
A.17 Python list access. .................................................. 82
A.18 Python list access .................................................... 82
Acknowledgements

I would like to thank my supervisors Geir Kjetil Sandve and Torbjørn Rognes for all the guidance, support and feedback. Thanks also goes to my family and friends, and particularly my fellow master students at the tenth floor of Ole Johan Dahls hus, for their support.

Finally I would like to thank my wonderful girlfriend Katrine for her never-ending encouragement and support during this whole process.

Øyvind Ingebrigtsen Øvergaard
University of Oslo
August 2011
Chapter 1

Introduction

1.1 Motivation

Programs written in high-level languages like Python are slower than programs implemented in lower-level languages like C or C++. However, developing a program in Python will typically take less time than developing an equivalent program in C or C++.

Basically we’re looking at a trade-off where we sacrifice low run time for low development time. We can throw more and better hardware at a program to make it run faster but we can’t keep throwing more developers at the development process of a program and expect it to be done faster.

There are several reasons why Python in particular is slower;

A) Python’s dynamic nature causes it to be inherently slower. This is (in part) because type checking and other semantic analysis that would be done at compile time in a statically typed language like C or C++, is done at run time in Python.

B) While Python itself and large parts of its standard library is implemented in C and is thus quite fast, there are parts of it that are implemented in Python, which causes it to be even slower.

If your Python program is too slow to be useful, what can you do? Well, you can rewrite the slow parts or even the entire program in a language that executes faster, like C or C++. However, this will likely cost you a significant amount of expensive developer time, so you’ll only want to do it if it is worth it. Is there some way to know how much you can gain from a rewrite, without actually doing the rewrite itself first? I’m exploring ways to achieve this in Chapter 4 on page 32.
I do a hands-on study of how to use specialized tools to find and resolve a performance issue in Chapter 5 on page 45.

In Chapter 6 on page 60 I investigate effects of program design and architecture on run time at several levels.

1.2 Case

The program I use as the case for this thesis is the Genomic Hyperbrowser, which is described in detail in Chapter 3 on page 21. The HyperBrowser is a relatively large and complex Python program which was designed with heavy focus on flexibility and maintainability. The performance and run time is not really a particularly big issue in the HyperBrowser at present. However, since low run time has not been a primary focus during the development, it is an interesting case to investigate.

1.3 Research questions

1. Is there a way of estimating how large a part of a program’s run time is overhead, where overhead has some precise definition?

2. What tools and techniques can we use to help reduce overhead in Python programs?

3. What design decisions can we make to help control overhead and ensure good performance, while still maintaining flexibility?
Chapter 2

Background

2.1 General

A relatively new field in biology is the study of the human genome. Although there has been done a lot of research involving statistical analysis on genome data, there is not really a common way of doing this. This means that for each publication, scientists typically end up hacking together some quick-and-dirty code that does their specific analysis.

This is what sparked the development of The Genomic HyperBrowser [19] (hereinafter: The HyperBrowser) — a tool that aspires to be the de facto standard tool for biologists wanting to do different sorts of statistical analysis on genome data. The HyperBrowser lets biologists ask questions about the data they are interested in, rather than deal with hard statistical problems.

The HyperBrowser is written in Python, a dynamically typed programming language, with most of the heavy computations done in numpy [16] — a library for fast vector operations.

While the computations themselves are relatively close to C speeds, courtesy of the numpy operations, profiling shows that a significant amount of the run time is spent doing other things.

2.2 Python

Python [22] is a very flexible, dynamically/duck typed programming language. Python is distributed with a large standard library with modules for
everything from reading/writing WAV\textsuperscript{1} files to CGI\textsuperscript{2}.

2.2.1 History

Python was developed by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the late 80’s and early 90’s. The first release of CPython, the Python reference implementation, was released in 1991. Since then, Python has been developed by Guido van Rossum and the Python community; Guido being appointed by the community as its “Benevolent Dictator For Life”. The Python Software Foundation has since 2001 managed the rights to the CPython implementation, which is released under the GNU GPL\textsuperscript{3} compatible Python Foundation Software License.

2.2.2 Implementations

As its name indicates, CPython is implemented in C. However, the CPython implementation is not the only Python implementation; among the others are Jython – Python for the JVM\textsuperscript{4}, IronPython – Python for the .NET CLR\textsuperscript{5} and PyPy – Python implemented in a restricted subset of Python called RPython. PyPy features a Just In Time compiler, which makes it faster than CPython in several benchmarks.

2.2.3 The Language

Type system

It is often hard to describe a type system with a single term. This is also the case for Python. Python’s features strong dynamic/duck typing. These terms are explained below:

Strong typing Strong types mean that the type system imposes restrictions on what operations can be done on objects of differing types. The opposite of strong types are weak types. The difference between the two is best explained by example:

In PHP for instance, this is allowed:

\textsuperscript{1}Raw audio files
\textsuperscript{2}Common Gateway Interface, a standard for how web servers can let external applications generate web pages.
\textsuperscript{3}GNU General Public License
\textsuperscript{4}Java Virtual Machine
\textsuperscript{5}Common Language Runtime
Listing 2.1: Example of weak typing in PHP.

```
$foo = 1 + "2";
echo $foo;
```

What happens here is that the string literal "2" is added to the integer literal 1, which gives us the integer 3. This is possible because PHP’s type system implements weak typing.

Trying to do the same thing in Python however, results in a TypeError, because Python has strong types.

