Bio-inspired optimization techniques using Apache Hadoop and Oozie

master thesis in computer science

by

Christian Gapp

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements
for the degree of Master of Science

supervisor: Dr. Juan José Durillo, Institute of Computer Science

Innsbruck, 18 November 2014

Certificate of authorship/originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Christian Gapp, Innsbruck on the 18 November 2014

Abstract

Problem optimization is a fundamental task we encounter everywhere, from everyday life to the most complex science areas. Finding the optimal solution often takes an unreasonable amount of time or computing resources, therefore approximation techniques are used to find near-optimal solutions. Bio-inspired algorithms provide such approximation techniques, they are based on existing solutions found in the nature. But even those techniques are sometimes to slow for extensive problems, so they need to be run in parallel.

In this master-thesis, implementations of some bio-inspired optimization techniques are provided, that can be run on an Apache Hadoop cluster, by using the capabilities of YARN. The runtimes of those algorithms are then compared to their sequential version. Finally, the implementations are made usable by Apache Oozie, which is a Hadoop workflow scheduler that uses XML for its workflow configuration. This way, those optimization techniques are made accessible to a broader range of users.

Contents

Chapter 1.

Introduction

The parallelization of algorithms can be a hard task, where knowledge and correct application of the available tools is a crucial factor for success. Beside the fact, that not every programmer has that kind of knowledge, it is an error prone process. So it is useful, if the low level parts of parallelization are hidden behind an API.

There are several approaches in parallelization. All of them boil down to distribute the computational work to different resources like CPU or GPU. The resources can reside on a local system, but also on remote computers. On local resources, processes and threads are common abstractions for parallel programs. They are sometimes hidden by higher level API's like OpenMP [1][2].

For the distribution and computation on remote systems, other solutions exist. One of the most popular is MPI [3], that uses message passing for the communication between its computational entities. Alternatives, like Global Arrays [4] and their successors, the partitioned global address space languages [5], try to provide a global address space over a distributed system. All those approaches have their advantages, like good portability and high performance, but they are often applied only in educational institutes or by the scientific community, because they have one main use case: high performance computation (HPC). HPC is for example used in simulations and machine learning. Only a few institutes and companies are interested to spend money for the hard- and software just for this purpose.

In contrast to the specialized solutions mentioned above, Apache Hadoop [6] provides an open source solution that continues to gain support in education, science and business. It has no special hardware needs and can run on a cluster of commodity hardware.

In the beginning, Hadoop was restricted to the MapReduce [7] programming model, that is well suited for parallel data analysis on a big scale. In this sense, it is often compared to database systems [8][9][10] or combined with them [11][12]. Other projects extend MapReduce to provide SQL-like capabilities. Pig [13] is a high-level dataflow system and provides an SQL-style language (called Pig

Latin), that is compiled to MapReduce jobs. Hive [14] is a data warehousing solution, that supports queries expressed in HiveQL, also a language similar to SQL. The analytical capabilities make MapReduce on Hadoop an interesting choice for many companies.

But MapReduce usually performs poor, when it is applied to problems that don't fit into its scheme [15][16][17][18]. An outstanding example for such problems is the category of iterative algorithms, Bio-inspired optimization techniques belong to them. There are approaches to expand MapReduces capabilities, like HaLoop [16], Pregrel [19] and Twister [20]. All of them try to use MapReduce for purposes it wasn't designed for - with varying degrees of success.

Since the upcoming of YARN [21], the next generation of Hadoop's compute platform, the programming model for Hadoop has moved its focus from the analytical model, to a general purpose execution style. YARN is expressive enough to cover MapReduce, which is still an important part of Hadoop. But it can also be adopted for other computational styles, like the iterative one.

YARN is relatively new, the first general available and supported release was for Apache Hadoop 2.2 at October 2013 [22]. Because of that, just a few frameworks support parallel iterative applications on YARN. Apache Spark [23][24] uses resilient distributed datasets (RDDs) [24]. RDDs are basically immutable collections, on which defined operations can be performed (e.g. map, filter and join). Because RDDs are stored in memory, it is really fast to work with them. But as the number of available operations is limited, Spark is not suitable for all kinds of computations. Spark is the foundation for other projects like Apache Mahout [25] (beginning with version 1.0), which is a machine learning library.

Another framework, that has gained a lot of attention, but doesn't use YARN as its resource manager, is Apache Storm [26]. Storm is typically used to process unbounded streams of data, although it can be used for a broad range of workloads. The streams are send through a configurable graph, where each node does some processing on the data. Apache Tez [27] and Apache Samza [28] are similar to Storm, but run on YARN.

Biohadoop, the program developed during this master thesis, is an alternative to the solutions presented above. It is a framework, that runs on top of YARN and is specifically designed to run iterative algorithms, for example Bio-inspired optimization algorithms. This is achieved by using a master - worker approach, where computationally intense work is distributed from the master to the workers. The data transmission between the master and its workers is configurable, so it can fit to the problem and environment. In addition, by providing a simple asynchronous API, the details of algorithm parallelization are hidden from the user.

