EXPERIMENTAL CS THESIS

JENS CHRISTIAN BREDAHL MADSEN

The Turbo framework
Designing and evaluating a general and efficient distributed parallel computation framework

Master's Thesis
Department of Computer Science
Science & Technology
Aarhus University

January 2017

ABSTRACT

Effective processing of demanding computational tasks is as relevant as ever, as an increasing volume of continously larger datasets are collected and machine learning and data mining is becoming widespread for various purposes, while many areas of science in addition has a strong demand for powerful computers. This thesis explores parallel distributed computing in which a set of problems are solved in parallel by distributing the problems over a network of computers, as a way to archieve powerful computing capabilities. By comparing the frameworks BOINC, Hadoop and Spark, the features that such a framework must provide and the issues that it must target effectively is identified. These findings have been used to construct a framework by the name of Turbo that provides a MapReduce-like approach to parallel distributed problem solving and a novel approach to fault-tolerance.

SAMMENFATNING

Effektiv udførsel af krævende computationelle problemer er et aktuelt relevant problem i funktion af at flere og større mængder data indsamles hele tiden, med et behov for at blive anlyseret gennem machine learning og/eller data-mining, ligesom at mange forskningsområder er iboende computationelt krævende. I dette speciale beskrives parallel distribueret problemløsning, hvor en samling problemer løses ved at blive spredt ud over et netværk af computere, der i samspil opnår stor kraft. Gennem sammenligning af Boinc, Hadoop og Spark frameworksene identificeres krav og udfordringer for et sådan framework, der bruges til udviklingen af mit eget framework, Turbo. Turbo baserer sig på en MapReduce-lignende tilgang til problemløsning og præsenterer en ny tilgang til fejlhåndtering.

ACKNOWLEDGMENTS

I would like to give a big thank you to Niels Olof Bouvin for his role in this thesis and his everpresent patience towards my unstructured being. I have always come out of our meetings smarter than before.
I would also like to give thanks all the educators at CS@AU for world class academic teaching and to Rambøll Management Consulting and Systematic A/S for providing me with the opportunity to hone my practical development skills.
Thanks to mom and dad.

Especially thanks to Turbo, the elephant, for allowing me to borrow his name for the framework.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Computers are widespread in homes for socializing and entertainment and used widely commercially as a tool for a wide variety of tasks. Additionally, computers are becoming an invisible part of our lives abstracted away in intelligent devices that make up the internet of things. The computer is, however, as important as ever in serving the function it originally was envisioned for; a machine that does calculations effectively. Throughout the history of the modern computer, work has been ongoing to improve the capacity of computers, as evident by Moore's law regarding transistors, as well as hardware evolution in general. There is a strong desire to make computers more effective, as more computations can then be made in less time. One of the key areas of modern computer science is so-called big data. Many research fields, commercial industries, medicine, etc are changing as it has become feasible to collect and store extremely detailed and complex data sets. These large amounts of data contain all kinds of insights and valuable information that must be extracted. Additionally, the NP-complete problems and other inherently computationally demanding tasks present issues due to their intensive demands. Computationally intensive tasks has use in a wide range of fields such as quantum mechanics, weather forecasting, climate research, oil and gas exploration, molecular modeling and in physical simulations and many others.

For these reasons, many tools has been proposed to archieve more efficient computing than a single general-purpose computer is able to provide. Solutions such as approximation algorithms and heuristics avoid computations by simplifying a problem by making certain assumptions about the problem and its solution. These algorithms do not fully solve the problem, but simply provide an answer that is close enough to the real solution. Another approach is parallelization, where a problem is split into subproblems that can be solved simultaneously with multiple cores. On a single general-computing machine this can be archieved through multithreading, where each core processes a task in parallel with the other cores. A problem that takes time t in a single-threaded application will when divided into m subjobs take t/m time, assuming that m is less or equal to the number of available processors and the overhead of parallelization is negligible. Parallel solution of a problem can also be performed by distributing the tasks across multiple machines, which is the basis for so-called supercomputers.

In this thesis I examine the architecture and design of distributed parallel computation frameworks and analyze their core features and challenges they must solve. I present my implementation and evaluation of a framework, coined *Turbo*. The framework allows the user to provide it with any task that implements a Java interface and it will automatically spread it across a network of nodes and solve it in parallel. The system has a unique approach to scheduling of tasks and fault tolerance.

2 RELATED WORK

To meet the goals of a parallel task-solving distributed system, a number of features has to be present in a framework. These include

- A interface so the user can provide tasks and data for processing
- A data source – Data can either be loaded directly or come from somewhere external
- A mechanism to distribute the task/data to the worker nodes so that parallelization is achieved
- Returning a result – the initial provider of the task needs to retrieve the result from somewhere.
- Reliability measures. As the risk of failure increases in a system of many computers connected by network, fault tolerance measures must be in place to prevent undesirable scenarios.
- Performance measures. A system should perform its intended function efficiently, otherwise its usability is limited.

2.1 OVERVIEW

Parallel distributed computing frameworks are generally designed either primarily for *cluster computing* or *volunteer computing*. A cluster is a set of dedicated computers, typically identical in hardware and operating systems (OS), connected by local area networks that perform in collaboration as a single highly effective unit (figure 1). These machines are usually physically close and can be assumed to have the same properties and capacity due to their identicalness. The units of a cluster may consist of either individual machines or one machine may simulate multiple machines using virtualization.

Volunteer computing is many ways the opposite of cluster computing. In volunteer computing, volunteers make their personal computers available for execution of applications retrieved from a remote server. Usually, the user downloads a client program that detects whenever the user is idle and executes the application until the user resumes activity. At some point the application finishes and the application has computed a result that is returned to the server after which a new task is requested. Volunteer machines do usually not communicate with other machines or collaborate; they simple recieve a task from a server, execute it and return a result to the server. As any person with a personal computer anywhere in the world can join as a

Figure 1: A 52-node Beowolf Cluster used for analysis of binary pulsars (Source: https://en.wikipedia.org/wiki/Beowulf_cluster)

volunteer, a high level of resource heterogenicity is to be expected and may vary at any point. Volunteer machines may differ in OS, memory capacity, disk space, read/write speeds, number of CPU cores, network latency, GPU units, etc. A volunteer may at any moment decide to not volunteer anymore and withdraw his machine from the pool of resources. Thus, while clusters are generally predictable and simple to reason about and maintain, volunteer networks are unpredictable and outside the control out of the framework.

Below is presented the design of a few different types of systems that highlight different approaches to parallel problem solving and emphasize different aspects of doing so, in their design. First, a short introduction are given for each framework, followed by a detailed description.

BOINC Berkeley Open Infrastructure for Network Computing (BOINC) is an open-source middleware software platform for volunteer computing, i.e. a form of distributed computing, where volunteer users make processing power and/or storage available, for science research projects. On the volunteer computer, a client program borrows resources when the user is idle. Inspired by SETI@home, a volunteer computing project for analysis of radio telescope data, development began in 2002 out of the desire to create a general software platform for volunteer-based computing and to improve certain aspects of pre-existing volunteer systems [1]. In 2003, various preexisting projects, including SETI@home, started migrating to the platform and today, it is widely used for volunteer computing projects in many areas of natural science [2].