Listing 2.2: Example of strong typing in Python.

```
foo = 1 + "2"
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

**Dynamic and duck typing**

Dynamic typing means that the type of an object is resolved at run time, not at compile time like in statically typed languages.

Duck typing is a variation of dynamic typing that encourages the programmer to check for an objects capability (e.g. the presence of a method) to do something rather than explicitly checking its type. This is used extensively throughout Python’s built-in classes. For instance, in Python, an iterator is any object that has a method named \_\_iter\_\_(). This means that you can iterate over any object, but if it does not implement the \_\_iter\_\_() method, a TypeError will be raised at run time. As a contrasting example, let’s look at how this would be done in Java, which is statically typed and not duck typed: An iterator object in Java would be an instance of a class that implements (i.e. inherits from) the Iterator interface. Trying to iterate over an object that doesn’t inherit from Iterator results in a compilation error.

**Managed memory**

CPython features memory management and garbage collection through reference counting. This means that the CPython virtual machine keeps track of how many references there are to a given object, and when this number reaches 0 the object is garbage collected.
Dynamic features

Python features advanced reflection and introspection capabilities as part of the core language. Many of these features are used extensively throughout the HyperBrowser codebase. These are features that are extremely powerful, but heavy usage can lead to very subtle bugs that are hard to track down.

Magic methods

The Python object model defines a set of methods, commonly referred to as “magic methods” that are called when special syntactic elements are used. Classes can implement these methods and modify the behavior of said syntactic elements; this is Python’s approach to operator overloading, a feature that is many other programming languages have. All of these methods are named according to the naming convention `__<name>__`, that is, a name surrounded by double underscores.

Some of these methods are listed below, to give an idea of the capabilities and scope of this mechanism:

- `__init__` is invoked after an instance of a class has been created, and is typically used to initialize instance variables.
- `__str__` is invoked by the `str` built-in function when it is called on an instance, and returns a string representation of the instance.
- `__call__` is invoked when an instance is called, like an other function (Python has first-class functions).
- `__add__` is invoked on an instance when the add (+) operator is used, for instance in an expression like `baz = foo + bar`.

2.2.4 Profiling

For profiling Python applications, there are three modules available in the standard library

`profile`

`profile` is a profiler for Python implemented in Python itself. Because of this, it is quite slow, and adds a significant overhead to the program it is profiling.

`hotshot`

`hotshot` is a profiler for Python which is implemented in C, as an attempt to solve the problem with excessive overhead when profiling code. `hotshot` is no longer maintained, and is more or less deprecated
for the time being. In fact, a note in its documentation recommends the use of cProfile instead.

**cProfile**

Introduced in Python 2.5, **cProfile** is yet another profiler for Python. **profile** and **cProfile** have roughly the same interface, and are more or less interchangeable, but since **cProfile** is implemented as a C extension it is significantly faster than **profile**. So unless you have to extend the profiler — which would be easier to do with **profile** since it is written in Python — you would most likely want to use **cProfile** because of its performance. **cProfile** is the base for the profiling instrumentation in the HyperBrowser.

**profile and cProfile API**  **profile** and **cProfile** have three entry points;

- **run()** which profiles a Python statement passed in as a string in the current environment.
- **runctx()** which behaves just like **run()** but lets you use your own **globals** and **locals** dictionaries (basically, it lets you specify the entire run time environment).
- **runcall()** which profiles a function call.

The profiler then tracks several parameters about every function call, and stores these in its internal data structure. This data structure can be interrogated by using the **pstats.Stats** class.

**The pstats.Stats class**  The **pstats.Stats** class basically has a copy of the profiler’s internal data structure and several methods for extracting data from it.

The internal data structure is a dictionary where each key is a 3-tuple containing information about the function itself, and each value is a 5-tuple with information about the calls of the corresponding function. As you can see from Figure 2.1 on the following page, the information you can gather about a given function in your program is:

- **Filename** The path to and name of the file in which it is defined.
- **Line number** The line number at which the function is defined in the file.
- **Function name** The name of the function.
- **Primitive calls** The number of **primitive calls** made to the function.

---

6 Calls that are not recursive.
**Total calls** The total number of function calls.

**Total time** The total time the function was the active stack frame — time spent in the function itself.

**Cumulative time** The total time the function was on the call stack — that is time spent inside the function, including time spent in other functions called inside the function.

**Callees** Functions that called this function. This is a dict of functions just like this. Making it possible to interpret this data structure as a directed graph where each node is a function (with information about its calls), with edges originating from each function in its callees dict.

Figure 2.1: Description of the internal data structure in the `pstats.Stats` class.

So the data we have about the execution of our program is basically a call graph with some information about each function, but how useful this information is depends on what we’re trying to achieve.

**Interrogating pstats.Stats instances** The `pstats.Stats` class has several functions to get the information you want from profiling data, but its interface is rather simple, and seems like it is intended to be used from the interactive prompt – the Python REPL\(^7\). The `pstats` module even includes a command-line interface which can be initiated by running `python -m pstats FILE`, where `FILE` is a file created by `pstats.Stats.dump_stats()`, which basically serializes the call graph to a file.

---

\(^7\)Read-Eval-Print Loop, an interactive interpreter for a programming language — in this case Python.
To get at profiling data programmatically, the easiest thing to do is likely to extend the `pstats.Stats` class and with functions that retrieve the information you need from the data structure itself. Indeed, this is what I did to implement the models for determining overhead in the HyperBrowser.

### 2.2.5 Common ways of optimizing Python programs

#### C extensions

The CPython runtime is not particularly fast in comparison to C, C++ or even Java/the JVM. However, CPython implements a C extension API that allows Python developers to implement Python modules in C.

Because of this, a common way of optimizing a Python program is to figure out where the bottlenecks with regards to runtime are in the program and reimplement these as efficient C extensions. Depending on where the bottlenecks are and how orthogonal your program’s architecture is, this might or might not be an easy task.

#### Cython

Cython [6] (not to be confused with CPython) is a superset of Python that is used to generate C code which is in turn compiled, resulting in native programs that generally run faster than Python on CPython.

Cython lets programmers use C-like features like static type declarations, pointers and structs as well as call C functions directly from Python. The resulting programming language looks like a C and Python hybrid language, hence Cython’s name.

### 2.3 MapReduce problem solving

MapReduce [10] is a way of splitting a problem into subproblems, which is then solved separately and ultimately combined into a final result. MapReduce is a two step process, the `map` step and the `reduce` step.

**Map**  In the map step, the problem we want to solve is subdivided into subproblems which are self-contained units that can compute their own result. The map step can easily be parallelized and distributed for speed.
Reduce In the reduce step the partial results from the map step is combined into the final result.

In the HyperBrowser, a MapReduce-like process is implemented through CompBins, which is explained further in Section 3.6.2 on page 29.

2.4 Overhead

Overhead is a term commonly used when talking about the performance of programs, or even programming languages and their runtimes. Overhead typically describes some sort of excessive resource usage, but it can mean different things in different contexts. Some of these are mentioned below.

2.4.1 Computational overhead

When the term overhead is used when talking about programs or parts of programs, like functions, it describes any resource usage that can not be directly associated with solving the problem that the program is intended to solve. That is; indirect resource usage. Resources in this respect is not just limited to CPU time, but also include things like memory, disk I/O and network I/O.

2.4.2 Programming language overhead

Overhead, when used in discussion about programming languages or programming language runtimes, describe the relative performance of one programming language to another; i.e. “Python has significant overhead compared to C”. The baseline in such comparisons are typically C or C++, because these languages are both compiled in to native machine code before running, and are thus quite fast.

2.5 HDF5

HDF5 [14] is the latest generation of the HDF file format; an open-source file format with an over 20 year long history in high performance scientific computing.
2.5.1 History

HDF was initially developed at the National Centre for Supercomputing Applications (NCSA) in the USA in an effort to create a unified file format for scientific computing applications that could be used across many different machine architectures. Since 2006 HDF has been maintained and supported by the HDF Group; a non-profit organization incorporated specifically for the purpose.

HDF5 files are used to store satellite imagery in NASA’s Earth Observation System, where several terabytes of data are produced every day. Other significant HDF5 users are the US Department of Defense, Boeing and HDF5 files were even used by the team that did special effects for the movie “Lord of the Rings”. I guess this is a testament to HDF5’s versatility as a format for storing whatever data you might want to store.

2.5.2 Features

HDF5 is a binary data format for storing structured data. The basic building blocks (objects) in HDF5 are groups and datasets, which can be organized in a tree-like structure.

**Datasets** HDF5 can be used to store just about anything, and the actual data we want to store are stored in datasets. If we look the internal structure of a HDF5 file as a tree, datasets are the leaf nodes.

Datasets are basically arrays of data, along with some metadata:

- **Name** The name of the dataset.
- **Datatype** The data type of the dataset; integers and floating point numbers in different sizes, strings as well as native C data types are among the available data types in HDF5.
- **Dataspaces** The dimensions of the dataset.
- **Storage layout** Data can be stored contiguously or in chunks. Storing data in chunks can have performance benefits when accessing subsets of the data, or when using compression.

**Groups** Groups are containers that can contain zero or more other groups or datasets. The notation for interacting with groups in HDF5 resembles that of interacting with directories and files in a UNIX file system:

- / denotes the root group.

---

8National Aeronautics and Space Administration
- `/foo` denotes a HDF5 object `foo` at the root level.
- `/foo/bar` denotes a HDF5 object `bar` inside the foo group, which itself is at the root level.
Chapter 3

The Genomic HyperBrowser

The Genomic HyperBrowser is a large web-based statistical analysis system for genomic data, intended as a tool for biologists doing research requiring genome analysis. The HyperBrowser supports several different types of genomic data, or genomic annotation tracks.

3.1 Web based run management

The HyperBrowser uses Galaxy [13] [8] as its web front-end. Galaxy provides a framework for integrating programs that closely resemble command-line tools into a web application that lets users pass arguments to and execute these tools from web forms. Galaxy keeps track of all Runs done by a user, listing them in a history that is tied to the user’s account. This way, users can browse previous Runs in a nice web-based interface.

A benefit of the HyperBrowser being web based compared to implemented as a command-line tool is that non-technical users do not have to remember lots of strange command-line flags. Furthermore, users do not have to worry about where the results should be saved for later reference, everything is just there; stored in the run history.

3.2 Runs

A Run in the context of the HyperBrowser is an analysis on one or more Tracks. A Track is a genomic annotation track, described further in Section 3.4 on page 23.
3.2.1 Analysis

In the context of the HyperBrowser, an analysis means a statistical analysis on a Track. This statistical analysis can be a descriptive statistic like calculating a mean or counting base pairs, or it can be a hypothesis test where we ask a question like “Are segments of track A overlapping segments of track B more than expected by chance?”. Any given analysis consists of two phases; first a local analysis, then a global analysis.

3.2.2 Local analysis

The local analysis phase splits the track or tracks into smaller parts called Bins, then does the statistical analysis on each Bin. This splitting is described in Section 3.6 on page 29. The results from the local analysis gives the user a more detailed view of the results from the global analysis. This might also reveal any inconsistencies in the results from one bin to another.

3.2.3 Global analysis

The global analysis phase considers the Track or Tracks in its/their entirety, and does the statistical analysis on the full data set.

3.3 Statistics

Whether the statistical analysis involves a descriptive statistic or a hypothesis test, it involves the application of one or more statistical functions onto the Tracks involved. This functionality is provided by a class hierarchy that provide functionality for many common statistical operations. When Statistics (capitalized) are mentioned throughout the thesis, it refers to these classes. The naming of Statistic modules follow a convention of something descriptive to the statistical function + “Stat”. For instance, a Statistic that calculates mean is named MeanStat, and a Statistic that calculates sum is named SumStat.

Statistics define a relation to other Statistics that typically forms a directed acyclic graph.

A Statistic can delegate calculating parts of the statistical function to other Statistics. This is done by defining a set of child Statistics, which are described in the createChildren method for the given Statistic. These child Statistics can have child Statistics of their own; the relationship between
Statistics can be described as a directed acyclic graph-like hierarchy. This is illustrated in Figure 3.1. Calculating a mean requires a count and a sum, so, \texttt{MeanStat} has two children: \texttt{CountStat} and \texttt{SumStat}, both of which in turn have \texttt{RawDataStat} as their child. \texttt{RawDataStat} deals with reading Tracks of the disk in a manner further described in Section 3.4.

![Figure 3.1: Flow of control in a descriptive statistic Run, calculating a mean.](image)

Every Statistic has a \_\texttt{compute} method which provides the functionality for doing the statistical analysis itself. When this method is called on a Statistic, its child graph is calculated, and then the Statistic itself is calculated, returning its result.

### 3.4 Data handling

The HyperBrowser deals with genomic data stored as \textit{genomic annotation tracks}, which basically are large sequences of \textit{base pairs} (bp). Whenever the word Track (capitalized) is used throughout the thesis, it refers to a genomic annotation track.
