Contents

Chapter 1

Introduction

With the growing amount of generated data, the size of files is increasing as well. To deal with this problem efficient loading and importing of these files is an important topic to deal with. Especially in times of bigdata where file sizes of serval gigabytes are common and each second of high-performance computing is very valuable. In many cases algorithms can't proceed with an incomplete dataset, which means that everything else has to wait on the loading task.
The data is often stored in graph like networks, where data is represented by vertices and edges. These graphs can consist of billions of small objects, in which case the metadata-overhead must be very small to work efficiently. There are many kinds of various graph file formats but this thesis will only focus on the most common and relevant ones, which serve the purpose of exchange the information between two systems.

This works tries to resolve the problems, which comes with trying to use the resources of a distributed system to accelerate the loading of graphs into memory and manage the chunk distribution in the network. To accomplish this the file format must be parallelizable. This needs to be determined first. The graph files formats, this work deals with, will be evaluated to their parallelizability, their relevance and current state of use. Over the course of time various graph file formats got introduced with many different properties and purposes. They vary from simple edge lists to complex formats in binary or xml format.

The current state-of-the-art standard is that a parallelizable file format gets used and mapped on a map-reduce framework, often simple formats are used, like connection list, where the overhead to deal with is very small and one line describes exactly one node and its edges. It's common practice that complex formats get parsed into simpler ones and then read in. This work will try to accomplish a map-reduce-like approach with many more formats, while trying to preserve the performance gained by the resources of the distributed system.

Chapter 2

Preliminaries

2.1 Graph Notations

Definition 1 *A graph G ist defined by a tuple (V,E) with V being a set of Verticies and E a set of Edges. $V = (v_1, ..., v_n)$*

1. *A graph is directed if the elements in E consists out of tuples.*
   $E = \{(v_i, v_j), ..\}$ for $v_i, v_j \in V$ with $v_i \neq v_j$
2. *A graph is undirected if the elements in E consists out of sets with the size of 2.*
   $E = \{\{v_i, v_j\}, ..\}$ for $v_i, v_j \in V$ with $v_i \neq v_j$

This definition is based on [**?**]. If $\{v_i, v_j\} \in E$ then v_i and v_j are neighbors. In a directed graph the direction of the edge matters, if $(v_i, v_j) \in E$ then v_i is a neigbor of v_j, but not the other way around.

Definition 2 *A graph G is called a*

1. *Multigraph if it contains multiple edges between the same vertices or edges from a vertex to the same vertex.*
2. *Hypergraph if it contains edges, which have more than two vertices connected to it.*

2.2 Graph File Formats

A graph file is a file, which contains the information to construct a Graph *G*.
Based on a the collection of graph file formats by [**?**].The most relevant file formats where chosen to be featured in this work. Relevant formats are formats that have been used in the recent time and still seem up-to-date, which is indicated by the overall use counts. Also, the amount of sample data of this file formats that can be found matters. File formats that don't provide any clear specifications won't be chosen either, duo to the fact that implementing them would result in a non-consistent parsing/reading, duo to missing constrains. This is based on the problem that other people don't have any specifications either. This could result in the missing conformity of their files. This work will try to select atleast one of each structure type to provide an example implementation of each structure type of graph file formats, so the amount of supported graph file formats can be extended easily.

Id	Graph File Format	Reference Time Frame	Structure
1	bintsv4 (GraphLab)	2009 - 2015	Simple binary
2	Dot	2000 - present	BNF
3	Graph::Easy	2004 - present	intermedia
4	GraphML	2000 - present	XML
5	JSON Graph	2014 - present	JSON
6	KrackPlot	1993 - present	simple
7	Matlab	1996 - present	HDF5
8	Ordered Graph Data Language	2002 - present	BNF
9	Open Graph Markup Language	2012 - present	XML
10	Simple Interaction Format	2003 - present	simple
11	Stanford Network Analysis Platform	2005 - present	simple
12	So NIA Son format	2002 - present	intermedia
13	Trival Graph Format	NA - NA	simple

Table 2.1: Selection of Graph File Formats based on [**?**]