1 http://www.angelfire.com/jkoulouris-boinc/
2 https://en.wikipedia.org/wiki/List_of_distributed_computing_projects

The power of volunteer computing is determined by the degree to which volunteers participate and share their processing resources. FLoating-point Operations Per Second (FLOPS) is a common computer performance measure. The world's currently fastest supercomputer, Sunway TaihuLight, achieves 93 petaFLOPS on a benchmark measurement [3]. In comparison, Folding@home, one of the major BOINC-based projects, is estimated to contain more than 100 petaFLOPS of total computing power [4].

HADOOP Hadoop refers both to the base Hadoop framework and a larger ecosystem of related packages that can be used with the framework. In this thesis it will be used exclusively for the former. Hadoop was first released in 2011 as a framework for cluster computing based on a distributed file system in the form of Apache Hadoop Distributed File System (HDFS) and Apache MapReduce, which are inspired of Google papers on the MapReduce paradigm and Google File System [1, 3]. Hadoop is widely used by tech giants ([5], [6]) for internal purposes.

SPARK Spark is also provided by Apache and designed for cluster computing, but takes a different approach to data processing than Hadoop. By using a special datastructure called a *resilient distributed dataset* (RDD), Spark improves upon limitations in MapReduce-based data processing.

Spark has APIs in Java, Scala, Python and R and an engine that supports general execution graphs. It provides SparkSQL for SQL and structured data processing, MLib for machine learning, GraphX for graph processing and Spark Streaming.

2.2 THE BOINC FRAMEWORK

2.2.1 *BOINC overview*

BOINC is based on a classical client-server architecture. To setup a new BOINC project, a researcher must setup a server based on Apache, PHP, MySQL and supply various configuration files. To participate as a volunteer, a client program must be downloaded for the volunteers computing platform. From this client, the user can specify one or more projects to participate in [7].

To give out some task to a volunteer host, the following takes place

1. The client sends a request to the server for some workunit.

3 https://www.top500.org/system/178764
4 http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats2
5 https://siftery.com/apache-hadoop
6 https://wiki.apache.org/hadoop/PoweredBy
7 https://en.wikipedia.org/wiki/BOINC_client_server_technology

2. The server responds to this request with an appropriate workunit - the actual instance of the workunit is refered to as a result, which can have multipe states (e.g. in progress, finished).

3. When the client has completed the workunit, a result is returned to the server.The server processes the result in multiple steps

   a) A validator process checks the result for correctness. Multiple validation methods are available, e.g. Multiple volunteers might recieve the same workunit and have their results compared.

   b) An assimilator process processes the correct result in a project-specific manner, e.g. the result might be put in a database or simply be stored in a file in some location.

   c) A filedeleter-process then removes files related to this particular workunit and result.

2.2.2 *BOINC details and terminology*

As suggested by the above steps for solving an application, BOINC offers many ways to adjust how each of these individual steps are performed that reflect issues that the software designers has deemed crucial to address. These features thus provide a window into what must be addressed in a similar distributed computing system. To provide an overview of these features, I will describe them step by step.

2.2.2.1 *Defining a workunit*

Intially a project is created by creating its workunits. BOINC's workunits are based around what is coined an "application" - a set of variants of a program that computes the same function, but is compiled for different platforms. The platform typically refers to a combination of CPU architecture and an operating system (e.g. Linux/Intel32, Mac OS X/Intel64). A client may support one or more platforms (e.g. a Win64 client can also run Win32 applications). A specific compilation target program variant is called an app version. The app version may be associated with multiple files.

A workunit is created with a name, the application that will perform the computation (note that it is not an app version - the workunit is platform independent). Necessary input files may either be provided from a data server or by generating the file on the client (typically from an earlier instance of the application).

A result describes an instance of a computation, which may be unstarted, in process or completed. A result has an associated workunit.

Applications are created through an administrative web interface. An application is submitted with a name for directory naming and another name to show volunteers. A number of properties can be specified aditionally.

An application may be also be provided with more specific behavior than platform targeting. By "execution planning", a class of app versions can be defined that can only be sent to hosts with particular CPU features. Alternatively, a "fat executable" can be provided that itself detects if the CPU has the particular hardware and calls the best suited branch of the code.

2.2.2.2 *Requesting a workunit*

The client pulls workunits from the server. A host sends a request that demands a job of a certain difficulty, specified as the time in which a task should be able to keep all the host processors busy based on the host's previous behavior

2.2.2.3 *Responding with a workunit*

However, not just any workunit are necessarily returned as rules can be set for how to respond. A scheduler deamon is responsible for retrieving a suitable task. Various parameters can be set for which hosts should be able to recieve a job or how a job should be treated if the client is insuffient:

RSC_FPOPS_EST An estimate for the number of floating point operations needed to complete the job - an estimation of the runtime on a given host

RSC_FPOPS_BOUND An upper bound for number of FLOPS needed to complete the job. The job will abort itself if this is exceded.

RSC_MEMORY_BOUND An estimate of the jobs' largest working set size. The job is only sent to hosts with at least this much RAM available.

RSC_DISK_BOUND An estimate of the total disk storage requirements for the jobs' input, temporary and output files. The job is only sent to hosts with at least this much disk space available. If bound is exceeded, the job will abort.

RSC_BANDWIDTH_BOUND If set, the job is only sent to host with at least this much download bandwidth. Intended for large files.

If large input files are used for an application and many tasks make use of this file, it makes sense to send tasks to a limited set of volunteers that have recieved input, instead of sending the input files over and over again to new recievers. This is possible through the use of "loyalty scheduling".

2.3 THE HADOOP FRAMEWORK

Hadoop is based on the Hadoop Distributed File System (HDFS) for storage and MapReduce for data processing. The Apache YARN framework is used for resource management and scheduling. To provide an overview, Hadoop task processing is first described superficially, and each component is then described in further detail.

Hadoop is based around the functional programming concepts of a Map function and a Reduce function. The Map function is a function that takes two arguments, a function and a collection, and applies the function to every element of the collection, returning a new collection. The Reduce function accepts a key and a set of values associated with that key and merge these values into a smaller set of values, creating the final result. The functions are provided by the user, which allows the framework to perform all computations that lends themselves to being solved efficiently by application of the distributed map function and subsequent reduce.

The Hadoop infrastructure solves problems by assigning nodes the roles of Master, Mapper and Reducer. The single node that is assigned master is responsible for directing the processing. HDFS stores large files across multiple machines. To ensure reliability, the data is replicated. By default, three nodes store the replicated data; two nodes which are found in the same rack and one node in another. The Master assigns tasks to mapper nodes by their proximity to the data. Ideally, the node containing the data should apply the mapping function, but in case this fails, nodes in the same rack will be prefered. The results from the Master nodes are sent to a reducer that computes the final result.

2.3.1 *HDFS*

HDFS is a distributed fault-tolerant file system made to run across thousands of machines and for storage of terabyte-sized files. In the file system, files are kept in a distributed fashion, where a file is broken down into blocks of similar size that are distributed across the cluster.

A HDFS cluster consists of two types of nodes in a master/slave architecture. The cluster contains a single *NameNode* and multiple *DataNodes*, typically one instance per node in the cluster. The NameNode is responsible for managing the filespace and administrates access to files in the file system. It handles file system operations like open, close and rename of files, and assigns blocks to DataNodes. The DataNodes serve read/write requests from clients of the file system and perform block creation, deletion and replication, when instructed to do so by the NameNode.

By default, blocks are replicated as follows. One block is placed on a node in a local rack, and to select some remote rack and place a block on two nodes in that rack. This is based on a philosophy of minimizing inter-rack network communications, compared to spreading the blocks across three racks, while still maintaining fault tolerance.