The HyperBrowser has a very large amount of genomic data available for statistic analysis. This comes from files in several different plain text formats; e.g. tabulator separated columns, or comma separated values. I think it goes without saying that this isn’t the most efficient way of storing data like this – the HyperBrowser developers thought so too, and decided to make their own format.

Tracks in the HyperBrowser can be marked/unmarked points, marked/unmarked segments, or functions. Marked in this context means that there are some sort of metadata associated with each point or segment in the Track that says something about its location or length (in the case of segments).

**Points** Points have a location on the Track. Interesting information about points include the location of a point in relation to points on the other track.

**Segments** Segments have a location as well as a length. Interesting questions to ask about segments include the amount of genome covered by a segment, or the segment’s location. If we are only interested in the location of segments it is easier to deal with points, so a conversion from segments to points will be done on the Tracks.

**Function** Functions are defined for a specific part or parts of the genome, called the functions domain. The function then yields a value for each base pair within its domain.

The data available to the HyperBrowser comes from various sources in several different file formats. These files are then preprocessed, and finally the data comprising these different representations of tracks are stored as numpy arrays [21] directly dumped from memory to disk.

### 3.4.1 Design

This is a top-down view of the design of the data handling subsystem of the HyperBrowser below. Figure 3.2 on the following page, a UML[^1] [17] sequence diagram of said subsystem, should serve as a good basis for understanding how data handling in the HyperBrowser works.

**RawDataStat**

The **RawDataStat** class is the highest level data abstraction in the HyperBrowser. Statistics use this class to get Track data. **RawDataStat** is a

[^1]: Unified Modelling Language
subclass of Statistic, so other Statistics can access track data by having a RawDataStat instance as one of their child Statistics.

RawDataStat’s initializer takes a Track instance as well as a GenomeRegion instance as its arguments.

Since RawDataStat is a subclass of Statistic, all of the work it does has a single entry point: its _compute method. Since RawDataStat also is the entry point of the HyperBrowser’s data handling subsystem, its _compute method is a good starting point when describing this subsystem.

Figure 3.2: UML Sequence diagram of the HyperBrowser’s data handling subsystem.

Track

Track is analogous to a genomic annotation track. Inside RawDataStat’s _compute method, the getTrackView method is called on its Track instance. This method takes a GenomeRegion instance, which is passed along from RawDataStat.

TrackSource

TrackSource’s job is to get the track data from disk, and return the part of the Track that we are interested in. This work is done in its getTrackData method.
The part of the track that we want is constrained by the start and end properties of the GenomeRegion instance that has been passed to TrackSource from RawDataStat.

**SmartMemmap**

SmartMemmap is a decorator around numpy.memmap. It provides a few convenience methods for dealing with the raw Track data, but most importantly; provides garbage collection of subsets of the Track data.

For the actual mapping of files into memory, numpy.memmap is used. This method basically does a mmap(2) system call, which maps memory addresses to point to a file on disk. The mmap(2) system call is defined as part of the POSIX\(^2\) [4] standard, ensuring that it is available on any POSIX-compliant operating systems.

**NAME**
mmap, munmap - map or unmap files or devices into memory

**SYNOPSIS**

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

**DESCRIPTION**
mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping.

(Excerpt from the mmap(2) manual page [1].)

The Track data in files opened with numpy.memmap is lazily read into memory when needed – this means that only the parts of the file that are actually accessed are read from disk. This reduces memory usage drastically compared to if the entire Track was loaded at once.

**TrackViewLoader**

TrackViewLoader is responsible for instantiating the TrackView instance based on the GenomeRegion instance and the track data from SmartMemmap. This is done in its loadTrackView method. When this method returns, the

\(^2\)Portable Operating System Interface for UNIX
TrackView instance is propagated further up the stack until it reaches the `RawDataStat::compute` call that initiated the process.

### 3.4.2 Problems

#### Too many files

The POSIX standard [4] require file systems in POSIX compliant operating systems to implement a data structure containing metadata about files, called an inode. Inodes are the lowest level data structure when dealing with files, and each inode must be unique.

POSIX states that the header `<sys/stat.h>` must define a type `ino_t` that should be an unsigned integer for storing inode numbers. When that number overflows (wraps back to 0), the operating system can’t allocate new inodes.

This has happened on the server that currently hosts the HyperBrowser. The way the data is stored right now is a large directory tree with many files per chromosome, and several chromosomes per track – so many files in total that there are no more available inodes on the system.

This was solved – temporarily – by upping the inode limit, but ideally this would be solved more permanently by changing the way tracks are stored to a storage model that does not require as many files.

#### Too many open files

Much like inodes (see Paragraph 3.4.2), the maximum number of open files are finite. POSIX states that the `<limits.h>` header file must define a macro `OPEN_MAX` – an integer that is one larger than the largest value one can assign to a file descriptor. Since file descriptors start at 0, the maximum number of open files is the value in `OPEN_MAX`.

In earlier versions of the HyperBrowser, this limit would create problems on some runs, because some of the tracks are stored over many small files.

#### Wasted space

Most common file systems are based on blocks, an abstraction that lets the operating system kernel deal with file data as smaller parts. This means that when you write a file to your hard drive, the operating system kernel allocates storage space for the file as a number of blocks on the hard drive that your file will fit into, and writes it into those blocks. A file’s data always
starts at the beginning of a block, but can end anywhere in the last block allocated for the file.

A problem with this is abstraction that when a file’s size is not a perfect multiple of the block size, the space between the end of the file and the end of the file’s last block is lost. This effect is commonly called *internal fragmentation* or *slack space*.

Figure 3.3 illustrates internal fragmentation; the red rectangles illustrate the lost space at the end of block 3 and block 6, after the end of file 1 and 2, respectively.

![Figure 3.3: Illustration of lost space between files in a block-based file system.](image)

Needless to say, a given file’s size is rarely a perfect multiple of the block size. In fact, assuming that file sizes are randomly distributed, the average lost space per file is half a block.

The genomic data available in the HyperBrowser is stored on a GPFS\(^3\) [20] volume provided by the TITAN compute cluster [2]. This volume is relatively large (70-80 terabytes), and as a consequence of this, it requires a larger block size than the smaller volumes you’ll usually see on workstation computers or small servers. This GPFS volume uses 32 kilobyte blocks, as opposed to the 4 kilobyte blocks commonly used on smaller volumes.

Since the total amount of lost space is a function of the block size and the number of files on the file system, the amount of lost space will increase proportionally to the increase in block size, for a given number of files on the file system.

This is a minor problem in the case of the server that hosts the HyperBrowser. A change in the way the genomic data is stored leading to a smaller number of files would lead to a slight decrease in lost space due to the effects of internal fragmentation.

\(^3\)General Parallel File System
3.5 Profiling in the HyperBrowser

In the HyperBrowser, one can turn on profiling by setting an option in one of the configuration files. When profiling is on, the HyperBrowser’s Profiler module is used to profile each Run. This module is a small wrapper around cProfile, which ultimately passes each Run through the runctx method of cProfile. Since this functionality is encapsulated in its own module, extending it is relatively easy.

3.6 MapReduce applied: Bins

In a given Run in the HyperBrowser there are two levels of splitting the data set and thus the computation of the results. This is done in a fashion resembling the MapReduce [10] algorithm. The parts the data set is split into are called Bins; CompBins and UserBins.

There is also two other technologies related to Binning; MemoBins and IndexBins. These are not used to split Tracks into smaller parts, but are used when looking up data (IndexBin), and for persisting results of analysis of parts of the subset to disk (MemoBin).

Figure 3.4 on the next page illustrates how these different types of partitioning of the data set and computation work together.

3.6.1 UserBins

The user can choose on the base pair level what parts of the genome he/she wants to do statistical analysis on. These selections are called UserBins, and are shown in green in Figure 3.4 on the following page. UserBins can go across several CompBins, and do not have to begin or end aligned with the start or end of a CompBin.

3.6.2 CompBins

CompBins are the main way of splitting the data set and computation. CompBins are shown as beige rectangles with blue borders in Figure 3.4 on the next page; here, the fact that UserBins are split into CompBins if possible, is illustrated.

Note that CompBins are aligned to the borders of IndexBins. This because currently the default size for IndexBins and CompBins in the HyperBrowser
are the same, but there is no reason why the sizes can’t differ. I investigate adjusting CompBin size separately from IndexBin size in Chapter 6 on page 60. The main point to take away from this though is the fact that CompBins have predetermined positions along the Tracks, and if a UserBin splits a CompBin, the CompBins stay aligned.

CompBins are intended to reduce memory usage by splitting the data set for a given Run into smaller pieces. By doing this, we eliminate the need to have the entire data set in memory the entire time of the Run, and can instead load only the necessary parts.

Furthermore, CompBins are necessary to allow the memoization of partial results from a Run through MemoBins, as described in 3.6.4.

3.6.3 IndexBins

IndexBins are like “bookmarks” in the genome. If we for instance have IndexBins for each 1000th base pair – an IndexBin size of 1000 – and we wanted to look up the 9890th base pair in the data set, we would first move a pointer to the 10000th base pair where there would be an index. Then we would go “backwards” until we hit the 9890th base pair. IndexBins are illustrated as black lines on top of the Track in Figure 3.4
3.6.4 MemoBins

MemoBins are a way of persisting the result of a computation to disk. Currently, MemoBins only work on the CompBin level; saving the result of the computation of each CompBin to disk. The general idea however is to extend what can be persisted to disk to include the results of computation on entire chromosomes.

MemoBins are basically a Python data structure containing several dicts which in turn contain the results themselves. This data structure is serialized to disk using pickle, a Python module for serializing objects to a file.

The use of MemoBins and disk memoization of results is a good idea in an algorithmical sense. However, MemoBinning is actually disabled in the current stable release of the HyperBrowser because most of the time, it does in fact result in higher run times. In these cases, it takes longer to read the memoized result from disk than it takes to simply re-calculate the result.
Chapter 4

Measuring Overhead

In section 2.4 on page 18 I mentioned two different types of overhead; computational overhead and programming language overhead. In the case of the HyperBrowser, it is relevant to look at both of these.

The HyperBrowser is designed based on a very object oriented architecture; tasks are separated cleanly into classes. While this is great for flexibility, maintainability and testing, it can have a negative impact on performance. Creating objects, passing objects back and forth, it all takes time, and it is only indirectly related to the statistical analysis that is done.

The main programming language used in the implementation of the HyperBrowser is Python. However, for just about all calculations and number crunching, numpy is used. Since numpy primarily is implemented in C, there is a distinct difference in run times between the Python level and numpy level of the HyperBrowser.

In this chapter I investigate ways to estimate overhead. I define models for estimating both computational overhead and programming language overhead, as described earlier.

4.1 Models for estimating overhead

I have developed several models to try to express how large a part of the total run time of a job in the HyperBrowser is computational overhead:
4.1.1 Using a function as a proxy for real computation

The HyperBrowser uses a large hierarchy of statistic classes for computing jobs. Each Statistic class has a `_compute` method in which all of its actual work is done. In this model I define overhead as anything that does not originate from a `_compute` method call. This is calculated using Equation 4.1.

\[
\text{overhead} = \frac{t_{\text{total}} - t_{\text{compute}}}{t_{\text{total}}}
\]  

(4.1)

Note that there is a lot of things besides computation going within these methods too; Statistics delegate some computations to other Statistics, so there might be a significant call graph originating from each `_compute` call.

4.1.2 Using third-party code as a proxy for real computation

The HyperBrowser makes extensive use of `numpy` [16] operations for doing its calculations. `numpy` provides vector operations on N-dimensional arrays at C-like speeds.

In this model I define overhead as everything that is not a call to a `numpy` function. This is based on the assumption that most of the run time spent in `numpy` code is time spent doing something useful, i.e. some sort of number crunching. This overhead is calculated using Equation 4.2.

\[
\text{overhead} = \frac{t_{\text{total}} - t_{\text{time numpy}}}{t_{\text{total}}}
\]  

(4.2)

4.1.3 Third-party code, revisited

This model is exactly like the model in 4.1.2, except that this model tries to account for the overhead caused by the context switch that happens when doing a call to a C function from Python, which is essentially what happens when a `numpy` function is called in Python.

To do this, we count the number of calls to `numpy` functions and multiply this number by an approximation of the run time of a “no-op” `numpy` call. Finally, this number is added to the part of the run time considered overhead. I used the run time of a call to `numpy.sum` on an empty `numpy` array to approximate a “no-op” `numpy` function call.
Equation 4.3 shows how overhead is calculated based on this model.

\[
\begin{align*}
    t_{\text{numpy-call}} &= t_{\text{no-op numpy call}} \times n_{\text{numpy-calls}} \\
    t_{\text{overhead}} &= (t - t_{\text{numpy}}) + t_{\text{numpy-call}} \\
    \text{overhead} &= \frac{t_{\text{overhead}}}{t}
\end{align*}
\] (4.3)

4.1.4 Relative performance; weighting operations

This model is an attempt at estimating the overhead inherent in Python compared to C or C++. To do this, we calculate a “hypothetical C++ run time” for the run in question. This is an estimate of the run time of the Run if it had been implemented in C++. This estimate is calculated by first getting the total Python run time for an operation in the given Run. This is shown in Equation 4.4.

\[
t_{\text{python}} = \sum t_i
\] (4.4)

Then we divide the total Python run time for each operation by a factor by which the given operation is slower in Python than C++ to get the total “hypothetical C++ run time”. This is shown in Equation 4.5.