2.2.1 Divisibility

One of the main topics of this work is to increase the loading speed, by dividing large files into smaller chunks and load them on multiple peers at once. To archive that the file format must be able to be divided into such chunks. Otherwise a distributed loading isn't possible and a single peer solution will be the best option.
Most of the file formats are divisible in some sort but the fastest way to split files is without or minimal reading of the file. So, the files still need to be dividable, if most of the information is skipped. Skipping most likely won't be an option for all formats but for the simple and trivial ones it definitely will be.
To determine if a format is divisibility the structure of the file needs to be specified and an-

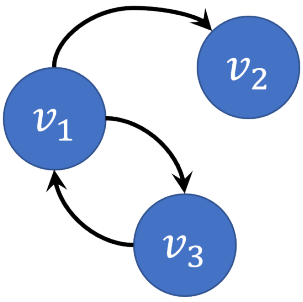
alyzed. Most important data inside the file shouldn't be dependent on other sections of the file. Sure, there will be formats that specify information about vertices in multiple sections of the file, but this information need to be independent. If the following data can only be processed after processing all lines before, then this section can be split. Eespectively this sections appear very often because most key/values-tuples can't be split. This section will be called contiguous regions of a file.
These contiguous regions are often closed of by separators. An seperator could be anything often newline characters, tabulations and semicolons are used, but also tags (XML), brackets (JSON) or even the position itself in binary sequences can be used to divide these sections. Some formats provide metadata, that will help split up files by defining the position of the information and their format.
A format is fully divisible if it could be split at the any seperator. This definition is deliberately vague, because many test cases can be constructed to defeat certain characteristics.

2.2.2 Simple Formats

Simple formats can be often referred to as formats that provide a trivial approach to create a graph format. Often no specification is provided and it isn't clear where the boundaries of this formats are, like the range of integer values, which character set is used or if the format contains metadata/comments. Many of these formats are fully dividable, duo to the fact that they often contain only one contiguous regions and have a plain hierachy. Also only one seperator is used, so that spkitting at every seperator is given. [**?**]

Trivial Graph Formats

As trivial graph file formats (TGF) are referred to a list of formats that vary by every implementation. Standard approaches are edge list, adjacency matrices, neighbor lists and path list. Most of the lines inside a simple edge list are the same list just connections between node a and node b separated by tabs, comma or spaces. It is clear that this format can be split easily by just jumping to a position and reading till the next line. All following data could be a new chunk, same approaches can be applied to adjacency matrices, neighbor lists and path list.
The SNAP format is very similar to an edge list with the only addition that comments can be added to the file with a hash tag at the beginning of the line, that's why it is group with the TGF.
In all these formats lines can be seen as contiguous regions and only one seperator is used, in fact newline characters are the most common separator for TGF.



adjacency matrices

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

neighbor list

1,2
1,3
3,1

edge list

1 : 2,3
2 :
3 : 1

path list

3,1,2
1,3

Figure 2.1: An example graph with three vertices in different trivial graph formats [**?**]

SIF Format

The SIF Format is a combination of an edge list with a neighbor list. Additionally, the type of connection can be specified with strings. It allows multiple edges between the same nodes, if the connection type varies, otherwise it is specified to ignore duplicates. This results in an Multigraph. One downside is that the format allows tabulations or spaces as separators, it is stated that if no tabulations are used in the whole file spaces are used as separators, but because all lines have an equal format, it can be stated, if the first line doesn't contain any tab character, spaces will be used as separators. Based of the fact that SIF is merge of two TGFs, it can be said that SIF is also fully divisable, only one separator is used per file and the lines are the contiguous regions. [**?**]

KrackPlot 3.0

KrackPlot 3.0 follows for simple formats a more complex syntax then the TGFs. The first line contains the number of nodes specified in the following data. This information helps with various problems, like splitting nodes appropriately into chunks. The next line can contain two options "!nc" or "!nl". The first option specifies that the following data contains no coordinates, the other one declares that no labels will be specified. If labels and/or coordinates are defined they start on the second line until line (node-amount) +1. After the second line or the labels/coordinates an adjacency matrix specifies the connections of the nodes. This design choices of the format, make reading huge files on multiple peers tedious, duo to the fact that labels and coordinates are split apart from one another. Also, the overhead of an adjacency matrix is huge for large files. Splitting this file is possible duo to the fact that we know the number of nodes. Only two lines need to be read in to know the whole nature of the file, which can be split according to the metadata. [**?**]
This formats consists out of two different sections. These sections contain different contiguous regions. This format is fully divisible becausse the size of the sections is know.