2.3.2 *YARN*

YARN is designed out of a philosophy to keep behaviors associated with resource management and job sheduling/monitoring respectively seperated, such that a deamon is used for each behavior ([8]. Ressources are monitored and dynamically assigned by the interplay of two types of managers and a master component with different responsibilities. A global RessourceManager(RM) is used to arbitrate resources, while an ApplicationMaster exists per application. The RessourceManagers main components are a Scheduler and a ApplicationsManager. Additionally, A NodeManager runs on each machine and monitors the container's resource usage and forwards the information to the RessourceManager. The ApplicationMaster is a framework specific library that negotiates resources, represented by containers, with the RessourceManager, and collaborates with the NodeManager to execute and monitor jobs.

The RessourceManager's Scheduler component allocates resources to running applications. The requirements of an application is expressed through the container that has elements such as memory, CPU, disk, network and more. Various scheduler variants can be provided for different prioritization.

The ApplicationsManager (not to be confused with the previously mentioned Application*Master*) recieves the job-submissions and negotiates the first container for running the ApplicationMaster. Additionally, it is responsible for restarting the ApplicationMaster on failure.

YARN has a feature called resource reservation, where the user can setup a temporal profile of resources and set deadlines for jobs, such that resources are allocated to meet a predictable execution of important jobs.

The Scheduler allocates resources to running applications subject to familiar constraints of capacity/queues. However, it does not monitor the status of a task and it does not guarantee recovery from failure. The Scheduler has a pluggable policy, meaning that its scheduling policy can be replaced by other policies. A number of such exists.

8 https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

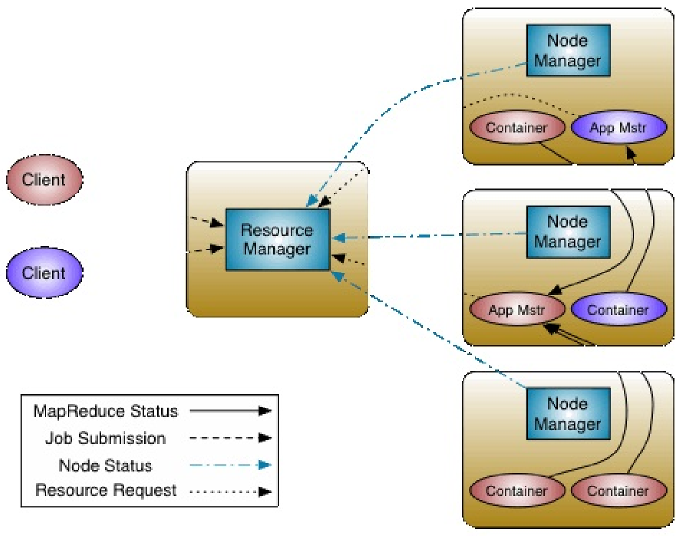

Figure 2: Clients submit jobs to the RessourceManager (ApplicationManager component). MapReduce statuses are sent from containers to the ApplicationMasters. The ApplicationMaster send requests for resources to the RessourceManager (Scheduler component). Statuses are sent from the NodeManagers to to RessourceManager. (Source: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html)

2.3.3 *MapReduce*

The MapReduce framework is built around processing of job and its configuration, which consists of parameters such as an input and output location (typically, in a HDFS filesystem) and by providing a map and a reduce function. The data in the input location is split and the resulting chunks are processed in parallel with the map function. The output of the function application is then sorted and becomes the input for the reducer function. The output of the function is stored in the output location. The parallel processing relies on YARN, such that a single master ResourceManager, a NodeManager per node in the cluster and a MRAPPMaster per application is used. The Hadoop job client submits the job and configuration to the ResourceManager, that assumes the responsibility of job distribution, scheduling and monitoring, which is forwarded to the client.

2.4 APACHE SPARK

The Spark framework is built on top of the same infrastructure as the Hadoop framework, as it integrates with YARN and HDFS, and it is similarly designed for cluster computing. However, Spark takes a different approach to executing computations. Spark is motivated by the observation that other frameworks have a very narrow scope in capacity and big data analysis usually needs processing by many systems, where the data has to be transformed into a suitable format before input to each. Spark provides a uniform infrastructure with many capabilities where the data can be kept in a consistent form and the user can easily express the various steps in the data processing. It is based on what the designers call *applications*. An application consists of a driver program that calls a user-provided main-function and runs parallel operations across the cluster. Spark's main selling point is the features of resilient distributed datasets that represents the data that is modified by an application. An RDD is a collection of elements that is partitioned across the cluster's nodes and can be operated on in parallel.RDDs are created from preexisting data, either in form of a HDFS file or a Scala collection in the driver program. An RDD is simple to persist in memory so that it can be reused across parallel operations and can automatically recover from node failure. RDDs are partitioned into one partition per file block, which are 128 MB by default in HDFS. RDD's can be operated on with two types of operations *transformations* and *actions*. A transformation applies a function to every element in a dataset, returning a new dataset in the form of a new RDD, while an action aggregates all the elements in an RDD and returns the result to the driver. Transformations are lazy in the sense that they are first applied when a result should actually be returned, i.e. when an action is applied.

Similar to conventional database engines, Spark creates a graph of the transformations and creates an execution plan. Multiple transformations are run over data in-memory, such that computationally expensive read/write operations to a database are avoided.

Conventionally, distributed computing systems are made fault tolerant by data replication, such as in HDFS, or check pointing, where a snapshot of the system's current state is saved. Spark uses another concept coined "lineage". The framework remembers the transformation graph, such that each operation that has been applied to some base data to create the current state are stored. If data in the current state is lost, the steps can simply be repeated on the base data. It is more efficient than replication for data-intensive purposes as writing data over the network is much slower than writing RAM and storage space in memory. The recovery will be faster than re-running the program from scratch as a failed node contains multiple RDD partitions that can be built in parallel on other nodes.

Spark is designed to be run on clusters and can be deployed using a cluster manager based on its own implementation, YARN or Mesos, which is responsible for allocation of resources to applications. A spark application consists of an independent set of processes on a cluster. The main program (the driver) has a SparkContext object that coordinates these. The SparkContext connects to the cluster manager and acquires executor processes on the nodes in the cluster; these are used by the application to run computations and store data. The SparkContext send these the application code as a JAR or Python file.

3 ANALYSIS

A successful piece of software is often characterized by being well-designed in respect to meeting the demands of the users and being pleasant to use. While these are vague and general descriptions, a system that does not perform well, i.e. is slow or unreliable, is generally not pleasant to use. A framework for distributed computing is subject to some specific challenges that must be approached appropriately. In this section, it is identified and discussed how the introduced frameworks meet performance and fault-tolerance goals. The discussion creates a basis for the following section where the design a of a framework based on this section's findings is presented.

3.1 PROBLEM PARALLELIZATION APPROACH

The central issue of this thesis is parallelization of tasks and it is the core functionality of the presented frameworks. As expressed by Amdahl's law [1], the theoretical speedup by parallelization is restricted by the parts of the task that do not benefit from parallelization. Thus, task processing can only become as fast as the computation model allows for, so it is a very fundamental choice how this is performed.

BOINC has a straightforward approach to how it archieves parallelization, as it has a database of available tasks provided by the owner of the task server. When a client is available, it sends a request for some amount of work and the task server will find a suitable task in the database. The client recieves the task and executes it whenever the user of the machine is idle.