\[
t_{\text{cpp}} = \sum \left( \frac{t_i}{f_i} \right)
\] (4.5)

These factors are obtained by timing Python operations and equivalent C++ operations, and dividing the Python run time by the C++ run time. Table 4.1 shows the factors for some operations. The implementation of the programs that time the Python and C++ operations is shown in appendix A on page 78.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Factor ((f))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function calls</td>
<td>48.61</td>
</tr>
<tr>
<td>Class instantiations</td>
<td>8.81</td>
</tr>
<tr>
<td>String concatenation</td>
<td>0.82</td>
</tr>
<tr>
<td>List access</td>
<td>12.02</td>
</tr>
<tr>
<td>numpy operations</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: Factors by which operations are slower in Python than C++. 

34
Weighted operations is calculated both with and without including numpy operations in the computation. This is because results show that with numpy operations, there is a strong bias towards the numpy operations because they represent a much larger part of the run time than all of the other operations combined.

4.2 Tools for estimating overhead in the HyperBrowser

I have implemented tools for calculating overhead based on each of the models explained above, on each Run in the HyperBrowser. These tools are implemented by extending the pstats.Stats class with functions that interrogate the datastructure that holds the profiling information. This information is then passed into functions that calculate overhead based on each model. Finally, the results are formatted and appended to the output of each Run.

Figure 4.1 on the next page shows how the output of a Run in the HyperBrowser looks. The output from the overhead tools can be seen under “Profiling information”, where they are formatted into a table. Some of models have different names in the table than the names I have used in this thesis:

- not _compute() is the function as proxy model.
- not numpy is the third party code as proxy model.
- not numpy, not call is the third party code, revisited model.

Weighted operations is also there, both with numpy calls included and excluded from the calculation. In addition, there are overhead calculations for object creations/class instantiations and function calls, which is something left over from before I implemented the weighted operations model which these are now part of.

4.3 Statistical analyses

I picked three different analyses, where two different Statistics are run on three different Tracks. These Tracks vary greatly in size, which should give us some interesting results. Below is a short overview of the three analyses involved.
Figure 4.1: The output of a Run, showing how overhead calculated based on each model is presented.

CompBin splitting is enabled
COMP_BIN_SIZE is 100000
INDEX_BIN_SIZE is 100000
LOAD_DISK_MEMORIZATION = False
STORE_DISK_MEMORIZATION = False

Global results table for:
The mean value of 'Meltmap (Melting)'

<table>
<thead>
<tr>
<th>Results</th>
<th>Global analysis</th>
<th>Local analysis</th>
<th>As track in history</th>
<th>Plot: histogram</th>
<th>Plot: values per bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>69.91</td>
<td>Html / Raw text</td>
<td>Load</td>
<td>N/A</td>
<td>Figure / Raw data</td>
</tr>
<tr>
<td>Assembly gap coverage</td>
<td>0.08999</td>
<td></td>
<td>Load</td>
<td>N/A</td>
<td>Figure / Raw data</td>
</tr>
</tbody>
</table>

Profiling information
Total run time: 2.571809 seconds
Profiling data (ostats.Stats dumped to file)
Call graphs
Somewhat stripped (PNG | SVG | dot)
Full (huge) (PNG | SVG | dot)

Overhead models

<table>
<thead>
<tr>
<th>Model</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>not in_compute()</td>
<td>43.795049%</td>
</tr>
<tr>
<td>not numpy</td>
<td>62.426137%</td>
</tr>
<tr>
<td>not numpy, not call</td>
<td>65.805494%</td>
</tr>
<tr>
<td>Calculation within threshold:</td>
<td>Yes</td>
</tr>
<tr>
<td>object creations</td>
<td>0.010093%</td>
</tr>
<tr>
<td>Calculation within threshold:</td>
<td>No</td>
</tr>
<tr>
<td>function calls</td>
<td>0.089563%</td>
</tr>
<tr>
<td>Calculation within threshold:</td>
<td>No</td>
</tr>
<tr>
<td>python/c++ weighted operations</td>
<td>0.330748%</td>
</tr>
<tr>
<td>Ratio of total run time used:</td>
<td>37.81478%</td>
</tr>
<tr>
<td>python/c++ weighted operations (without numpy)</td>
<td>30.819088%</td>
</tr>
<tr>
<td>Ratio of total run time used:</td>
<td>0.407615%</td>
</tr>
</tbody>
</table>
4.3.1 MeanStat on Meltmap

MeanStat calculates the mean value of a Track. The Track in question in this case is Meltmap, a function type Track that has about 3 billion elements.

4.3.2 CountPointStat on Sequence, Repeating elements as Segments

CountPointStat counts the base pairs of the Track. In analysis, the Track used is Sequence, Repeating elements. This is a segment type Track, which means that the segments have to be converted to points to be counted by CountPointStat. This is done by interpreting the middle of each segment as a point. The number of segments in the Sequence, Repeating elements Track is in the range of a few million.

4.3.3 CountPointStat on Genes as Segments

This analysis uses the same Statistic as above, but runs it on a different Track; Genes. This Track is of type segments, so it too has to be converted to points in the same fashion as above. This Track only has about 30000 segments though, so it is a significantly smaller data set.

4.4 Results

All analyses are run with IndexBin size set to 100k and UserBin size set to 1M. Note that the leftmost column in the tables below show what percentage of the total run time were used to calculate the “weighted operations” overheads.

4.4.1 Comparing overhead estimates across Runs

It might at first glance seem that comparing overhead estimates between two Runs with different Run times is a good idea. However, looking at these estimates in the context of the run time of the Runs they are based on might give us a better impression of how overhead changes between different runs.

Let’s say that our hypothesis is that CompBin size 1M is faster and more efficient (less overhead) than 100k, and we want to investigate the effect the change in CompBin size has on overhead. Equation 4.6 shows how we
calculate the change in overhead as a factor, from the 100k CompBin size Run to the 1M CompBin size Run.

\[
\Delta \text{overhead}_{\text{factor}} = \frac{t_{100k} \times \text{overhead}_{100k}}{t_{1M} \times \text{overhead}_{1M}}
\]  
(4.6)

In addition to calculating a factor of change in overhead, we can also look at the raw change in the run time we define as overhead between each of the runs. Equation 4.7 shows how this is calculated.

\[
\Delta \text{overhead}_{\text{time}} = (t_{100k} \times \text{overhead}_{100k}) - (t_{1M} \times \text{overhead}_{1M})
\]  
(4.7)

By using these methods for comparison, we can look at the change in the run time we define as overhead between the Runs instead of the percentage of the run time of each Run. This makes sense, because the run times of each Run can differ significantly.

**Use with weighted operations** Note that since these calculations are based on the total run time of each Run, we won’t get numbers that make sense for the “weighted operations” model. This is because the estimates calculated using the “weighted operations” model aren’t themselves based on the total run time of each Run. Passing a “weighted operations” estimate into Equation 4.7 or 4.7 will therefore not give us numbers that are meaningful.

**MeanStat on Meltmap** The overhead measurements of the MeanStat on Meltmap Run is shown in Table 4.2 on the next page.

From this table we can read that all overhead models except for “weighted operations excluding numpy” show a slight decrease in overhead when the CompBin size is changed from 100k to 1M. Furthermore, note that the difference between “function as a proxy” and “third party code as a proxy” is larger at the lower CompBin size. This could be because the lower CompBin size means that there are more Bins, thus more time is spent doing bookkeeping; things not directly related to the computation.

The “weighted operations” model shows very low overhead when numpy operations are included. If we don’t count numpy operations however, the overhead estimate we get is based on a very small part of the total run time. This means that the estimate doesn’t necessarily translate well to the rest of the run time.

Table 4.2c on the following page shows the change in overhead, calculated as outlined in Section 4.4.1 on the previous page, from the Run of Meanstat on
Meltmap at CompBin size 100k to the Run at CompBin size 1M. We have already established that the change in overhead from the first Run to the second looks like a slight decrease based on the “function as proxy”, “third party code as proxy” and “third party, revisited” models. In Table 4.2c we can see that there is actually a two to three fold decrease in overhead from the Run using CompBin size 100k to the Run using CompBin size 1M.

<table>
<thead>
<tr>
<th>Model</th>
<th>Overhead</th>
<th>Based on % of run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function as proxy</td>
<td>56.04%</td>
<td>n/a</td>
</tr>
<tr>
<td>Third party code as proxy</td>
<td>71.77%</td>
<td>n/a</td>
</tr>
<tr>
<td>Third party code, revisited</td>
<td>74.54%</td>
<td>n/a</td>
</tr>
<tr>
<td>Weighted operations</td>
<td>0.39%</td>
<td>28.59%</td>
</tr>
<tr>
<td>Weighted operations (excluding \texttt{numpy})</td>
<td>31.31%</td>
<td>0.35%</td>
</tr>
</tbody>
</table>

(a) CompBin size 100k.

<table>
<thead>
<tr>
<th>Model</th>
<th>Overhead</th>
<th>Based on % of run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function as proxy</td>
<td>46.41%</td>
<td>n/a</td>
</tr>
<tr>
<td>Third party code as proxy</td>
<td>46.89%</td>
<td>n/a</td>
</tr>
<tr>
<td>Third party code, revisited</td>
<td>47.80%</td>
<td>n/a</td>
</tr>
<tr>
<td>Weighted operations</td>
<td>0.35%</td>
<td>53.36%</td>
</tr>
<tr>
<td>Weighted operations (excluding \texttt{numpy})</td>
<td>74.03%</td>
<td>0.25%</td>
</tr>
</tbody>
</table>

(b) CompBin size 1M.

<table>
<thead>
<tr>
<th>Model</th>
<th>(\Delta)Overhead as factor</th>
<th>(\Delta)Overhead as time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function as proxy</td>
<td>2.52</td>
<td>1.06 s</td>
</tr>
<tr>
<td>Third party code as proxy</td>
<td>3.19</td>
<td>1.54 s</td>
</tr>
<tr>
<td>Third party code, revisited</td>
<td>3.30</td>
<td>1.62 s</td>
</tr>
<tr>
<td>Weighted operations</td>
<td>2.33</td>
<td>0.007 s</td>
</tr>
<tr>
<td>Weighted operations (excl. \texttt{numpy})</td>
<td>0.88</td>
<td>-0.13 s</td>
</tr>
</tbody>
</table>

(c) The reduction (delta) in overhead measured in time for each overhead model from the Run at CompBin size 100k to the Run at CompBin size 1M. The total run times were 3.13s for CompBin size 100k and 1.5s for 1M.

Table 4.2: Overhead measurements for MeanStat on Meltmap.

\textbf{CountPointStat on Sequence, Repeating elements}  This Run was also was also run with CompBin size set to 100k and 1M. The results of these Runs are shown in Table 4.3 on page 41. On the contrary to the results from MeanStat on Meltmap, these results indicate that the overhead for “function as proxy” and “third party code as proxy” actually increase from the Run with CompBin size 100k to the Run with CompBin size 1M.
The “third party code, revisited” model shows overhead above 100% for both Runs. This obviously does not make sense at all; we can’t use more than all of the run time doing things only loosely related to the run time.

In these Runs, the “weighted operations” model is based on a quite small part of the total run time both with and without numpy operations. The estimate including numpy operations is a bit better than the one without numpy operations. However, the estimates we get are not necessarily representative for the rest of the total Run time.

Table 4.3c on the next page, which shows the change in overhead, tells different story though. By these metrics it looks like there is in fact a two to three fold decrease in overhead based on the “function as proxy” and “third party code as proxy” models.

CountPointStat on Genes

The results for the CountPointStat on Genes analysis is shown in Table 4.4 on page 42. From this table we observe what looks like a quite significant increase (almost 2x) in the overhead calculated using “function as proxy” when going from CompBin size 100k to 1M. The same change causes a slight increase in overhead using the “third party code as proxy” model. The “Third party code, revisited” model gives us overhead above 100% in both Runs, just like in the CountPointStat on Sequence Runs above.

The estimates based on the “weighted operations” model shows the same flaws as in the other two analyses; the run time basis of the estimates are very low.

By looking at the overhead delta metrics in Table 4.4c on page 42, we can see that what looked like a doubling in overhead for the “function as proxy” model is in fact a decrease in overhead by a factor of more than two. The reduction in overhead shown by the “third party code as proxy” model is even higher, at a factor of almost 3.5.

4.5 Discussion

Comparing overhead estimates across Runs

Overhead estimates are certainly useful when looking at a single Run. However, comparing overhead estimates between different Runs with different run times directly doesn’t accurately show what is going on. This is covered in detail in Section 4.4.1 on page 37.

This way of showing change in overhead between two Runs does not work that well for the “weighted operations” models though. This is because
Model Overhead Based on % of run time

Function as proxy 39.54% n/a
Third party code as proxy 91.51% n/a
Third party code, revisited 114.47% n/a
Weighted operations 27.58% 12.37%
Weighted operations (excluding **numpy**) 88.03% 3.88%

(a) CompBin size 100k.

<table>
<thead>
<tr>
<th>Model</th>
<th>Overhead</th>
<th>Based on % of run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function as proxy</td>
<td>56.13%</td>
<td>n/a</td>
</tr>
<tr>
<td>Third party code as proxy</td>
<td>97.54%</td>
<td>n/a</td>
</tr>
<tr>
<td>Third party code, revisited</td>
<td>104.66%</td>
<td>n/a</td>
</tr>
<tr>
<td>Weighted operations</td>
<td>15.06%</td>
<td>2.97%</td>
</tr>
<tr>
<td>Weighted operations (excluding <strong>numpy</strong>)</td>
<td>87.18%</td>
<td>0.51%</td>
</tr>
</tbody>
</table>

(b) CompBin size 1M.

<table>
<thead>
<tr>
<th>Model</th>
<th>ΔOverhead as factor</th>
<th>ΔOverhead as time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function as proxy</td>
<td>2.18</td>
<td>0.66 s</td>
</tr>
<tr>
<td>Third party code as proxy</td>
<td>2.90</td>
<td>1.85 s</td>
</tr>
<tr>
<td>Third party code, revisited</td>
<td>3.38</td>
<td>2.49 s</td>
</tr>
<tr>
<td>Weighted operations</td>
<td>5.66</td>
<td>0.70 s</td>
</tr>
<tr>
<td>Weighted operations (excl. <strong>numpy</strong>)</td>
<td>3.12</td>
<td>1.85 s</td>
</tr>
</tbody>
</table>

(c) The reduction (delta) in overhead measured in time for each overhead model from the Run at CompBin size 100k to the Run at CompBin size 1M. The total run times were 3.09s for CompBin size 100k and 1.0s for 1M.

Table 4.3: Overhead measurements for CountPointStat on Sequence, Repeating elements as Segments.
<table>
<thead>
<tr>
<th>Model</th>
<th>Overhead</th>
<th>Based on % of run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function as proxy</td>
<td>45.78%</td>
<td>n/a</td>
</tr>
<tr>
<td>Third party code as proxy</td>
<td>91.24%</td>
<td>n/a</td>
</tr>
<tr>
<td>Third party code, revisited</td>
<td>114.81%</td>
<td>n/a</td>
</tr>
<tr>
<td>Weighted operations</td>
<td>23.65%</td>
<td>12.05%</td>
</tr>
<tr>
<td>Weighted operations (excluding <strong>numpy</strong>)</td>
<td>86.68%</td>
<td>3.29%</td>
</tr>
</tbody>
</table>

(a) CompBin size 100k.

<table>
<thead>
<tr>
<th>Model</th>
<th>Overhead</th>
<th>Based on % of run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function as proxy</td>
<td>79.98%</td>
<td>n/a</td>
</tr>
<tr>
<td>Third party code as proxy</td>
<td>96.79%</td>
<td>n/a</td>
</tr>
<tr>
<td>Third party code, revisited</td>
<td>105.57%</td>
<td>n/a</td>
</tr>
<tr>
<td>Weighted operations</td>
<td>16.23%</td>
<td>3.94%</td>
</tr>
<tr>
<td>Weighted operations (excluding <strong>numpy</strong>)</td>
<td>87.81%</td>
<td>0.73%</td>
</tr>
</tbody>
</table>

(b) CompBin size 1M.

<table>
<thead>
<tr>
<th>Model</th>
<th>ΔOverhead as factor</th>
<th>ΔOverhead as time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function as proxy</td>
<td>2.11</td>
<td>0.64 s</td>
</tr>
<tr>
<td>Third party code as proxy</td>
<td>3.48</td>
<td>1.71 s</td>
</tr>
<tr>