2.2.3 XML Formats

There are various implementations of graph file formats using XML. The XML format is based around tags, which define the object it is describing. XML was most popular in the 90s and it is a structure descriptive language. Reading its information is as result not line based rather it is tag based. This format is hard to divide without reading it completly, because the XML format conists of multiple layers. This results in the problem to determin the layer on which the object is located, this problem can only be solved by counting the opened and closed tags or using a flat hierachy. [**?**, **?**]

GraphML

GraphML is an XML based graph file format. It consists out of one graph element which can contain unordered node and edge elements. GraphML supports hyperedges and nested networks. This results in the problem of deep hierachies, which can only be solved by tag counting do determin the layer. GraphML support many kinds of graphs, this variety causes many different cases to consider. GraphML doesn't specify the positions of vertices or edges inside the format, so that chunks could result in an unequal distrubution of different objects. As result GraphML isn't fully divisble and needs to be read chunk by chunk, no information can be skipped. [**?**, **?**]

Open Graph Markup Language

The Open Graph Markup Language is part of the Open Graph Drawing Framework. The OPML got some similarities to GraphML, duo to the fact that both formats resolve around XML. Also, this format implements many graph style and drawing operations. Also the OPML doesn't support nested graphs unlike GraphML. [**?**]

Resource Description Framework

The Resource Description Framwork/XML Format is format that features the model of three information types. RDF defines via namespaces objects and attributes, which makes RDF a portable format, which got used to share graphs overtime and wasnt developed with this goal in mind. [**?**, **?**]

2.2.4 JSON Graph

JSON Graph is unlike XML just a syntax convention based on the JSON-syntax-specification. This file format extends JSON files by introducing objects for graph description. These files are valid JSON-files and will be accepted by any JSON parser. Inside the JSON Graph file, a graph object is defined which contains nodes and edges with metadata. This format comes with a huge flaw, because the number of nodes, edges or any assistance to navigate inside the file with our reading it until that point isn't given. Nodes and edges can span over multiple lines and don't have a fixed size in any way. Metadata is expandable without any limits. The result of this design choice is that the file needs to be read sequentially before splitting it up into chunks. This will result in a huge performance loss.
One way of reading this format into memory while also using our maste/slave-system would be filling a buffer with the information read. And after that sending it to the processing slave. [**?**, **?**]

Other Formats

This thesis can't deal with all graph formats that are available. For those cases the application will provide an API to extend its range of covered graph file formats.

Chapter 3

Architecture

This chapter will give a brief introduction into DXRAM. Then the schema according to which the graphs are going to be loaded will be explained. After that problems that will occure will be discussed.

3.1 DXRAM

DXRAM is a distrubuted in-memory key/value-store. It is implemented in Java and optimized to manage billions of small data objects. For low latency data access DXRAM keeps 100% of the data in RAM. DXRAM provides a low data overhead, which suits hrapg based application very well. Nodes of DXRAM can take the role of a ”normal” peer or a superpeer. Superpeers are arranged in a chord like ring structure.