Hadoop relies on the computational approach presented in [1]. The MapReduce paradigm was designed to simplify large-scale parallel computing based on the observation that a large number of the cluster computations they performed at Google could be expressed in terms of Map function application, followed by Reduce function application. The MapReduce computation model takes set of input key/value pairs and generates a new set of output key/value pairs. The user provides a Map function and a Reduce function. The Map function accepts an input pair and creates a set of intermediate key/-value pairs. The intermediate values are grouped together by shared intermediate keys. The reduce function accepts an intermediate key and a list of values (figure 3).

Practically, the different problem solving approaches used in BOINC and the cluster frameworks lead to different utilization of available re-

1 https://en.wikipedia.org/wiki/Amdahl's_law

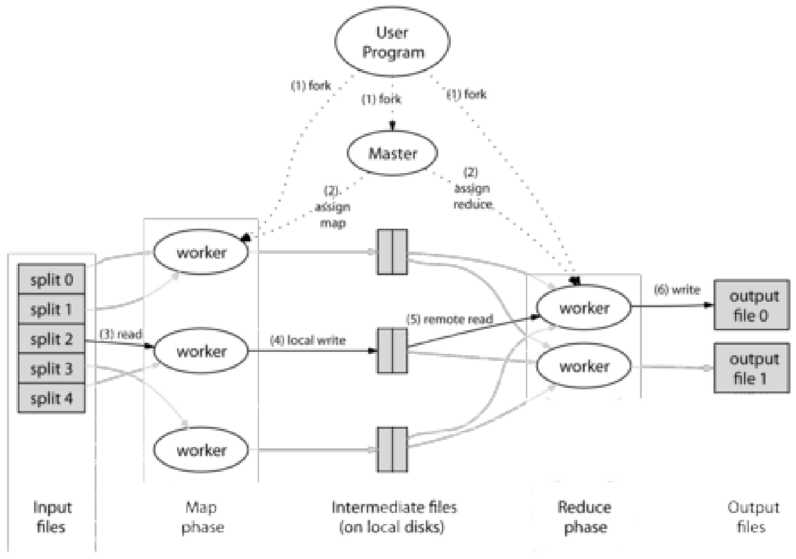

Figure 3: Overview of computations in MapReduce. First, split files are provided to the workers that apply a map function to create intermediate keys. Then a reduce function is applied by another worker to create an output file.

sources. In a BOINC setup where n tasks are available for solving and m clients are available, where $m > n$, up to n clients will be running a task, e.g. if a single task has been provided by a researcher and 100 volunteer machines are available, only a single machine can be put to work. In Hadoop, a single task will be parallelized across multiple machines, so more units can execute computations and the task will likely be completed in shorter time, even though there is an extra overhead of splitting the result, shuffling the map outputs and applying the reduce function to these.

One drawback of Hadoop and Spark compared to BOINC is that it has a higher complexity of usage and more parameters that must be configured for best performance, which requires understanding of both the HDFS, YARN and MapReduce framework. BOINC is conceptually simple and is thus easier to develop a correct mental model of. Both approaches are characterized by a high degree of user involvement, i.e. in BOINC, the user has to provide a characterization of the tasks that can be used for scheduling while Hadoop requires the setup of a cluster with various configuration files.

3.2 OVERVIEW OF CORE FUNCTIONS

To break down "performance" in regard to this type of framework, I will provide an analysis of sources of overhead and how the above frameworks approach these.

There are some essential functions that by definition must be present in such a framework

- For machines to collaborate they must be connected by some medium, i.e. a physical or wireless network
- Tasks including data must be provided somewhere but spread across machines, also by network
- A task must be executed by a machine, using the CPU and taking up memory and disk space
- The system must be fault tolerant - there are many ways to archieve this

In short, to understand how a framework achieves performance, it is necessary to identify which computationally expensive operations that cannot be avoided and how to minimize the overhead associated with these. Below it is discussed what can lead to performance deterioration in the emphasized frameworks.

3.2.1 *Network communication*

The World Wide Web is based on the TCP/IP protocol that contains the essential network protocal stack described in the Open Systems Interconnection Reference Model (the physical layer, the data link layer, the network layer and the transport layer). Communication is based around *packets* that contains the data necessary for routing along with the payload data that the transfered information is broken down into.

The time it takes to transfer data between a sender and a reciver over the internet is a combination of bandwidth and latency. Optimal data transfer takes place when there is a smooth continous flow of packets. If there are large and variable delays in packet arrival, performance suffers. Latency is largely a result of the length of the route data has to travel between sender and reciever and the interaction between TCP reliability and congestion control protocols. Key sources of latency are propagation delay, serialization, data protocols, routing and switching and queuing and buffering. Propagation delay is the primary source of latency usually. It refers to how long it takes light to travel through the communication media. Serialization delay occurs when bytes in memory are converted into a serial bitstream. Handshakes are used in various layers to update link status and error correction between sender and reciever. In TCP/IP, a

IP-packet, the unit of communication, reaches its intended reciever by *routing* through a series of IP routers or switches. An outage or congestion along the path can change the routing path which can affect the latency. In the transport layer *queueing latency* may occur, where an outgoing IP packet is queued and awaits transmission due to overutilization of the outgoing link.

In addition to inherent limitations of the nature of networking, users in a volunteer system are usually subject to bandwidth throttling and data capping by their internet service providers.

BOINC uses HTTP for all communication, including sending of tasks[2]. In BOINC it is possible to use compression both on the application that is provided or on overthing transferred from server to client by using gzip or deflate. Similarly, Hadoop supports compression for intermediate map-outputs and job outputs and is bundled with various compression codecs.

As clusters are usually connected as a local area network, a framework intended for clusters is expected to deal with a much lower latency than wide-area networking infrastructures, as packets has to travel shorter physically and in addition are subject to less routing. However, Hadoop and Spark also integrate with cloud solutions such as Google Cloud[3] and Microsoft Azure[4] where latency may become relevant. Also, intra-cluster bandwidth in a rack is generally superior to inter-cluster communication as the data must pass through switches between the racks.

The presented frameworks have little control over the sources of latency, as it is largely a result of how the internet works and the fundamental laws of physics. However, a framework can avoid network communications whenever possible. Both Hadoop and BOINC operate on a philosophy of "send computations to data", as tasks will likely be considerable smaller than the datasets that it is applied to.

BOINC uses a mechanism called *locality scheduling* that can be enabled. When this is used, if several tasks depend on the same dataset, it makes sure that tasks are sent to a limited set of nodes, that have recieved the data already, instead of sending the data again and again to new nodes.

Hadoop does something similar with *rack awareness*, where task data is distributed across the cluster using HDFS and tasks are sent to the nodes that have the data that the task needs.

3.2.2 *Data representation*

When sending Java a object over the network or saving it to storage, it must be converted into a suitable format that can be converted back

2 https://boinc.berkeley.edu/trac/wiki/CommIntro
3 https://cloud.google.com/hadoop/
4 https://azure.microsoft.com/en-us/services/hdinsight/

into an object. This is refered to as *serialization* and *deserialization*. In serializing the object, it becomes a stream of bytes that contains the state of the object and relevant metadata for the object's class.

Both Hadoop and Spark are based on Java. Spark accepts tasks in Java, Scala (based on the JVM) and Python. The Spark tuning guide[5] describes how Spark provides several alternatives for serialization.

JAVA SERIALIZABLE Java is built with a *Serializable* tagging-interface that the classes provided by the user must implement for the JVM to serialize it. Behind the scenes *ObjectOutputStream* is responsible for creating the byte stream. Java's own serialization is slow and bulky though.