Compared to the other solutions, Biohadoop is also capable to distribute it's tasks to external workers, that are not under the control of Hadoop, like web browsers or mobile phones. Through this mechanism, it is possible to harness additional computational capacities. Possible usage scenarios are distributed computing projects, like Folding@home [29] or SETI@home [30].

In addition to Biohadoop, an extension to Apache Oozie [31][32] was implemented. Oozie is a workflow management system for Hadoop. Through this extension, Oozie is able to include Biohadoop in its workflows. The workflows are useful, if several steps are needed to complete a bigger task, for example the simulation of a protein and analysis of the results afterwards.

The rest of the document is organized as follows: in chapter 2, an introduction to Bio-inspired algorithms is given, as well as an overview of common representatives. The Bio-inspired algorithms are used as examples of a class of problems, that can be efficiently solved in a distributed manner by Biohadoop. Chapter 3 provides the information about Apache Hadoop and Oozie, that is needed to understand the functionality of Biohadoop. Chapter 4 then introduces Biohadoop's architecture and its components, and shows the implemented modifications for Oozie in section **??**. Chapter 5 shows, how Biohadoop's API can be used to implement new algorithms. It also demonstrates the steps necessary to execute Biohadoop. Chapter 6 shows and compares the performance of Biohadoop, using its various communication facilities. The conclusions in chapter 7 summarize the master thesis and the results obtained. The appendix contains additional material, that completes the information given in the prior chapters. It is also there, where information about a pre-build environment for Hadoop and Oozie can be found, together with a quickstart guide on how to run Biohadoop on this environment.

Chapter 2.

Bio-inspired optimization techniques

Optimization is the task of finding a solution to a problem, that is better, or even the best compared to other solutions. A common optimization example is the traveling salesman problem (TSP) [33]. In TSP, a salesman needs to visit a bunch of cities, that are connected to each other through paths of varying length. The goal is to find the shortest tour, such that the salesman visits each city exactly once and, at the end, returns to the city where he started the travel.

2.1. Single objective optimization

Optimization is generally done according to a defined goal, also called objective. In the TSP example, it's the objective to find the shortest path for a complete tour. If there is just one objective, the problem is called single objective optimization problem (SOP).
Definition (SOP): a SOP is defined by the pair $P = (S, f)$, where

- S is the set of possible solutions, also called solution space
- $f: \ S \mapsto \mathbb{R}$ is the objective function, that we want to minimize or maximize

The method for finding the global optimum, is called global optimization. The global minimum optimization for the problem stated above is given in formula (2.1).

$$s' \in S \mid (f(s') \leq f(s) \text{ for all } s \in S) \tag{2.1}$$

That means, that we want to find a solution, that is better than, or at least as good, as the other solutions.

2.2. Multi objective optimization

We talk about a multi objective optimization problem (MOP), if we have several objectives that we want to optimize at the same time.

Definition (MOP): a MOP is defined as finding a vector $\mathbf{x} = [x_1, ..., x_n] \in \Omega$, which satisfies the m inequality constraints $g_i(\mathbf{x}) \geq 0, i = 1, ..., m$, the p equality constrains $h_i(\mathbf{x}) = 0, i = 1, ..., p$ and minimizes (maximizes) the components of the vector function $\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), ..., f_k(\mathbf{x}))$, where $f_1, ..., f_k$ are the k objective functions. It is noted that $g_i(\mathbf{x}) \geq 0$ and $h_i(\mathbf{x}) = 0$ represent constraints that must be fulfilled while minimizing (or maximizing) $\mathbf{F}(\mathbf{x})$. The universe Ω contains all possible $\mathbf{x}$ that can be used to satisfy an evaluation of $\mathbf{F}(\mathbf{x})$.

Reusing the TSP example, our two objectives would now be to find a) the shortest path, that b) costs as little as possible. The objectives are usually in conflict with each other, otherwise they could be combined. To elaborate on the TSP example, this could mean that the shortest path includes driving on the highway (causing higher costs due to toll fee), the cheapest path would be to drive on a normal street (longer distance). So we have to find a compromise between the goals which means that there isn't a single best solution, but a set of solutions. Some solutions may result in a shorter path, where other ones may result in lower costs. The task of MOP is to find a set of solutions, from which a decision maker (usually a human) selects the final solution.

It's not obvious how to compare two solutions in MOP. Figure 2.1 gives three examples of a problem with four objectives, in each example the solutions A and B are compared, the task is to minimize a problem. In picture (a), solution A is better than solution B, as all values of A are smaller than their respective values in A. In picture (b), B is better than A for the same reason. The situation in picture (c) is more complicated and doesn't give a clear answer to the problem, as some values in A are smaller than their respective values in B and vice versa. One could now argue, that solution A is better than solution B, because there are more elements in A that are smaller with respect to their elements in B. But this does not hold true, as the number of smaller values does not say anything about the optimality of the solution. Considering the TSP example, what is a better solution, the distance or the gas consumption? This can not be answered in general, therefor, to find a set of solutions for a MOP, the Pareto Optimality theory is used [34].