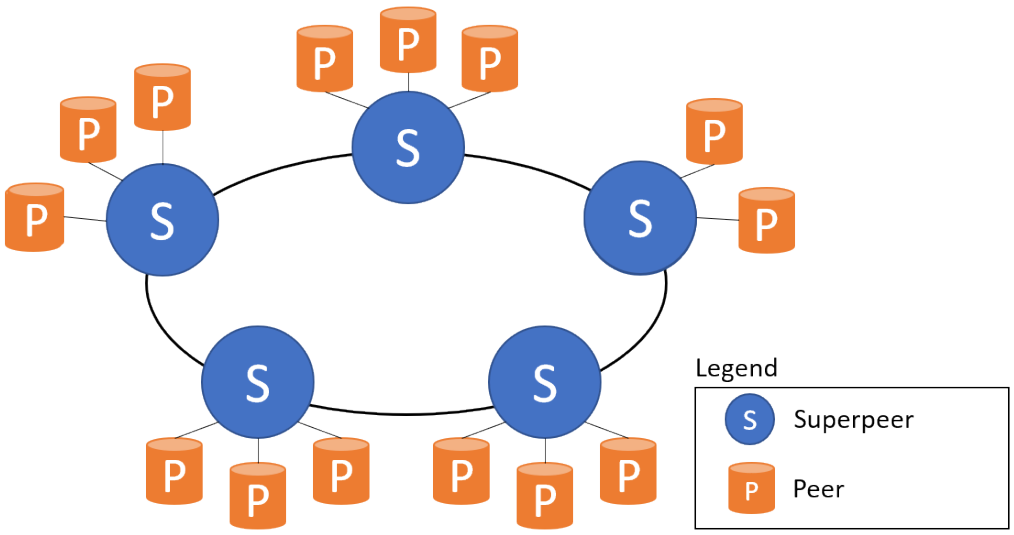


Figure 3.1: Topology of DXRAM

Peers are always assigned to one superpeer. Superpeer take over adminstativ tasks within the distrubuted system while peers serve the role of a storage or backup nodes. DXRAM provides its own application programming interface through services.DXRAM uses a distrubuted file system, so every node can access every file. They can be loaded by applications to get access to storage, computing power and more operations. Custom applications can be loaded as jar-files, but must implement the `AbstractApplication` class and register thereself. All nodes in the system are accessible through the `BootService`, which can list all online peers and theire role inside the system. The key/value-store can be accessed by the `ChunkService`, who provides several operations for creating, getting and storing data structures in chunks. All data structurs, which should be storable, must implement the `AbstractChunk` class.
The computing power of the network can be used in serveral ways. One way is the `JobService` which manages lightweigth jobs. Every job has a unique id and inherits from `AbstractJob`. Every job has its own thread and can run on a local nodes or be pushed to a remote node. Job will be pushed into a job queue and wait for their execution.. Another way to compute something is the `MasterSlaveService` enables to run parallel computations on groups of peers. In this group peers can takes different rolee, one peers takes the role of the master, which coordinates the task. The other peers are either workers(slaves) or take no role. One master and several workers make up one computation group. [**?**, **?**]

3.1.1 Integration in DXRAM

There are several ways of integrating foreign code in DXRAM. One goal is to be able to load data via a terminal command, but also to be able to easily integrate the loader as a library. Therefore the main part of the application will be accessible through a custom Job, which can be used as library. To provide terminal access, without the current possibility to execute jobs from the terminal, the job will be wrapped by an DxApp.

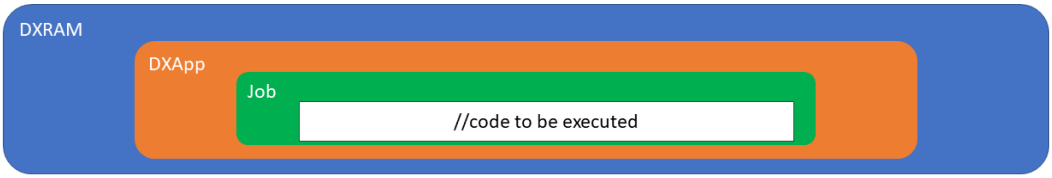


Figure 3.2: Schema of code integeration in DXRAM

One problem that needs to be solving is that at the current state of development, DXRAM isn't able to register jobs on remote peers. This problem occurs to dependency issues, another problem is that jobs can only be registered on the current peer, but local pushing of jobs is by default disabled. To solve this issue The main application invokes the call of a `JobRegistrationApp` Application, which are loaded dynamically at runtime. This helper application takes in a list of jobs (function would be possible as well) and registers them on every peer in the network. This offers this application the possibility to execute remote jobs.