KRYO As a third party alternative, Spark makes Kryo available [6] for serialization. Kryo is considerable faster and compact, but also has some drawbacks.

3.2.3 *Storage*

A computer usually has two types of storage; Random-access memory (from here on simply "memory") used for programs that are currently used by the processor and disk memory for persisting data and programs. One of the main characteristics of memory is that it is very fast to both write to and read from and speeds are independent of physical location. Accessing a byte in RAM takes is measured in billionths of a second, or nanoseconds. On a disk this takes thousandths of a second, or milliseconds The main memory and disk memory are frequently used in interplay. Most operating systems rely on virtual memory , where parts of the hard drive are dedicated to paging. When the system runs low on physical memory, the hard drive is used for *swapping*, where some of the RAM's data is written to the disk. In light of the presented facts, this is unattractive in regard to performance. Even worse, if the machine runs out of memory, it might not be able to complete the task. Thus, if it is possible, a task should not be run on a machine where it cannot fit in RAM.

In BOINC, the hardware demands of a task is provided manually with the task. The scheduler will then use this information to avoid sending jobs to machines that cannot match these demands. Spark is highly effective because of the way in which RDD's are designed for in-memory computing.

3.2.4 *Scheduling*

Given a number of available machines, choosing which one to use for a given task is comparable to the challenge faced by the scheduler

5 http://spark.apache.org/docs/latest/tuning.html
6 https://github.com/EsotericSoftware/kryo

component of an operative system that matches CPUs with processes : given a number of processes and a number of processors, it is the function of the scheduler to match these to effectively meet some goal. Similarly, a frameworks must match tasks that will be run as processes with the available units according to some philosophy. While operative system scheduling is designed to efficiently meet the ad-hoc demands of the user efficiently, the goal of a parallel computing system would be to complete a task as quickly as possible. Additionally, task scheduling has a higher complexity as attributes of the task, the hardware on a given machine, storage, network latency and bandwith and other factors must be taken into consideration. Clusters are however usually homogenous in hardware and connection speed, so machines are equally good, but may be subject to different loads at any time. They are also dedicated to computing, not for human usage, so they are free to hog resources for longer durations of time.

In BOINC, a client requests a workload of a certain demand. To respond, the Scheduler loads the available tasks from a database and filters them according to a number of parameters, of which those important for performance are that disk and memory requirements are met by the host, if expected latency exceeds the workunit's latency bound.Additionally, the scheduler takes into consideration the locality scheduling described previously.

In Hadoop and Spark relies on scheduling is outsourced to YARN.

3.2.5 *Fault tolerance*

A computer consists of several components that may fail due to wear or errors in manufactoring. Additionally, network or power shortages can occur. All scenarios has the same consequence that the machine becomes unavailable if it "fails badly" (in contast to "failing well", where measures are put in place to avoid total failure) ([7]). It is safe to assume that as the number of machines in a network increases, the risk of failure of any instance increases too. Additionally, in a volunteer-based system, a user may at any point in time decide not to volunteer anymore. Thus, reliability engineering through fault tolerance measures is an important part of the design of these frameworks.

In BOINC, clients use *checkpointing* where the current state of the task execution is saved, so it can be resumed from there later on. However, if a user decides not to do this means that the task must be re-computed from scratch. In Hadoop and Spark fault tolerance is to a large extent provided by the previously described workings of the underlying HDFS and, for Spark, additionally RDDs.

7 https://en.wikipedia.org/wiki/Failing_badly

3.3 SUMMARY

While Hadoop is fairly complex, it provides a superior way to perform computations compared to BOINC, as BOINC only offers limited parallelization.

Much of the performance is however up to the user to provide, as scheduling relies partially on the demands of a task, which needs to be provided by the user.

A general observation from the design of YARN, HDFS and MapReduce is that a single node is typically assigned to monitor the resources of the application.

There are other dimensions of performance, such of security, that I have not addressed, where design choices could also affect computation time (e.g. secure HTTP communication is based on the SSL protocol that requires handshakes to be performed and data to be encrypted and decrypted).

4 DESIGN AND IMPLEMENTATION

In this section I provide an overview of the alternative framework that I have implemented. The considerations and implementation of key aspects of the application are presented below.

4.1 THE TURBO FRAMEWORK - INTRODUCTION

In essence, the Turbo framework provides a Hadoop-like network infrastructure using a simplified MapReduce approach, designed for parallelization and parallel distributed processing over WAN using the TCP/IP protocol. It applies the superior MapReduce parallelization approach to the heterogenous computation problems that BOINC is applied for. The fault-tolerance measures in Hadoop and Spark are based on the underlying distributed file system of the cluster, which is a mechanism that cannot be made us of in this framework, so instead I have provided my own fault-tolerance mechanism.

The Turbo framework is a Java framework for solving general tasks in parallel by using a simplified approach to the MapReduce paradigm. The framework supports all tasks implemented in Java (in which only the standard library is used). The framework solves a problem by splitting an associated dataset into lesser parts and applying the same function to all of these. However, the user can also add data-less tasks which will be distributed across different computers and solved sequentially by providing a splitsize of 1 for a task - the splitsize refers to the number of subtasks to create. A parallel-solvable task is solved in 3 steps

1. The associated data is split into smaller datasets by a scheduler node using a split function

2. The data-splits are sent to worker nodes that apply the map function to create a subresult

3. The subresults are sent to reducer nodes that apply the reduce function to create a final result

In the network, there is exactly one scheduler node, while multiple worker nodes and reducer nodes may be present.

4.2 TERMINOLOGY

To precisely describe node interactions and computations, a number of terms are used to refer to different components of the framework and their input and output. Their exact semantics and context is described in the following section. The *scheduler* is the coordinator of task processing and matching of tasks with workers. *A task* is provided by the user with a map and reduce function and is parallelized by being split into *subtasks*. The subtasks are solved in parallel by multiple *Workers* that are the nodes that apply the Map function. When a subtask has been processed, a *subresult* is created. The subresults for any given tasks (one subresult for each subtask) is sent to the same *reducer*-node, that applies the Reduce function on the subresults to create a final *result*. For shortness, a scheduler node will simply be refered to as a scheduler, and similarly for worker and reducer nodes.

4.3 MAPREDUCE SIMPLIFIED

The MapReduce approach decomposes a problem into two stages of conclusion. First, data is spread into smaller parts that can be independently analysed to draw some local conclusion by applying the Map function. Then data is grouped together by keys and subject to the reduce function to reach a global conclusion about the dataset.

To simplify this, I have removed the intermediate-value grouping aspect from the implementation. A problem is solved by splitting the data, applying the map-function to the data subset and sending the subresult to a reducer that applies the reducer function, when all subresult have been recieved, to output a final result.

4.4 PROVIDING A TASK

There are two challenges associated with implementing handling of tasks. First, for a worker to be able to execute a task, it must be provided with an implementation of the task that is executed on some data, i.e. a map function in the MapReduce paradigm. Similarly, the reducer role nodes must be able to execute a reducer-function argument. However, Java does not support first-class functions. Secondarily, Java is strongly typed, but the user-provided function needs to accept a list and output a single value.