<td>Third party code, revisited</td>
<td>4.01</td>
<td>2.27 s</td>
</tr>
<tr>
<td>Weighted operations</td>
<td>5.38</td>
<td>0.51 s</td>
</tr>
<tr>
<td>Weighted operations (excluding <strong>numpy</strong>)</td>
<td>3.64</td>
<td>1.66 s</td>
</tr>
</tbody>
</table>

(c) The reduction (delta) in overhead measured in time for each overhead model from the Run at CompBin size 100k to the Run at CompBin size 1M. The total run times were 2.62s for CompBin size 100k and 0.71s for 1M.

Table 4.4: Overhead measurements for CountPointStat on Genes as Segments.
overhead estimates calculated using this model are only based on a part of the total run time, not the full run time.

**Third party code, revisited** Based on the measurements of the two CountPointStat analyses, it definitely looks like the “third party code, revisited” model more or less consistently overestimates the run time.

This model tries to account for the overhead in calling C functions from Python by multiplying a “numpy call cost” with the number of numpy calls. This is explained in detail in Section 4.1.3 on page 33. The reason why the estimates are high is probably that the approximation of the run time of a “no-op” numpy call that is used in this model is too high. This run time approximation doesn’t even have to be off by much; since it is multiplied by the number of numpy calls, which can be a relatively large number, it adds up quickly.

**Weighted operations** A benefit of the “weighted operations” model over the other models is that in principle, we don’t need to know anything about the program we want to estimate overhead in to implement the model. The estimates we get by excluding numpy operations are completely general. To estimate overhead for the HyperBrowser, it makes sense to include numpy operations because they are an integral part in how the HyperBrowser work. This is a way of tailoring the general model to the application, resulting in a more or less application specific model which should give a more accurate estimate.

While all this sounds great in theory, the run time used when calculating estimates using the “weighted operations” model is consistently a small part of the total run time. Because of this, the estimates we get are not necessarily representative across the total run time of the Run. Look at the estimate from “weighted operations (excluding numpy)” in Table 4.2a on page 39 for instance. What this estimate says is that 31.31% of the run time is overhead, but the run time used here is actually only 0.35% of the total run time. So what does this estimate tell us about overhead in the entire run? Not very much.

Another, related problem is that when including numpy operations, the estimates the model create are heavily biased towards the numpy operations. This is because numpy operations are a significantly larger part of the total run time than all of the other operations used in the model combined.

The extent of both of these problems can be reduced by adding overhead factors for more operations to the model. This will make the estimates calculated using the model more representative towards the rest of the total run
time, and it will reduce the bias towards numpy operations in the resulting estimates.

**Inaccuracy inherent in the profiler**  When profiling Python code, cProfile incurs an overhead on the code we are profiling, which makes the run times we measure artificially slow. This is like the observer effect; when the fact that we are observing something changes the behavior of what we are observing. This effect is more pronounced when measuring small, fast function calls, because the profiler overhead becomes a larger part of the time we measure than in the case of a slow function call. If we for instance have a small function that is called a large number of times, the profiler overhead starts to add up in the total time spent in the function. Things like this make the overhead calculations less accurate, and this contributes to the cases where overhead percentages to go above 100% of the run time.

**So is there a way of estimating how large a part of a program’s run time is overhead?**  To answer the research question in 1.3 on page 10; yes, there is. Making the estimate accurate is a bit of a challenge though. To get an accurate estimate of overhead in a program, it is useful to use a overhead model that is specific to the program you are analyzing, rather than a general model. It isn’t necessarily easy to know when a model gives an accurate estimate and when something is off either, which further complicates the issue.
Chapter 5

Run time optimizing large Python programs

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.

William Allan Wulf

5.1 Tools for finding bottlenecks

The distribution of run time across the various modules that comprise your program is often unevenly distributed. Sometimes there will be a small number of modules or functions that are responsible for some large percentage of the run time of your program. These modules or functions are then referred to as hot spots or a bottlenecks with regards to run time.

Before trying to optimize a program for run time, it is important to find the bottlenecks (if any) of said program. Attempting to optimize the parts of the program that by intuition or assumption are bottlenecks is a surefire way to waste your time. Always measure before optimizing.

5.1.1 Profiling in the HyperBrowser

Before I started working on this thesis, all the functionality the HyperBrowser’s Profiler module provided in the way of presenting the results of the profiling was printing the raw profiler output below the results of the Run that was profiled.
I think good tools are important, so I decided to improve the tooling around the Profiler module in the HyperBrowser. As part of that effort, I have implemented two tools for finding bottlenecks in Python programs based on output from cProfile.

5.1.2 Visualizing the total run time spent inside a function

One interesting metric with regards to bottlenecks on a per function level, is the percentage of the total run time spent inside a given function. By “time spent inside” a function I mean the total time a call of the function was the active stack frame. This metric is formalized in Equation 5.1.

\[
\text{percentage} = \frac{t_{\text{active}}}{t_{\text{total}}} \times 100
\]

(5.1)

This tool calculates the percentage of total run time for each function in the call graph, then sorts it by percentage. This plot is intended as a quick indicator of whether there are any significant bottlenecks in a program or not.

An example of a plot like this is shown in Figure 5.1 on the next page. This plot is from profiling data from a specific Run in the HyperBrowser. Each dot in the scatter plot is a function. The value on the Y axis for each function is the percentage calculated by Equation 5.1. These values are then sorted in descending order across the X axis, to give an impression of how the run time is distributed across functions. The values on the X axis thus have no real meaning except showing how many functions there are, which is not terribly useful to know in the first place.

The useful information we can read from the plot is that there are no single functions responsible for more than 5.5% of the total run time. Furthermore; by and large, functions spend less than 2% of the total run time as the active stack frame, with only 8 functions between 2% and 5.5%. While these aren’t big bottlenecks, investigating what functions they are and why they are responsible for such a large amount of the run time might be lead to better (lower) run time.

5.1.3 Visualizing the call graph

The functions in a program can be modeled as a directed graph with functions as nodes and function calls as edges; a call graph. Thus, a good way of getting a detailed view of what is happening inside a program is to somehow visualize that call graph.
Figure 5.1: An example of a bottleneck plot.
Existing tools

There already exists a set of tools that in combination lets you do exactly this already. First, there is gprof2dot [12] which parses the output of the python profilers as well as several other profilers, and converts it into a DOT [3] graph. DOT is a domain-specific language for describing graphs: directed and undirected. The DOT graph can then be viewed in an interactive DOT graph viewer like ZGRViewer [18], or rendered into an image using the dot command-line tool, which is part of the Graphviz [11] graph drawing package.

There is a toolchain available, that is great. However, it is important that tools are easy to use, so I decided to integrate these tools better with the HyperBrowser. Rendering a call graph through this toolchain means downloading the dumped pstats data from the results of your Run in the HyperBrowser web application, navigating to the file on the command-line, and running some shell command. I wanted to just get the rendered call graph right there in the results of the Run, alongside the rest of the profiling data.

Integration

To implement this, I had to hook into the gprof2dot program to get at the classes that can parse cProfile output. While it doesn’t seem like gprof2dot is designed to be used in this fashion, it actually was a relatively straightforward job. I wrote a fairly small function that parses the output from cProfile, and then writes the DOT graph to a file on the HyperBrowser server machine.

After gprof2dot has written the DOT graph, we still need to render the dot graph as an image. To do this I used the Python bindings to Graphviz, PyGraphviz [15], to write a function that rendered the DOT graph as both a PNG (raster) image and a SVG (vector) image.

When this was done, I modified the Run result writing subsystem to create download links to the resulting DOT, PNG, and SVG files in each profiled Run.

The resulting modification creates the exact same graphs as gprof2dot combined with dot on the command-line. These call graphs are nice, but they don’t really translate well to the printed page, because they have to be fairly large to make the text in the nodes readable, and the edges visible. Thus I have had to only include subsections of the call graphs in this thesis, with
links to the full graph online. Figure 5.2\textsuperscript{1} is an example of how a subsection of a callgraph can look.

![Subsection of a visualized call graph.](http://oyvining.at.ifi.uio.no/m/callgraph-example.png)

Figure 5.2: Subsection of a visualized call graph.

The information printed on the nodes in the graph is structured as follows:

- **module name**:line number**:function name
- percentage of total run time spent in function, including subcalls
- percentage of total run time spent in function, excluding subcalls
- number of calls

\textsuperscript{1}Full graph for Figure 5.2: http://oyvining.at.ifi.uio.no/m/callgraph-example.png
This tool is a nice augmentation to the bottleneck plot, as the bottleneck plot does not tell us much beyond the distribution of slow or often called functions and fast or seldom called functions. The call can help get an overview of where the run time of your program is spent. This should make it easier to locate bottlenecks and performance related bugs.

5.2 Finding and handling a bottleneck

This is an example of how the tools mentioned in Section 5.1.2 on page 46 can be used in practice to determine how the run time is distributed across the functions of your program.

I investigate a specific Run in the HyperBrowser, but these tools and this method of using them can be applied to any program. During this analysis, I locate a significant hot spot/bottleneck in the code, and end up resolving a performance issue.

5.2.1 The Run

The Run in question is a CountPointStat Run across chromosome 1 of the human genome. The CountPointStat Statistic counts the number of points in the Track. The Track is stored as segments though, which means it has to be converted to points. This is done by interpreting the beginning of each segment as a point.

CompBin splitting is enabled, with 100k bp CompBin size, resulting in about 2500 Bins/atoms, since the length of the human genome’s chromosome 1 is approximately 250 million bp.

5.2.2 Bottlenecks

In the bottleneck plot in Figure 5.3 on the following page, it looks like one function is a significant bottleneck, accounting for approximately 36% of the total run time. This warrants further investigation.

5.2.3 Profiling

A portion of the output from cProfile for the Run through the pstats command-line interface is listed below. It shows a listing of each function that was invoked in the Run, sorted by internal time; that is time spent inside
Visualizing bottlenecks

Function calls

Percentage of total run time as active stack frame

0 5 10 15 20 25

Figure 5.3: Bottleneck plot of the Run described in Section 5.2.1 on the previous page.
each function. For brevity’s sake I’m only listing the top five functions, and irrelevant parts of the file paths are removed.

2112572 function calls (1275636 primitive calls) in 3.471 seconds
Ordered by: internal time

    ncalls  tottime  cumtime filename:lineno(function)
   801252/2473  1.141  1.141 (...)/gold/track/VirtualNumpyArray.py:24(__getattr__)
     136075  0.183  1.324 {hasattr}
       49460  0.094  0.113 (...)/numpy/core/memmap.py:254(__array_finalize__)
        7419  0.088  0.407 (...)/gold/track/TrackView.py:124(__init__)
        2473  0.074  0.074 {posix.listdir}
        7443  0.065  0.065 {method '__reduce_ex__' of 'object' objects}

From the profiler output we can see that the hot spot is in the function __getattr__ at line 24 in the module gold.track.VirtualNumpyArray.

__getattr__ is one of Python’s “magic methods”, described in 2.2.3 on page 14. This particular method is invoked on an instance when an attribute reference is done, like in the statement baz = foo.bar, where baz is set to the bar attribute of the instance foo.

The gold.track.VirtualNumpyArray module is a low-level interface to the raw Track data, which are stored as numpy arrays. In fact, four of the five functions listed below is directly related to the data handling subsystem; the exception being hasattr. hasattr is a Python built-in function that uses introspection to check if an object has an attribute by some name.

    hasattr(object, name)

    The arguments are an object and a string. The result is True if the string is the name of one of the object’s attributes, False if not. (This is implemented by calling getattr(object, name) and seeing whether it raises an exception or not.)

    (The Python Standard Library Documentation)

One thing is that a part of the data handling subsystem is slow, but another thing is that it is a built-in function that does such a seemingly simple thing is responsible for such a large part of the run time of a Run. Inspecting the call graph might yield a better understanding of what is going on.

5.2.4 Call graph

Using the call graph makes it easier to understand the flow of control. Figure 5.4 on the following page\(^2\) shows the section of the call graph showing the

\(^2\) Full graph for Figure 5.4 on the next page: http://oyvining.at.ifi.uio.no/m/cps-seg-prefix-callgraph-full.svg
hasattr and VirtualNumpyArray: \_\_getattr\_\_ calls. From this we can see that hasattr calls VirtualNumpyArray: \_\_getattr\_\_, which in and of itself is responsible for almost 33% of the run time.

Figure 5.4: Section of the call graph of the Run described in Section 5.2.1 on page 50 showing the hasattr and VirtualNumpyArray: \_\_getattr\_\_ calls.

The green edge entering the left of the hasattr node comes from the function shown in Figure 5.5 on the following page\(^3\), VirtualNumpyArray: \_\_sub\_\_. hasattr is apparently called 2473 times by this function.

5.2.5 Diving into the code

Initial state

It appears that a good place to start is by looking at the gold.track.VirtualNumpyArray module. This module has a single class VirtualNumpyArray, which is shown in Listing 5.1 on the following page. This class acts as a lazy loading proxy for a numpy array. The lazy loading is done through dynamically adding a member variable \_\_cachedNumpyArray\_, with a reference to a numpy array,

\(^3\)Full graph for Figure 5.5 on the next page: http://oyvining.at.ifi.uio.no/m/cps-seg-prefix-callgraph-full.svg
but not before said member variable is needed. This is implemented through
the snippet in lines 6 and 7, which is repeated in the \_\_sub\_\_ and \_\_add\_\_ methods as well. \texttt{VirtualNumpyArray} is used as an abstract class – i.e.
it depends on subclasses to provide a method returning a reference to the
\texttt{numpy} array it is supposed to cache.

Listing 5.1: The \texttt{gold.track.VirtualNumpyArray} module in its original
state.