3.2 Architecture of the Application

The architecture of the application consist out of multiple steps. The first steps is to setup our environment, this includes calling the `JobRegistrationApp` to register all needed jobs on all peers and also validating the passed in parameters. Some parameters are required like the file(s) to load and the format, other parameters are optional like the amount of peers or an list of peers to run this application on, by default this application will try to use every peer available for maximizing the loading speed.

3.2.1 Deployment of File data on nodes

The deployment can be done in multiple ways, which have to be evaluated. One possibility would splitting the original file into multiple smaller files and access them via the NFS. This approach would be straight forward, but results in one read and write operation of the whole original file size. Also the file would need to be loaded again into memory for extracting its data. This would result in $O(n)$ on the initiating peer and $O(n/chunks)$. Another approach would be to load the data into memory and send chunk structures to the according peers. This reduces the writing operation and another reading operation on the peer, because the data lays already in memory. The last method would be to access the original file via the NFS and extract the chunk directly on the remote peer. This would result in distriubution of the loading of the original file in one run, but could result in the bottleneck of the one storing node with the original file. This would probably be slower duo to the fact that all nodes accessing one file, would cause a random read sequence, which is often slower then sequential reads.

```
1   ChunkService chunkService = getService(Chunkservice.class);
2   String filePath = ''examplefile.txt'';
3   int chunkSize = 4096 * 1028; //in bytes
4   List<short> peers = getService(BootService.class).getOnlinePeers();
5
6   FileChunkCreator chunkCreator = new FileChunkCreator(filepath, chunkSize);
7
8   while(chunkCreator.hasRemaining()){
9       for(short p:peers){
10          FileChunk fileChunk = chunkCreator.getNextChunk();
11          if(fileChunk != null){
12              chunkService.create().create(p, fileChunk);
13              chunkService.put().put(fileChunk);
14          }
15      }
16  }
```

While the chunks are created from the file, the chunks could already be processed on the remote peers, so that the concurrency reduces the loading time significantly.

Efficient Loading of File into Memory

One key ability of the application is that the loading of the file system should be fast. Often the reading/writing from Storage Devices is a bottleneck. The first problem is that most Storage Devices are designed for sequential read and write operations and often support only one operation at a time. This leads to the problem that multiple threads reading the storage device is a waste of performance and won't speed up the input rate of the file. Another problem is that accessing a storage device will cause a context switch. This results in the application stopping, while the kernel executes sensitive functions. This problem is often address with buffering the files and by requesting more data then needed in one step. Buffer sizes are one optimization problem to consider. [5]
The traditional way of accessing the data of a file is a common routine. First the file gets opened after that the data can be read, written in sequential or random order. This method causes many context switches, which force the application to stop, while the kernel executes sensitive functions. That's why in nearly every case IO gets buffered, so the context switches occur less often. [6]
To reduce the amount of context switches to a minimum. This problem is addressed by memory mapping and suits large files where we can reduce the context switches drastically. A virtual memory mapping between the filesystem and the application address space is created. So expansive system calls can be avoided. The setup of this method is more expensive then the setup of file IO. [5,6]**??**

3.2.2 Processing/Parsing of the Data

The processing/ parsing of the data is unique for each format. Some formats like the TGF need a a different parser in contrast to XML or JSON sytle formats. The best option would be to create as many threads as possible to read in the data.

3.2.3 Storing

The extracted vertices, edges and optional metadata needs to be stored. For this task the `ChunkService` will be used to push the created structs into the key/value-store. ne problem of the distrubuted loading is that peers cannot wait on other peers to get information about vertices, because this would result in too much messages and network traffic for large datasets. This still leaves the problem of not synchronized vertex objects on different peers, which cant be written to the key/value-store because their information could be incomplete.