In programming languages where functions are first-class citizens functions can be provided as arguments for other functions that can then apply those functions to other types or data structures. In a programming language like Java, there are no isolated functions; a function is always associated with a class as a method that is made available by the instantiation of that class (or directly callable with no instantiation in the case of static methods). The common design pattern to mediate this problem is termed the *Command*-pattern [2]. In this pattern, an interface specifies a Command by a single execute-method. When desiring a variable function argument to a method in the code, we represent this by the interface in the method signature and call the execute method to obtain some result in the invoking method. Concrete implementations of the interface are then used to provide various functions, such that a execute-method that accepts two integer argument might be implemented in a SumCommand to return the sum of these and in another MultiplicationCommand as the product of the integers. In short, the command pattern makes it possible to simulate function passing. I use this pattern to represent both a map and reduce function to be applied in the task, through the *Function* interface.

One of the main goals of the implementation is to be able to execute a generic variety of tasks with the the single requirement that they are embarassingly parallelizable and can be processed MapReduce-wise. Java is characterized by its static typing system. The consequence of strong typing is that every value in Java is either one of the predefined primitive types of the language or an instance of an object that inherits from the basic Object type. A method signature species exactly which value type the method returns and what argument types and cardinality is expected. This is enforced by the compiler at compile-time.

As highlighted in the discussion of the command pattern, interfaces are one way to archieve variability, as any object that implements an interface can be used where the interface is declared as the type of a variable. There are however scenarios where an interface declaration is not vague enough typewise. As every object in Java inherits from the Object type, it means that every non-primitive value is an instance of Object (with the exception of null, that has its own type). The practical consequence of this is that by declaring a value with the Object-type, any complex type can be used in its place. If the need arises for a primitive value, which is the lone exception that does not extend Object, autoboxing can be used to represent that value as an object. I use this as well as the Collection class, that all Java collections inherit from in the method signature and in the framework code that comes into contact with the task interface. The framework code does not need to know the actual types. However, the user must use proper casting in the provided functions or the task will likely fail on run-time.

Appart from the functions, a user must provide a textual name for the task, a method to get the data that should be split, and a number for the amount of subtasks to create from the data. Below is provided the interface for a task.

```
public interface Task extends Serializable {

Collection getData();

Collection<Collection> split(Collection data);

String getName();
Function getMapFunction();

Function getReduceFunction();

int getSplitSize();
}
```

4.5 SCHEDULING

As discussed in the analysis of preexisting frameworks, it is a challenge to split and distribute tasks in such a way as to make ideal use of the available distributed ressources in a heterogenous node infrastructure where tasks at any point may be provided and nodes can be in various states of occupation, depending on their current workload.

The framework has been designed in such a way that the task distribution function is built around the *Strategy* design pattern, such that it is easy to evaluate various scheduling strategies. In the Strategy pattern, an interface with a single method specifies the signature of the algorithm by defining the input as parameter types and the output return type. To provide the variability, the interface method is called wherever the variable algorithm is needed and the user is then free to instantiate an object with any implementation of the interface. In my implementation, the strategy algorithm is provided with a list of tasks and a list workers and is then up to the strategy to do this satisfactorily.

As the framework is intended to be used for heterogenous computing using WAN, scheduling is not trivial. To test different approaches to scheduling, the worker nodes are continously evaluated. Whenever a node finishes a subtask, i.e. has executed the map function on the provided data, it notifies the scheduler of the completion time. When the scheduler has recieved completion times from all workers associated with a given task, it computes the average time and compares every workers time with the average. If a worker has solved more subtasks associated with the same task, every subtask gets a rating. These ratings are saved by the scheduler node thoughout the lifetime of a worker node.

For scheduling, this information can be used in various ways according to scheduling philosophy. The overall goal is to make best use of ressources at any given point in time and to complete tasks as fast as possible, but there are many strategies that can be applied to archieve this. Should the scheduler be designed with a bias for nodes that on average perform better than the average worker? Or should the scheduler instead only look at the n most recent evaluations, as a work may start out strong and deterirate in performance? Similarly, it may be better to prefer a stably average node than a node that fluctuates between good and bad. In deciding which worker to use, the algorithm may also consider the subtask. A subtask may be fairly fast to complete, so it does not matter much which node is the recieving node, as the overhead contributes very little, but for a demanding job it may be critical. On addition to these considerations, the scheduler must also take the nature of the task, the locality of a worker node and possibly even locality of all workers associated with a given task and the reducer assigned to the task. If the workers allow for multithreading, scheduling becomes a more complex situation as workerload is no longer a matter of being occupied or not occupied, but rather becomes a matter of the degree of occupation.

A scheduler could be designed with various degrees of conservatism, i.e. How willing it is to send tasks to workers that have not yet been evaluated. A scheduling strategy may prefer workers with a good rating, but infrequently use a unknown worker, in the hopes of the unknown worker providing a good performance. Alternatively, the scheduler could initially evaluate all nodes on some benchmark subtask instead of having to wait for each node to complete a task. While these scores may change, it provides a starting point for identifying optimal workers.

It is also worth noting that while a node is typically described as either running or crashed, there are likely states of the system from which failure can be forecast. While a machine may become abruptly unavailable due to loss of power or internet connectivity, hardware failure will often manifest itself in reduced performance or other symptoms through a period of time before fully giving out. A scheduling strategy based on recent performance on subtasks would prioritize other nodes over a slowly failing node if failure symptoms lead to consistently poor performance.

While the evaluation mechanism is implemented and running, it has not been possible at the time of deadline to evaluate various strategies and the implementation does not currently make use of the strategy pattern. The framework instead uses a eager default scheduling strategy where it initially spreads subtasks across all available workers and if there are more tasks than workers, it will send the excess tasks to randomly selected occupied workers, that will then enqueue the subtask until the currently executing subtask is completed.

4.6 THE TASK LIFECYCLE

To put the above information together and describe various details, that did not fit into the other sections, I will now describe the processing of a task and how it affects the state of the various node types.

When one or more tasks has been provided, they become instantiated and placed in a queue for another thread for processing. This is a classical producer-consumer problem where one thread must wait for elements to be put in a queue by another thread. To synchronize these threads, Java's native monitor object mechanism is used. When the task-processing thread becomes active, it tries to acquire the monitor, but fails. The task-instantiating thread then notifies the waiting thread, when tasks become available.

As a consequence new tasks can continously be provided, while the scheduler is running.

A Task-object is constructed with a split-size argument, that specifies how large a split that is desired, i.e. how many subtasks should a task's associated data be broken into. To archieve the goal of generality, the general Collection type is used, while the task implementer is responsible for doing the necessary type casts in his code.

When a task is split into subtasks, the tasks and its subtasks are assigned permanent ids using Java's UUID class, which makes collisions between assigned ids unlikely ([1]). For easy lookup later, a hashmap is used to store this relationship, such that every subtask id is a key associated with the original task's id as value; hashmap have generally been the data structure of choice, as they provide a simple way to associate a pair of values and look up of that relationship.

The subtasks are converted into message objects that contain the id of the task and subtask, the split of the data, and a reducer, for which to send the result of the map function application. Note that a reducer is associated with a task, not just the subtask, as every subtask from a given task must be processed by the same reducer. The message object is provided with a type, such that the worker can distinguish it from other types of messages. The scheduling strategy is invoked for sending and the subtasks become spread across workers. The message object is serialised using Java's own ObjectOutputstreamImplementation. Note that as a consequence every class that appears in any type of message implements the Serializable interface. It is additionally stored which workers that have recieved which subtask(s). The scheduler has a representation of every worker in the form of a WorkerInfo object that contains the IP-address, port and id of the worker. The WorkerInfo object is updated whenever task-related communication is exchanged. When the scheduler sends a subtask to the worker, it is stored that the worker is responsible for this. When a worker finishes a subtask, it is stored that it has been finished and the worker evaluation is updated when all subtasks belonging to a given task has been completed. When a task as a whole has been completed, the WorkerInfo is also updated.