```python
class VirtualNumpyArray(object):
    def \_\_getattr\_\_(self, name):
        if name == \ '_cachedNumpyArray'\_\_:
            return self._cachedNumpyArray
        if not hasattr(self, \ '_cachedNumpyArray'\_\_):
            self._cachedNumpyArray =
            self._asNumpyArray()
            return getattr(self._cachedNumpyArray, name)

    def \_\_sub\_\_(self, other):
        if not hasattr(self, \ '_cachedNumpyArray'\_\_):
            self._cachedNumpyArray =
            self._asNumpyArray()
            return self._cachedNumpyArray._\_sub\_\_(other)

    def \_\_add\_\_(self, other):
        if not hasattr(self, \ '_cachedNumpyArray'\_\_):
            self._cachedNumpyArray =
            self._asNumpyArray()
            return self._cachedNumpyArray._\_sub\_\_(other)
```

Figure 5.5: Section of the call graph of the Run described in Section 5.2.1
on page 50 showing the \texttt{VirtualNumpyArray:__sub__} calls.
self._cachedNumpyArray =
self._asNumpyArray()
return self._cachedNumpyArray.__add__(other)

Attempt at improving performance

Apparently the use of `hasattr` in the fashion it is used in Listing 5.1 on the previous page is slow, so I changed the `VirtualNumpyArray` class to avoid using it. I did this in a way I thought was elegant and “pythonic”. The result is shown in Listing 5.2.

Listing 5.2: The `gold.track.VirtualNumpyArray` module after first try at resolving the performance issue.

class VirtualNumpyArray(object):
    @property
    def _cachedNumpyArray(self):
        try:
            return self._actualNumpyArray
        except AttributeError:
            self._actualNumpyArray =
                self._asNumpyArray()
        return self._actualNumpyArray

def __getattr__(self, name):
    if name == '_cachedNumpyArray':
        return self._cachedNumpyArray
    else:
        return getattr(self._cachedNumpyArray, name)

def __sub__(self, other):
    return self._cachedNumpyArray.__sub__(other)

def __add__(self, other):
    return self._cachedNumpyArray.__add__(other)

After modifying the class, deploying the code and doing another Run, the profiling results showed an even worse run time. The thing is, the code as it is in Listing 5.2 does the same thing as the code in Listing 5.1 on the preceding page: `hasattr` calls `getattr` and checks for an exception. The exact same thing is done in the `_cacheNumpyArray` method in Listing 5.2.
A subsection of the call graph for the Run with this code is shown in Figure 5.6. Here, we can see that this class is still responsible for a significant amount of the total run time.

Figure 5.6: Section of the call graph of the Run described in Section 5.2.1 on page 50 showing the VirtualNumpyArray:__getattr__ calls.

Keep it simple

To avoid excessive overhead in the Python dynamic features, I modified the VirtualNumpyArray class to do just about the simplest thing possible.

\footnote{Full graph for Figure 5.6: http://oyvining.at.ifi.uio.no/m/cps-seg-midfix-vna.svg}
The resulting code is shown in Listing 5.3. The _cachedNumpyArray is set to None in the object initializer (.__init__). The lazy loading is done by checking if _cachedNumpyArray is None, and loading the numpy array if that is the case.

Listing 5.3: The gold.track.VirtualNumpyArray module after resolving the performance issue.