Duplicate node objects

Some formats (especially in edge lists) nodes just get described by their edges. One problem is if two different peers create the same nodes by different edges, so they don't know if other peers already created a node object for specific nodes. As result they create a local node object with the information they got. In that case after reading the file in its entirety some merge of all hash maps and objects would need to be done. This would result in the

problem that one assumption of this work is that one peer can't keep all nodes and edges in memory, because there are too many objects.
Hash-Distribution of the Nodes while reading
As result of the fact of reorganizing the nodes after they have been loaded is very inefficient and doesn't suit our needs, the next approach would be to organize the nodes at loading/parsing-time. One way could be to hash the labels/ids used in the graph. Note that the hashes created have no security constrains and don't need to be unique. This hash could be used to divide the nodes onto the peers. As result every peer would buffer all nodes, that don't belong to its range of hash values, and send the information he gathered to the according peers. This way we don't have to deal with duplicate nodes or merge. One problem of this approach is, that peers could end up with a chunk, that contains no nodes that hash to itself.
Hash-Distribution after reading
The peers could finish reading the nodes of its chunk and create their own versions in their key-value store. After reading and parsing is finished, they could like in first solution send the key-value-objects to the according nodes based on the hashes of their ids. The target peer then merges all versions of that node and stores the copy that has all information, the other duplicates get deleted. This could result in many objects for one node for each peer

3.3 Issues

A requirement is that the file is located inside one of our peer's storage devices and isn't part of our distributed system in any way. Loading files via an internet connection, will not be featured. Some start parameters are indispensable for this type of application, like the absolute file path, the format of the file or rather the parser that should be used and the number of peers to load the file. Frist one peer should deal with the file, to distribute it to our others peers. The best thing would be, if the peer, who deals with file, would be the same peer where the file is located on. Otherwise the file transfer to the according peer would be a waste of time, especially for file sizes of several gigabytes. After the file chunks got deployed on the network, they could be parsed and loaded by the prior assigned peers into our DXRAM key-value store. [4]

First the file data gets loaded into small chunks which can be distrubted tFirst the file gets partially mapped into the memory, where it can be read by the master slave. The master slave first deals with the format specifications of the file, providing the division schema for splitting the file into chunks. chunks and will be dealing with the file format. Duo to the fact that the data, is in memory, multiple thread can access the data parallel to accelerate processing. The processed data will be collected in chunks which will be deployed to the assigned slaves. The support of infiniband would be increasing the performance drastically, due to spreading the data in the distributed system would be faster. When the chunks arrive on the according slaves, they could immediately start processing their chunks.

Duo to fact that some formats specify data of nodes in two or multiple places inside the file, it would be an interesting approach of using for example two memory maps of the same

file in different sections. So, the chunk (buffer) can be filled with both information at the same time and the node information lie on all on one slave.

3.4 Application Programming Interface

The loader application provides an simple API. To add own formats, the class needs to be added to the `SupportedFormats` class and extend the `GraphFormat` class, which provides information of the required splitting type of the format and, how it should be read.

3.4.1 SupportedFormats

The `SupportedFormats` class, is the lookup table for all formats and which loader and splitter to use. It provides three simple functionalities.

addFormat(String key, Class<? extends GraphFormat> format)
This function adds a format to the supported formats by assigning a key to a class type. For example ”edgellist” is the key for the class `EdgeListFormat`.

getFormat(final String key, final String[] files)
This function returns the class file for a given format key.

isSupported(String key)
This methods returns if a given key is supported.

3.4.2 GraphFormat

To add a custom format, the class needs to extend the `GraphFormat` class. A custom format specifies a Splitter and a Parser.

Chapter 4

Evaluation

4.1 Benchmark: File Reading Methods

Three diffrent methods where tested. The first method utilizes the Java `Stream` api, to stream the lines of the file. This resulted in a short and intuitiv way of file handeling. The second method uses a `BufferedReader`, this method was also very easy to set up and reading files by line is a native function. RThe last method was java's implementation of mapped memory. At first the implementation this work would be based on wasn't faster then the other methods, duo to searching in Arrays and String for newline characters. The final implementation iterates over a char buffer provided by the `MappedByteBuffer` API and retuns a String.