The main thread of a worker is running a ServerSocket that listens for new connections. When a connection is established, the socket is

1 UUID.getRandom() returns a so-called version 4 UUID. See https://en.wikipedia.org/wiki/Universally_unique_identifier

provided to a SocketHandler thread where the message object is read using a the objectInputstream for de-serialization.

The incoming message is enqueued, similar to when tasks are instantiated by the scheduler. While the worker has been designed with capacity for running multiple subtasks concurrently, its default behavior is to run subtasks sequentially such that a single subtask is processed at a time.

When a subtask completes, a message is sent to the scheduler with the time that the worker used to complete the task, along with the worker id and a message type. The scheduler handles incoming message similar to the workers, and uses the message type for processing the message correctly. For every task, a timekeeping object is created that stores a list of all the recieved times for the subtasks. When the final subtask-finished message is recieved, which is determined by the size of the list of recieved times, the WorkerInfo objects of every worker who contributed to this task's solving have their evaluations computed.

The subtask's result is sent to the reducer that has been assigned to the task when the worker has finished the subtask, but the worker will actually store the result for longer, which will be explained in fault tolerance section. When the reducer recieves a subresult, it stores the value in a list in a hashmap, where the key is the id of the task. Thus, the size of the list is compared to the splitsize whenever a subresult is recieved and if the list size matches the splitsize, it must mean that all workers have completed their part of the task. The reducer can then apply the reduce function, similar to how the map function is applied and compute a final result, which is then sent to the scheduler.

4.7 COMMUNICATION AND COORDINATION

Communication is based on the TCP/IP protocol by using sockets. Every node listens on a specified port for incoming connections. Message is exchanged by using a Message object that contains fields for message type, sender node and a generic data-Object for providing additional information that can be cast according to the message type. The scheduler only sends messages to the workers, the workers send messages to both the scheduler and reducers, and the reducers only send messages to the scheduler. The types of messages used are

NEW TASK Sent from scheduler to worker. Contains a subtask for the worker to execute that contains the data and the map and reduce functions. Additionally contains the reducer to use for the task.

SUBTASK COMPLETED Sent from a worker to the scheduler. Informs the scheduler of the completion time for a subtask that the worker has executed.

REDUCE TASK Sent from a worker to a reducer. Contains the reduce function and the result that has been computed by application of the map function.

TASK COMPLETED Sent from a reducer to the scheduler. Informs the scheduler that the task has now been completed and provides the result.

REDUCER CRASHED Sent from the scheduler to the workers. Contains the id of a crashed reducer, that the worker was supposed to send subresult to, and provides the IP-address and port of another reducer to use instead.

HEARTBEAT Sent from both the workers and reducers to the scheduler. Informs the scheduler that the node is healthy.

The message types are designed to be minimal to avoid unecessary serialization and sending of irrelevant fields.

4.8 FAULT TOLERANCE

One of the main goals of the system apart from creating and solving problems in parallel is to withstand failure of a node and to be able to complete a task, even though one of the nodes, worker or reducer, that have been designated to work on a the task, becomes unavailable. Fault tolerance is typically based on replication, where multiple copies of data exists multiple places, such that the data can be retrivied even if a source should fail. I have built a fault tolerance model where the scheduler saves subtask-related information and the workers save subresults until a final result for the task has been obtained.

Both a worker and a reducer node is associated with a heartbeat, meaning that a Heartbeat-message is sent to the scheduler node regularly. This allows the scheduler to detect failure of a node by the lack of heartbeat. When a heartbeat is recieved, the scheduler stores the current time. The scheduler iterates over a list of workers and reducers and compares the last time of recieving a heartbeat with the current time. If too much time has passed since the last heartbeat, the worker or reducer is assumed as crashed. It is then removed from the pool of available nodes and does not become assigned to anything. The reducer nodes are a associated with a heartbeat for as long as they are available. A worker only sends a heartbeat when it is actively processing a task, so the task, that the worker is responsible for, can be quickly re-sent to another worker. If a worker is not processing a task, the scheduler will discover that it is not available the next time when it attempts to send a task to the worker, and can quickly redirect the task somewhere else. This means that when a worker completes all subtasks that it has been assigned, it is removed from the pool of workers to check for a heartbeat. If a reducer becomes unavailable on the other hand, we cannot wait for a worker to detect it, as the reducer is a bottleneck for completion of all tasks that have been assigned to this reducer. Instead, it is import to discover it as quickly as possibly, so the tasks for the reducer can be redirected. Therefore, the reducer provides a constant heartbeat.

To describe the failure scenaries more precise, I will discuss the consequences of each type of node crashing in detail and describe what the recovery mechanisms are.

Before a task is sent, the scheduler stores metainformation about the task. The Map and Reduce functions as well as the data sent in a given split is saved. The original task object is saved as an object that can be looked up by its id, and the subtask data is saved in another object that also contains the id of the task it was spawned from.

The scheduler has a object representation of every worker and that contains a list of the ids for the subtasks that the worker is currently working on (i.e. any task it has been sent but not completed). When the task leaves the scheduler, it is also already assigned to a reducer that it will be sent to, which is also associated with an id that is stored.

When the worker starts processing a subtask, its heartbeat becomes active. If a worker crashes, the consquence is that it will not finish what it was working on - these tasks will have to be re-sent to one or more other workers. Additionally, no new tasks must be sent to that worker, so it must be removed from the set of available workers in the scheduler node. The heartbeat is saved with the id of the node from which it came, so it is simple to look up the corresponding the worker information object and get the active subtasks from there. These are then resent to other notes that must process these from scratch. As these have the correct reducer information, every task will end up at the same reducer as every other task from that split.

If a Reducer crashes, it cannot recieve and process anything from workers and everything that it was processing currently is lost. This means that all subtasks from which a subresult has been reduced would have to be redone, if no fault tolerance mechanism was in place. It is likely that a worker will try to send a crashed reducer a message, before the failure detection thread in the scheduler discovers the crash. In that case, the worker has an outgoing tasks queue to which it is saved along with the id of the reducer. When a worker succesfully sends a reducer a message, it also stores the information locally in a result queue.

When the failure detection thread discovers that a reducer has crashed, it must reinstate another reducer to recieve the subresult from the workers and remove the crashed reducer from the set of available reducers. The scheduler randomly selects another reducer. It then notifies all workers that eventually would try to send or already have attempted to send a subresult to the crashed crashed reducer. The recieving workers update the reducer field of all enqueued subtasks and looks up the crashed reducer id in the outgoing queue. If there are any subresult stored, they are then sent to the new reducer and removed from the queue. If a worker has already sent a subresult to the crashed reducer, the worker must look up the result in the result queue and resend the result to the new reducer. When a reducer completes a task, every worker who sent results to that reducer is then free to delete its backup copy of the subresults.

5 EVALUATION

To evaluate the the performance of the framework and whether the goals of the implementation have been met, a number of experiments have been performed.

5.1 EXPERIMENTAL SETUP

All measurements are performed by running all nodes on the same computer and all measurements have been performed in triplicate. Measurements graphs are based on mean with standard deviation.

5.2 EXECUTION TIME VERSUS TASK SPLITTING AND WORKER SCALING

To examine the performance gains of parallelization and the effect of splitting a problem into a different number of subtasks, a series of evaluations have been made. A computationally challenging task was provided, as the map task computes the Fibonacci function repeatedly on the same number, using the BigInteger class. The map function argument is a list of values (the number 20 repeated a thousand times) for which to apply the Fibonacci function. The computation have been performed by using a combination of various split sizes (1, 10, 100 and 1000) and number of nodes (1, 10 and 1000). The result is presented below.