```python
class VirtualNumpyArray(object):
    def __init__(self):
        self._cachedNumpyArray = None

    def _cacheNumpyArray(self):
        if self._cachedNumpyArray is None:
            self._cachedNumpyArray = self._asNumpyArray()

    def __getattr__(self, name):
        if name == '_cachedNumpyArray':
            self._cacheNumpyArray()
        return self._cachedNumpyArray

    def __sub__(self, other):
        self._cacheNumpyArray()
        return self._cachedNumpyArray.__sub__(other)

    def __add__(self, other):
        self._cacheNumpyArray()
        return self._cachedNumpyArray.__add__(other)
```

5.2.6 Results

Table 5.1 on the next page shows how the changes in Section 5.2.5 on page 53 affects the run time of the Run in Section 5.2.1 on page 50. The run time is in seconds, and the change in run time is in relation to the run time of the Run with the code in its initial state, i.e. the first line in the table.

In the end, the run time decreased by 36.02%, which is quite respectable for something I would consider low-hanging fruit, in the context of improving performance. This number also corresponds well with the run time
<table>
<thead>
<tr>
<th>Code</th>
<th>Run time</th>
<th>Change in run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 5.1</td>
<td>3.47 s</td>
<td>-</td>
</tr>
<tr>
<td>Listing 5.2</td>
<td>4.21 s</td>
<td>+21.33%</td>
</tr>
<tr>
<td>Listing 5.3</td>
<td>2.22 s</td>
<td>-36.02%</td>
</tr>
</tbody>
</table>

Table 5.1: Run times of the Run in Section 5.2.1 on page 50 in various states of the code.

of VirtualNumpyArray:__sub__ seen in the call graph in Figure 5.5 on page 54.

While this number is only valid for this specific Run, the VirtualNumpyArray class is, among other things, used whenever a Track of segments are converted to points, which is done in a significant amount of Runs.

**Bottleneck**

In Figure 5.7 on the next page we can see that the hot spot/bottleneck from the earlier plot in Figure 5.3 on page 51 is now gone.

**Profiling**

Profiling shows the current bottlenecks, which aren’t really responsible for that large a part of the run time. It is a bit strange that `hasattr` comes out on top, but it might just be because it is called so often (133602 times).

1321212 function calls (1283055 primitive calls) in 2.220 seconds
Ordered by: internal time

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>cumtime</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>133602</td>
<td>0.102</td>
<td>0.102</td>
<td>{hasattr}</td>
</tr>
<tr>
<td>7419</td>
<td>0.093</td>
<td>0.412</td>
<td>(...)/gold/track/TrackView.py:124(<strong>init</strong>)</td>
</tr>
<tr>
<td>49460</td>
<td>0.088</td>
<td>0.107</td>
<td>(...)/numpy/core/memmap.py:254(<strong>array_finalize</strong>)</td>
</tr>
<tr>
<td>2473</td>
<td>0.072</td>
<td>0.072</td>
<td>{posix.listdir}</td>
</tr>
<tr>
<td>7443</td>
<td>0.064</td>
<td>0.064</td>
<td>{method '<strong>reduce_ex</strong>' of 'object' objects}</td>
</tr>
</tbody>
</table>
Figure 5.7: Bottleneck plot of the Run described in Section 5.2.1 on page 50, after removing a bottleneck.
Chapter 6

Being conscious about granularity at the Python and file level

6.1 Balancing flexibility and speed

When using computational strategies like divide and conquer or map/reduce, in the case of the HyperBrowser, it is important to consider the granularity at which the computation is done.

To draw an analogy, think about the way a country is governed; everything is not micromanaged from one place. Norway, for instance, is split into 19 counties, which are in turn split into 430 municipalities. The parliament cannot make rulings on every minute issue in every municipality; there are so many issues in so many municipalities that there would be no time to govern the country at a macro level. This is why administrative control is deferred to counties and from counties further down to municipalities.

This is transferrable in some sense to the way computation is split up in the map phase of the map/reduce process in the HyperBrowser. Bins, as described in Section 3.6 on page 29, can be said to be the smallest atomic unit in this context. There is another layer of splitting in the HyperBrowser though; the file layer. The way data is handled in the HyperBrowser is described in Section 3.4 on page 23. On the file layer, the chromosome is the smallest atomic unit.

The granularity at which we make these splits can easily have a very significant effect on the run time of our program. The extent of this effect is what I investigate in this chapter.
6.2 Granularity at the Python level

6.2.1 Controlling overhead in Bins

The HyperBrowser makes heavy use of the concept of CompBins and UserBins, for splitting the computation and data sets involved in a Run into smaller parts. This is discussed in detail in Section 3.6 on page 29.

These Bins work like proxies to subsets of the Track data, and are responsible for doing computation on their respective subsets. Figure 6.1 illustrates how CompBins proxy 100k base pair long \texttt{numpy} arrays on a 400k base pair long Track.

![Figure 6.1: A Track, split into four CompBins, showing how these proxy \texttt{numpy} objects](image)

When splitting the data set like this, it is important that any ancillary work, or “bookkeeping” done in each Bin is limited and as efficient as possible. This is important because any work done in a Bin is repeated for each Bin, and the total amount of work can add up to a significant amount of the run time when the number of Bins increase.

**How many Python operations can we do for a given \texttt{numpy} operation?**

In the HyperBrowser, the Bins form the layer merging the Python and \texttt{numpy} level of the computation part of the system. Because of this, it is interesting to investigate the speed of \texttt{numpy} operations relative to the speed of Python operations.

To do this, I wrote a small program that sums \texttt{numpy} arrays and Python lists of increasing sizes, measures the run time and finally plots the results.
The interesting part about these plots is that the unit on the Y axis is not measured in seconds or minutes, but Python operations; function calls or class instantiations. This is to attempt to measure the amount of Python operations we can do in a Bin before the Python run time begins to exceed the numpy run time.

**Results**

In the plots in Figure 6.2 on the next page the summing of Python and numpy lists with lengths between 1 and 10000 in the first plot, and 1 and 1 million in the second plot, are measured against the run time of Python function calls. From these plots we can read that summing a 10000 elements long numpy array is equivalent to doing 100 Python function calls. If we were summing a regular Python list, it can’t be much more than 2200 elements long, else the run time would be longer. Furthermore, the run time of summing a 1M elements long numpy array is equivalent to about 6400 Python function calls.

In the plots in Figure 6.3 on page 64 the summing of Python and numpy lists with lengths between 1 and 10000 in the first plot, and 1 and 1M in the second plot, are measured against the run time of Python class instantiations. From these plots we can read that summing a 10000 elements long numpy array is equivalent to about 55 Python class instantiations. Within that same time, we only have time to sum an approximately 2100 elements long Python list. Furthermore, the run time of summing a 1M elements long numpy array is equivalent to about 4000 Python class instantiations.

Another interesting observation is that summing numpy arrays is actually faster than summing Python lists at list sizes as small as 6-700 elements.

6.2.2 Measuring the effect of binning on run time

To measure the effect of binning on run time I investigated the run times of the same three statistical analyses used in Chapter 4. These are described in detail in Section 4.3 on page 35.

These three different statistical analyses are run at two different IndexBin sizes, four different UserBin sizes and three different CompBin sizes. In addition, they are run with CompBin splitting disabled; this effectively makes the CompBin size the length of the data set. All in all, this is a total of 96 different configurations; 32 per analysis.
(a) List lengths from 1 to 10000.

(b) List lengths from 1 to 1M.

Figure 6.2: Measuring the run time of summing numpy and Python lists in number of Python function calls.
Figure 6.3: Measuring the run time of summing numpy and Python lists in number of Python class instantiations.
This analysis is an effort to formalize the way the different types of binning that the HyperBrowser supports affect run time. We already know that running an analysis with CompBin splitting enabled is slower than with CompBin splitting disabled. One of the things I want to assess with this analysis is to investigate the hypothesis that increasing the CompBin size, thus reducing the number of Bins, reduces the run time when CompBin splitting is enabled.

Binning also affects memory usage; smaller Bins means that the data set in a given Run is subdivided into smaller parts. This in turn leads to less data in memory at a given time. Although this is important to take into consideration when deciding Bin sizes, this is beyond the scope of this analysis.

Plots

To present the results of all these Runs, I use plots that show the run time and “atom overhead” – an attempt at an estimate of the overhead each Bin causes. These plots might not be completely straightforward to read and understand, so a short explanation is offered below.

Run time This is a grouped histogram of run times where the grouping criteria is UserBin size. The intention here is to show how CompBin size affects run time at different UserBin sizes. Each of these plots involves one statistic analysis, one IndexBin size, four UserBin sizes and four CompBin sizes.

The data for these plots were generated by executing each run five times, measuring the run time each time; the mean run time of these is the resulting run time shown on top of each column.

The groups of columns correspond to different UserBin sizes, which can be seen below each group.

The columns in grey tones correspond to different CompBin sizes, each of which is shown in the legend to the right.

Atom overhead This plot shows the estimated overhead per atom (Bin) involved in each of the runs used in Section 6.2.2. Overhead in this context is the difference in run time between a Run with CompBin splitting enabled at a given CompBin size compared to the same Run with CompBin splitting disabled. The run time of the Run with CompBin splitting disabled is used as the baseline run time.
This means that a negative overhead should be interpreted as that Comp-Bin splitting at the given combination of Bin sizes actually improves run time.

Atom overhead is calculated as shown in Equation 6.1. This is an attempt at formalizing the effect that splitting the data set and computation has on run time, by calculating how much slower each atom makes the computation.

\[
\text{overhead} = \frac{t_{\text{CompBin size } n} - t_{\text{CompBin splitting off}}}{\text{number of CompBins}}
\]  

\text{(6.1)}

Results

All call graph subsections referred to below are found in Appendix C on page 87.

IndexBin size

I first did these runs with IndexBin size set to 100k, which is the current IndexBin size set in the stable release of the HyperBrowser. Then I tried setting the IndexBin size to 1000, which increases the number of indexes 100 fold. It makes sense to assume that more indexes will reduce the time spent seeking in the data files and looking up data, and thus reduce the total run time. However, the results I got seems to indicate that that assumption is wrong, because there is hardly any difference in run time at all.

For this reason, I only show the plots of results form Runs where IndexBin size is set to 100k below. The results from the Runs with IndexBin size 1000 is included in Appendix B on page 83.

UserBin size

MeanStat on Meltmap  Perhaps the most noticeable thing in the plot of the run times of MeanStat on Meltmap runs in Figure 6.4a on page 68 is that all the Runs with UserBin size 100k are significantly slower than all the other runs. To investigate this further, we can take a look at the callgraphs for the different Runs.

Figure C.1 on page 87 shows the calls to Statistic:compute for both Runs. This function is the point of origin for all calls to _compute on all Statistic objects involved, and is thus a good function to use as the starting point for looking at what is actual computation and what is overhead. Observe
that the cumulative run time for this function is 34.9% of the total for UserBin size 100k and 83.95% for UserBin size 1M. The total run times for these Runs is 7.9 and 3.2 seconds, respectively. We can use this information to calculate the total time spent inside `Statistic.compute` for each Run; 7.9 \times 0.349 = 2.757 for UserBin size 100k and 3.2 \times 0.8395 = 2.686 for UserBin size 1M. Doing the actual computation apparently takes approximately the same time at both UserBin sizes. This effect is pronounced in the Runs of the two other analyses as well.

**CountPointStat on Genes** This is recognizable in the plot of the CountPointStat on Genes Runs in Figure 6.5a on page 69 for instance. Figure C.3 on page 88 shows subsections of the callgraph for CountPointStat on Genes Runs at UserBin size 100k and 1M. The run times for these runs are 7.0 and 2.6 seconds, respectively. If we calculate the time spent inside `Statistic:compute` we see the same thing as in MeanStat on Meltmap; approximately the same time is spent inside the function in both Runs. 7.0 \times 0.2878 = 2.015 for UserBin size 100k and 2.6 \times 0.834 = 2.168 for UserBin size 1M – very close to the same run time.

**CountPointStat on Sequence, Repeating elements** In the plots of the CountPointStat on Sequence runs in Figure 6.6a on page 71, we observe the same effect. We can do the same calculation here; calls to `Statistic:compute` in the call graphs are shown in Figure C.5 on page 89. The run times are 7.5 seconds for UserBin size 100k and 3.1 seconds for UserBin size 1M. The time spent inside `Statistic:compute` is then 7.5 \times 0.322 = 2.415 and 3.1 \times 0.832 = 2.579 – very similar run times for these Runs as well.

**CompBin size**

Another pronounced pattern in the run time plots is the fact that for all Runs with UserBin size above 100k, the Runs where CompBin size is 100k are significantly slower than the rest of the Runs.

To look at what constitutes the difference in run time between these two configurations, we can do a similar analysis to that in Section 6.2.2.

**MeanStat on Meltmap** Figure C.2 on page 87 shows the call to `Statistic:compute` in a Run of MeanStat on Meltmap with CompBin size 1M. The total run time of this Run is 1.51 seconds, which gives us a total time spent inside `Statistic:compute` of 1.51 \times 0.6364 = 0.961 seconds. In 6.2.2, we saw that the time spent inside the same function for an equivalent Run, where the only difference was that the CompBin size was set to 100k, was 2.688
(a) The effect of UserBin size and CompBin size on run time.

(b) The effect of UserBin size and CompBin size on atom overhead.

Figure 6.4: Plots of how UserBin size and CompBin size affects run time and atom overhead of MeanStat on Meltmap at 100k IndexBin size.
(a) The effect of UserBin size and CompBin size on run time.

(b) The effect of UserBin size and CompBin size on atom overhead.

Figure 6.5: Plots of how UserBin size and CompBin size affects run time and atom overhead of CountPointStat on Genes at 100k IndexBin size.
seconds. This seems to indicate that on the contrary to UserBin size, the increase in CompBin size reduces not only the total run time, but the amount of time spent doing actual computation.

In fact, if we look at the difference in total run time in relation to the difference in run time of `Statistic:compute`, we can see that the change in the latter is close to proportional to the change in the former. The change in total run time is \( \frac{2.119}{2.119} = 2.119 \) and the change in `Statistic:compute` run time is \( \frac{2.688}{0.967} = 2.797 \).

**CountPointStat on Genes** To do the same analysis on CountPointStat on Genes we have a cutout of the call graph of a Run where CompBin size is set to 1M in Figure C.4 on page 88. The total run time of this Run was 0.70 seconds, which gives us a `Statistic:compute` run time of \( 0.7 \times 0.2775 = 0.1943 \) seconds. This is significantly lower than the 2.168 seconds spent inside the same function when the CompBin size was set to 100k in Section 6.2.2 on page 66. It seems that the change in CompBin size significantly reduced run time of the actual computation for this Run as well.

**CountPointStat on Sequence, Repeating elements** Figure C.6 on page 89 shows a subsection of the call graph of a Run of the CountPointStat analysis on a Sequence, Repeating elements Track. Note that the run time of `Statistic:compute` is 49.4%. The total run time of the Run is 1.01 seconds, which gives us a `Statistic:compute` run time of \( 1.01 \times 0.494 = 0.499 \) seconds. This is a significant difference from the 2.579 seconds observed for the same Run, only with CompBin size 100k, in Section 6.2.2 on page 66. The larger CompBin size seems to reduce the time the actual computation takes in this Run too.

**Discussion**

**Atom overhead** In theory, atom overhead should be close to the same for every Run. In practice however, it differs quite a bit between different configurations of CompBin size, UserBin size and IndexBin size. This makes it hard to say anything really general or even concrete for a single Run about the estimates it calculate.

**IndexBin size** It seems reasonable to assume that the number of indexes into a Track would have a positive effect on run time because less time is spent looking up data in the Track. The results indicate that this is not the case; the run times for IndexBin size 1000 and IndexBin size 100k are
Figure 6.6: Plots of how UserBin size and CompBin size affects run time and atom overhead of CountPointStat on Sequence at 100k IndexBin size.

(a) The effect of UserBin size and CompBin size on run time.

(b) The effect of UserBin size and CompBin size on atom overhead.
virtually the same. This is likely because the time spent looking up data with indexes is not a particularly significant part of the total run time.

**UserBin size**  UserBin size doesn’t have a particularly large effect on run time in the Runs I have done, except for in one case; UserBin size 100k. In this configuration, Runs are significantly slower across the board. Call graph analysis done in Section 6.2.2 on page 66 shows that the time spent doing useful computation is virtually the same as in Runs with UserBin size 1M. This implies that the difference in run time is actually caused by a significant increase in pure overhead. This increase in overhead is likely because of the large amount of CompBins involved when the CompBin size is that low. There is quite a bit of work that is not actual computation in each Bin; this adds up quick. Furthermore, creating, managing and releasing all of the Bins take quite a bit of time in itself.

The size of UserBins are completely dependent upon what data the user decides to run an analysis on, so there isn’t a lot we can do to change the size of the Bins.

**CompBin size** Somewhat similarly to UserBin size, CompBin size 1M and above seems to consistently give significantly faster Runs than CompBin size 100k. This decrease in run time isn’t caused by a decrease in overhead like in the case of UserBin size however. Call graph analysis done in 6.2.2 on page 67 shows that the decrease in run time is in fact approximately proportional to the decrease in actual computation. This means that a change in CompBin size from 100k, which is now the default setting, to 1M causes a significant improvement in computation run time across the board.

The run times we get with CompBin size set to 1M is actually very close to the run times we get with CompBin splitting disabled. This means that we can reap the benefits CompBin splitting enables, like reduced memory usage and disk memoization, without suffering a significant hit to run time.

### 6.3 Granularity at the file level

I mentioned several problems with the data handling subsystem in the HyperBrowser in Section 3.4.2 on page 27; problems like running out of inodes, or exceeding the maximum limit of open files at a given time.

These problems are only exacerbated by the fact that partially sequenced Tracks from some DNA sequencing machines are split into smaller parts called scaffolds.
In partially sequenced Tracks, we have what are called *scaffolds*; sparse groups of DNA sequences, sometimes with large gaps between each bit of sequence. In the HyperBrowser’s data handling subsystem, these scaffolds are basically treated equivalently to chromosomes in a fully sequenced Track. One thing to note about this is that while the number of chromosomes in most genomes are relatively low, the number of scaffolds in a partially sequenced genome like this can easily exceed 100. There are Tracks available in the HyperBrowser right now with more than 400k scaffolds.

While the atomic unit on the Python level of the HyperBrowser is the Bin, the atomic unit on the data or file level is really the chromosome. The HyperBrowser’s data handling subsystem is based on the fact that the number of chromosomes in a given Track is typically fairly low – in the hundreds, or low thousands.

One possible solution to these problems might be to store the large number of files per Track inside a of highly performant container format like HDF5, which is outlined in Section 2.5 on page 18.

### 6.3.1 Benchmarking

A good way to investigate if using HDF5 as a container format for the Track data in the HyperBrowser is feasible performance wise, is writing a prototype program and run a benchmark.

**HDF5**

To measure the speed of reading Track data from a HDF5 file, we must first have a HDF5 file containing Track data. Thus, our first job is to create a HDF5 file from a Track.

To manipulate HDF5 files from Python I used the h5py [9] library, which provides Python bindings to the HDF5 C library. h5py is actually in part implemented in Cython, which I mentioned in Section 2.2.5 on page 17.

Tracks in the HyperBrowser are stored on disk in a file hierarchy where the top-level directory in the hierarchy is the Track. Inside the Track directory there is a directory for each chromosome; each of these directories contain a number of `numpy` memmaps. It was logical to structure the HDF5 files the same way; the top level – the file itself – is the Track. Inside the file is a group for each chromosome, and inside each chromosome is a data set for each `numpy` memmap it contains.

When the HDF5 file is created, we can start the actual benchmark. The benchmark is a small program that takes a HDF5 file as argument, loops
over each group (chromosome) inside the file, and reads each data set from every group into memory.

**numpy.memmap**

To have something equivalent to compare the HDF5 run times with, I wrote a small program that just loops over each chromosome directory in the Track directory, and reads all the data from each `numpy` memmap. Very much like the HDF5 program.

To do the actual benchmark, I wrote a small program that takes the path to a Track directory as argument and runs both the HDF5 and `numpy` programs on the Track data. Both of them are run through `cProfile` to get the total run time for each program.

**Data sets**

To get an impression of how HDF5 and `numpy.memmap` performs on data sets with different characteristics, I set up four data sets with different amounts of data and files. All these data sets are either subsets or supersets of actual Tracks from the HyperBrowser data sets.

**Data set 1** 1 MB of data in 144 files, spread over 24 chromosomes.

**Data set 2** 70 MB of data in 6000 files, spread over 1000 chromosomes.

**Data set 3** 1 GB of data in 4 files, spread over 4 chromosomes.

**Data set 4** 6 GB of data in 72 files, spread over 24 chromosomes.

### 6.3.2 Results

**Data set 1**

The results of running the benchmark on data set 1 is shown in Table 6.1. `numpy.memmap` reads the 1 MB of data about 8 times faster than HDF5 through h5py.

<table>
<thead>
<tr>
<th>Runtimes for dataset 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDF5 run time</td>
</tr>
<tr>
<td><code>numpy.memmap</code> run time</td>
</tr>
</tbody>
</table>

| HDF5 slowdown       |
| 7.959935 x slower   |

Table 6.1: Runtime overview in data set 1.

74
Data set 2

The results for data set 2, shown in Table 6.2, tell the same tale as the results for data set 1; \texttt{numpy.memmap} is still significantly faster than HDF5. It seems that the relatively large amount of files does not affect the run time of neither \texttt{numpy.memmap} nor HDF5 particularly much.

\begin{center}
\begin{tabular}{ll}
\hline
HDF5 run time & 11.302481 s \\
\texttt{numpy.memmap} run time & 1.672305 s \\
HDF5 slowdown & 6.758624 x slower \\
\hline
\end{tabular}
\end{center}

Table 6.2: Runtime overview in data set 2.

Data set 3

The results for data set 3 is shown in Table 6.3. Even though data set 3 is over 10 times as large as data set 1 and 2, the ratio of HDF5 run time to \texttt{numpy.memmap} run time is still very similar to that of dataset 1 and 2.

\begin{center}
\begin{tabular}{ll}
\hline
HDF5 run time & 29.824221 s \\
\texttt{numpy.memmap} run time & 3.363129 s \\
HDF5 slowdown & 8.867998 x slower \\
\hline
\end{tabular}
\end{center}

Table 6.3: Runtime overview in data set 3.

Data set 4

Data set 4 (results shown in Table 6.4 on the next page, appears to be large enough that the run is getting I/O bound, judging from the ratio of HDF5 run time to \texttt{numpy.memmap} run time. \texttt{numpy.memmap} is now only a bit more than 2 times faster than HDF5.

6.3.3 Discussion

The HyperBrowser’s data handling subsystem is a very integral part of each Run. I mean this both in the sense that it is an important part of the Run, and in the sense that it often is responsible for a significant part of the total
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>HDF5 run time</td>
<td>224.802941 s</td>
</tr>
<tr>
<td>numpy.memmap run time</td>
<td>98.778870 s</td>
</tr>
<tr>
<td>HDF5 slowdown</td>
<td>2.275820 x slower</td>
</tr>
</tbody>
</table>

Table 6.4: Runtime overview in data set 4.

run time. It is not uncommon that as much as 50% of the total run time is spent inside `RawDataStat::compute`. Because of this, it is important that what happens inside the data handling subsystem is as efficient as possible.

Based on the results in 6.3.2 on page 74, it looks like using HDF5 as a container format for Track data will cause an observable increase in run time. Even though the difference between the run times of HDF5 and `numpy.memmap` seems to change in the favor of HDF5 when the data set increases in size. The data handling subsystem accounts for a significant amount of the total run time of many Runs already, and adding another slow layer of abstraction here will not improve that.
Chapter 7

Future work

7.1 Dynamic CompBin size

It might be worth it to investigate algorithms for setting the CompBin size for a run dynamically on a per-Run basis. This algorithm could be based on some heuristics where we use size of data set, statistic and number of chromosomes as input.

7.2 Further analysis of granularity on the file level

Currently, the lowest level atomic unit in the HyperBrowser’s data handling subsystem is the chromosome. We might want to investigate the effect of using different granularities at this level as well as on the Python level. One possibility could be to try using Bins as the lowest level atomic unit on this level as well.

7.3 Chunking data sets in HDF5 files

HDF5 supports a feature called chunks, which is a way to subdivide data sets internally in the HDF5 file. Apparently, this is supposed to make random access of slices of the data set faster, as well as enable efficient compression of data.

It might be worth it to investigate this further to determine if it is possible to achieve decent performance with HDF5 compared to numpy.memmap.
Appendix A

Determining factors for weighting operations

I have used the timeit Python module — which is in the Python standard library — to measure the run time of each Python operation.

The run time of the equivalent C++ operations are a bit harder to measure; basically I call clock() from time.h, then run the operation $n$ times, then call clock() again to get a time interval.

A.1 Functions for timing C++ operations

Listing A.1: A C++ function for timing C++ operations.