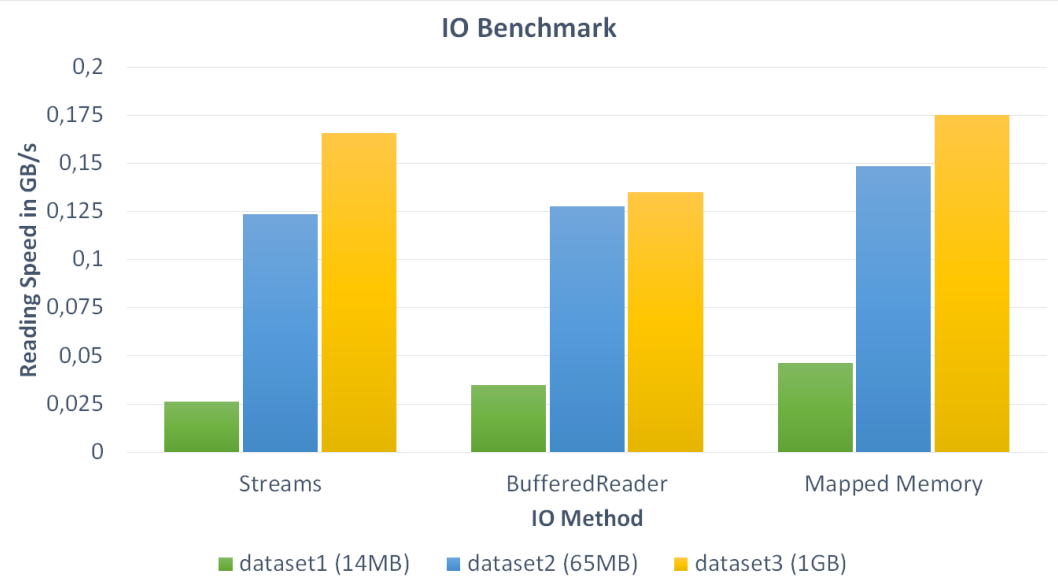


Figure 4.1: IO Benchmark for File reading in Java

The resut of these optimizations can be seen, mapped memory is (not by far) the fastest method, for byte wise input mapped memory was by far the fastest. It can be seen that for the small files from dataset1 the initialization cost is very high compared to the total time taken, but when reading larger files, the initialization seems to be constant and it can be seen that the reading speed converges.

Chapter 5

Implementation

5.1 Benchmark

Input: A[0..N-1], value
Output: value if found, otherwise not_found
`// Lokale Variablen`
let *low* = 0 let *high* = N - 1
`// Es gibt auch noch weitere Loops wie ForEach`
while low <= high **do**
  `// Invariante: value > A[i] for all i <low`
  `// Invariante: value < A[i] for all i >high`
  mid = (low + high) / 2;
  if A[mid] >value **then**
    high = mid - 1;
  else
    if A[mid] <value **then**
      low = mid + 1;
    else
      return mid;
    end
  end
  return not_found;
end

Algorithmus 1: Binärsuche in Pseudocode

5.2 Java-Code

```
1  @Benchmark
2  @Warmup(iterations = 5)
3  @BenchmarkMode(Mode.SingleShotTime)
```

```
4       @Measurement(iterations = 20, batchSize = 1)
5       @OutputTimeUnit(TimeUnit.SECONDS)
6       public void memMap_lines(Blackhole blackhole) throws IOException {
7
8           //heavy lifting
9           RandomAccessFile file = new RandomAccessFile(new File("data/" + path), "r↩
                ");
10          FileChannel fileChannel = file.getChannel();
11          MappedByteBuffer buffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, ↩
                0, fileChannel.size());
12          file.close();
13          fileChannel.close();
14
15          //reading by byte and sort into a extendable char array
16          int approx_line_length = 128;
17          char[] chars = new char[approx_line_length];
18          {
19              while (buffer.hasRemaining()) {
20                  char c;
21                  int i = 0;
22                  do {
23                      c = (char) buffer.get();
24                      try {
25                          chars[i] = c;
26                      } catch (ArrayIndexOutOfBoundsException e) {
27                          chars = Arrays.copyOf(chars, chars.length + ↩
                                approx_line_length);
28                      }
29              i++;
30                  } while (c != '\n' && buffer.hasRemaining());
31                  blackhole.consume(new String(chars, 0, i));
32              }
33          }
34      }
```

Appendix A

Mein Anhang

Klassendiagramme und weitere Anhänge sind hier einzufügen.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Gasterstädt, Sven

Düsseldorf, 15.03.2019