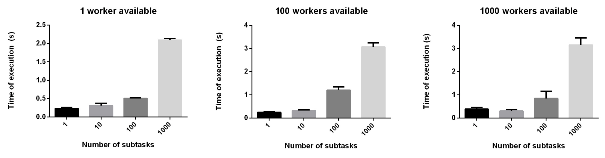

Figure 4: Task processing time as a function of number of available workers versus number of subtasks/task. Note the different Y-axis values for the graphs.

As can be seen in figure 4, for every split size, task execution time increases as a function of the number of workers. A consistent slowdown in performance is seen with an increase in the number of subtasks, independent of the number of available workers. Additionally, task processing becomes slower overall as the number of workers are increased. Using a single worker appears the most effective strategy for all split sizes. However, even the fastest scenario is more than ten times slower than solving the task locally in a single thread (data not shown).

5.3 EXECUTION TIME VERSUS TASK SPLITTING AND WORKER SCALING WITH A HARD TASK

To see if the above results could be explained by a larger overhead than the gain of parallelization, the same experiments were performed with a more demanding task. The same map function as in the previous experiment, but applied to list of size 1000 containing the value 40 repeated. Due to some issues (see discussion), not all experiments could be repeated.

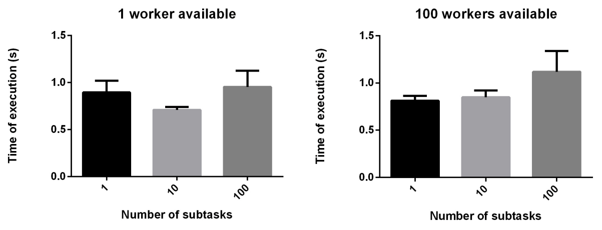

Figure 5: Task processing time as a function of number of available workers versus number of subtasks/task. Note the different Y-axis values for the graphs.

When the number of workers become too high (1000) or the split size, computation failed. However, for the measured data resembles the pattern of the first experiment.

It was expected that parallelization by increasing the number of available workers would lead to a significant speedup in task execution time, where execution time would inversely correlate the extent to a task is executed in parallel. The effect of varying the number of tasks is indeterminate and depends on the task, as there is an added cost of splitting and sending more subtasks, but intuitively, these aspects of the application should not add significant overhead to the overall computation. What was observed, however, was that even when the number of subtasks for processing was constant, just having a larger available pool of workers affected performance negatively. There are several reasons that could explain the unexpected result.

It is possible that thread creation is at fault for the performance issues. Each time a node recieves a request in the application, a new thread is spawned, in addition to the pre-existing threads. Whenever a thread is spawned in Java, it needs allocation and initialization of a new thread stack, registration system calls to the native OS, and creation of descripters for the internal data structures of the JVM. One way to improve upon this aspect of the application is by the use of thread pools, where threads are reused, such that the above does activities does not have to take place.

Another problem is that race conditions became evident, when the system was evaluated. When race conditions occur, multiple threads share the same ressource that they simultanously modify. As an example, two threads might work on shared empty list. They both read the list as empty and write to the first index at the same time. Thus one of the write values is lost. Simultaneously, it was observed that when using a large number of workers or a large split size, it appeared as if tasks were lost, as the reducer would recieve some subresults, but never enough to apply the reducer function. This suggests a race condition somewhere in in the implementation. It is possible that the race condition occurs in the worker, as it adds incoming tasks to its processing queue, as it was observed that for a large split size in a infrastructure of only a single worker, the reducer did not recieve all the subresults it needed. When spreading the same number of subtasks across multiple workers, the scheduler recieved all subresult succesfully, even though it probably performed more concurrent processing than when it only recieved the output of the single worker. Thus, it appears that the bug is in the worker implementation and must be subject to further investigation.

5.4 PERFORMANCE BENEFITS BY INCREASING NUMBER OF REDUCERS

The goal of the framework is to speed up solving of problems by parallelization. While the main focus of the thesis is parallelization through horizontal scaling of worker nodes, it makes equal sense to use more reducers. A single reducer may be under heavy load if many workers provide subresults for them to process, both in handling all the incoming connections and the associated data, but also in applying the reduce functions . It is easy to see that the reducer can become a bottleneck in performance. To evaluate this, multiple jobs must be input to the framework as the benefits of multiple reducers cannot be reaped by solving a single task, as there is no reducer parallelization; the task is purely processed by a single reducer. To evaluate the effects of multiple reducers on performance, the framework was evaluated with 10 available workers, 3 input tasks (the hard task described above, input repeatedly as new tasks) and a variable number of reducers (1, 2 and 3).

Surprisingly, using a single reducer was consistently superior to using a pair of reducers, while using three reducers has a similar performance to using just one. Thus, these observations are inconclusive and more measurements must be performed.

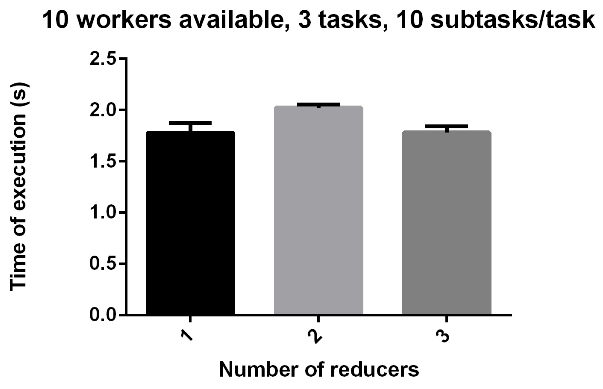

Figure 6: Increasing number of reducers when processing multiple tasks

6 CONCLUSION

Distributed parallel computing is a popular approach to solving scientific questions and examination large data sets. Due to the wide variety of possible jobs that can be input to such a system, tweaking performance is not simple and the preexisting frameworks highlight many aspects of performance that must be adressed. In this thesis I have provided my implementation of a framework for distributed parallel computation based on a MapReduce-like approach, where tasks are processed in three steps. The framework uses a unique approach to evaluation performance of nodes that should be evaluated as part of various scheduling strategies in future studies. The framework provides a novel fault-tolerance model where data is replicated across nodes until it can be safely deleted. Experiments have been made to evaluate the performance of the framework. Evaluation of the framework proved disapointing, as parallelization of tasks did not improve performance on task solving. Parallelization was increased both by increasing the number of subtasks for a given task, such that more workers can work on a task and by increasing the number of available workers. However, the fastest computation time was archieved when just a single worker executed the task, independent of the number of subtasks, indicating that there is a large overhead associated with the scheduling mode. Parallelization in regard to reducers was evaluated, as more reducers reduce the load of a single reducer. The results were inconclusive, as using two reducers was associated with slower task completion than using one, while three reducers had a comparable execution time. Time did not allow for evaluating the performance of various scheduling strategies or the fault tolerance mechanisms. The measurements indicate that there are flaws in the implementation that must be adressed as the framework does not meet its design goals satisfyingly.

BIBLIOGRAPHY

[1] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System.” In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 29–43. ISSN: 0163-5980. DOI: 10.1145/1165389.945450. URL: http://doi.acm.org/10.1145/1165389.945450.

DECLARATION

Put your declaration here.

Aarhus, January 2017

Jens Christian Bredahl Madsen