```cpp
double timeOperation(int iterations) {
    clock_t start = clock();
    for (int i = 0; i < iterations; i++) {
        // C++ operation goes here
    }
    clock_t end = clock();
    return diffInSeconds(start, end);
}
```

Where diffInSeconds just subtracts $start$ from $end$ and converts the result from clock cycles to seconds:

Listing A.2: A C++ function for getting the time delta between two clock_t instances.

```cpp
double diffInSeconds(clock_t start, clock_t end) {
```
A.2 C++ function call

This is just timing a basic function call.

Setup code:

Listing A.3: Setup code for timing C++ function calls.

```cpp
int func() {
    return 1;
}
```

The operation that is timed:

Listing A.4: Timed operation for C++ function call.

```cpp
func();
```

A.3 Python function call

Setup code:

Listing A.5: Setup code for timing Python function calls.

```python
def func():
    return 1
```

The operation that is timed:

Listing A.6: Timed operation for Python function call.

```python
func()
```

A.4 C++ class instantiation

Setup code:

Listing A.7: Setup code for C++ class instantiations.

```cpp
class Klazz {
    int i;
    public:
    Klazz(int);
}
```
```cpp
int getI(void);
};

Klazz::Klazz(int i) {
    this->i = i;
}
```

The operation that is timed:

Listing A.8: Timed operation for C++ class instantiation.

```cpp
Klazz *k = new Klazz(1);
```

## A.5 Python class instantiation

Setup code:

Listing A.9: Setup code for Python class instantiations.

```python
class Klazz(object):
    def __init__(self, i):
        self.i = i
```

The operation that is timed:

Listing A.10: Timed operation for Python class instantiation.

```python
k = Klazz(1)
```

## A.6 C++ string concatenation

For strings in C++, programmers have two possible choices, given that we do not consider third-party libraries; cstrings, which are the same as in regular C, and `std::string`, which is part of the C++ standard library. I did some quick testing with both, and it seems that concatenation of cstrings are a bit faster than concatenation of `std::string` instances. However, I decided to use `std::string` in this case because I have the impression that generally, `std::string` is used unless there is a very good reason not to. Furthermore, I use the containers from the standard library, so why not use the string class too.

Setup code:
Listing A.11: Setup code for C++ string concatenation.

```cpp
1. std::string s1 = "foo bar baz";
2. std::string s2 = "flim flam flum";
```

The operation that is timed:

Listing A.12: Timed operation for C++ string concatenation.

```cpp
1. std::string s3 = s1 + s2;
```

### A.7 Python string concatenation

Setup code:

Listing A.13: Setup code for Python string concatenation

```python
1. s1 = "foo bar baz";
2. s2 = "flim flam flum"
```

The operation that is timed:

Listing A.14: Timed operation for Python string concatenation

```python
1. s1 += s2
```

### A.8 C++ list access

I used the `std::vector`, the vector implementation in the C++ standard library for timing list accesses.

Setup code:

Listing A.15: Setup code for C++ list accesses.

```cpp
1. int arr[] = {1,2,3,4,5,6,7,8,9,0};
2. std::vector<int> lst (arr, arr + sizeof(arr) / sizeof(int));
```

The operation that is timed:

Listing A.16: Timed operation for C++ list access.

```cpp
1. int item = lst[6];
```
A.9 Python list access

Setup code:

Listing A.17: Python list access.

```python
lst = [1,2,3,4,5,6,7,8,9,0]
```

The operation that is timed:

Listing A.18: Python list access

```python
item = lst[6]
```

A.10 numpy operations

Because numpy is largely written in Fortran and C, and is linked to highly optimized, battle-tested libraries like BLAS\(^1\) [7], ATLAS\(^2\) [23] [24] and LAPACK\(^3\) [5] it is relatively fast.

It is entirely possible that a pure C or C++ implementation of the numpy layer in the HyperBrowser might in fact be slower than the existing implementation. On the flipside; because numpy relies on vector based calculations, we might be to apply some other techniques in a hypothetical pure C or C++ implementation of the numpy layer to make it as fast or even faster.

Because of the fact that this issue might swing both ways with regards to run time, I use a factor of 1:1 for run time of numpy operations to run time of C or C++ operations. This means that numpy operations will have a weight of 1 in this model.

\(^1\)Basic Linear Algebra Subprograms
\(^2\)Automatically Tuned Linear Algebra Software
\(^3\)Linear Algebra PACKage
Appendix B

Plots for binning Runs with IndexBin size 1000
(a) The effect of UserBin size and CompBin size on run time.

(b) The effect of UserBin size and CompBin size on atom overhead.

Figure B.1: Plots of how UserBin size and CompBin size affects run time and atom overhead of MeanStat on Meltmap at 1k IndexBin size.
(a) The effect of UserBin size and CompBin size on run time.

(b) The effect of UserBin size and CompBin size on atom overhead.

Figure B.2: Plots of how UserBin size and CompBin size affects run time and atom overhead of CountPointStat on Genes at 1k IndexBin size.
(a) The effect of UserBin size and CompBin size on run time.

(b) The effect of UserBin size and CompBin size on atom overhead.

Figure B.3: Plots of how UserBin size and CompBin size affects run time and atom overhead of CountPointStat on Sequence at 1k IndexBin size.
Appendix C

Call graph subsections for binning Runs

Figure C.1: Cutouts from the call graphs of MeanStat on Meltmap with UserBin size set to 100k and 1M.

Figure C.2: Cutout from the call graph of MeanStat on Meltmap with UserBin size and CompBin size set to 1M.
Figure C.3: Cutouts from the callgraphs of CountPointStat on Genes with UserBin size set to 100k and 1M.

Figure C.4: Cutout from the call graph of CountPointStat on Genes with UserBin size and CompBin size set to 1M.
CountPointStat on Segments, Repeating elements

Figure C.5: Cutouts from the callgraphs of CountPointStat on Sequence with UserBin size set to 100k and 1M.

(a) UserBin size 100k.
(b) UserBin size 1M.

Figure C.6: Cutout from the call graph of CountPointStat on Sequence, Repeating elements with UserBin size and CompBin size set to 1M.
Bibliography