The Pareto Optimality theory defines the concept of Pareto Dominance, that can be used to compare two solutions. Figure 2.2 gives an example of Pareto Dominance.

Definition (Pareto Dominance): a vector $\mathbf{u} = (u_1, ..., u_k)$ is said to dominate a vector $\mathbf{v} = (v_1, ..., v_k)$ (denoted by $\mathbf{u} \preceq \mathbf{v}$), if and only if $\mathbf{u}$ is partially less

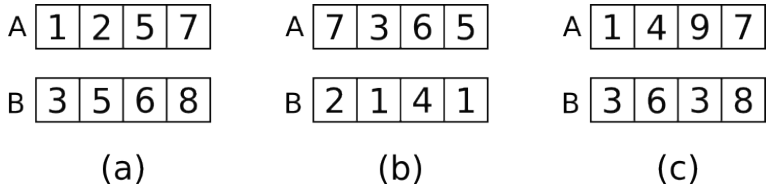

Figure 2.1.: (a) a is better than b, (b) b is better than a, (c) none is better - a and b are non-dominated

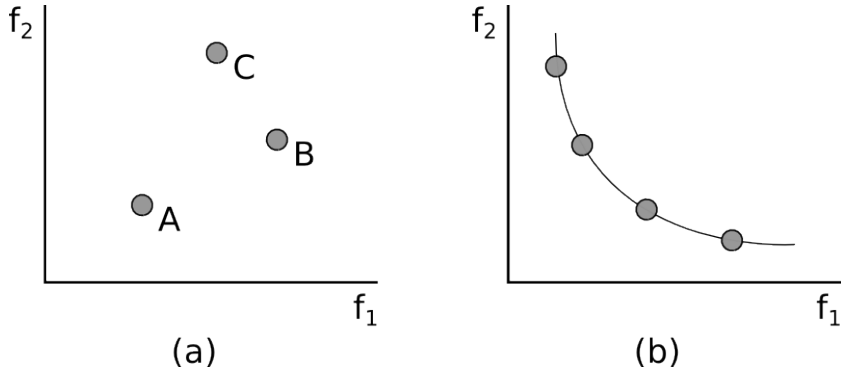

Figure 2.2.: (a) A dominates B and C, (b) all points are non-dominated

than $\mathbf{v}$, i.e., $\forall_i \in \{1, ..., k\}, u_i \leq v_i \wedge \exists i \in \{1, ..., k\} : u_i < v_i$.

In figure 2.2, picture (a), we see that A dominates the solutions B and C, because

- the values for f_1 and f_2 of A are the same or smaller than the corresponding values for B respectively C
- at least one of the values for A is smaller than the corresponding values for B respectively C

In picture (b) we see that no solution dominates another solution.

Using the concept of dominance, it is possible to define when a solution is optimal, this is known as Pareto Optimality.

Definition (Pareto Optimality): a solution $\mathbf{x}$ is Pareto Optimal, if there is no $\mathbf{x}' \in \Omega$ for which $\mathbf{v} = \mathbf{F}(\mathbf{x}') = (f_1(\mathbf{x}'), ..., f_k(\mathbf{x}'))$ dominates $\mathbf{u} = \mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), ..., f_k(\mathbf{x}))$.

This means, that no objective of a Pareto Optimal solution $\mathbf{x}$ can be improved, without negatively affecting at least one of it's other objectives.

The solution to a MOP is then the set of non-dominated solutions, also called the Pareto Optimal Set.

Definition (Pareto Optimal Set): for a given MOP $\mathbf{F}(\mathbf{x})$, the Pareto Optimal Set is defined as $\mathcal{P}^* = \{\mathbf{x} \in \Omega | \neg\exists \mathbf{x}' \in \Omega, \mathbf{F}(\mathbf{x}') \preceq \mathbf{F}(\mathbf{x})\}$

Its correspondence in the objective space (that is, the space where the results of the the objective functions lay) is called the Pareto Optimal Front, or just Pareto Front.

Definition (Pareto Front): for a given MOP $\mathbf{F}(\mathbf{x})$ and Pareto Optimal Set $\mathcal{P}^*$, the Pareto Front $\mathcal{PF}^*$ is defined as $\mathcal{PF}^* = \{\mathbf{F}(\mathbf{x}) | \mathbf{x} \in \Omega\}$

When searching for the solutions of a MOP, the goal is to find a Pareto Front that:

- has good convergence to the optimal Pareto Front, i.e. it is as near to an optimal solution as possible
- has good diversity, i.e. the solutions are well distributed throughout the Pareto Front

Figure 2.3 gives an example for convergence and diversity of the Pareto Front. In picture (a) we have a Pareto Front with a bad convergence, as it is far away from the optimal/true Pareto Front. Picture (b) shows a Pareto Front that has bad divergence, i.e. several sections of the optimal Pareto Front are not covered with solutions. Picture (c) shows the ideal case, where the Pareto Front matches with the optimal Pareto Front - this is the desired solution.

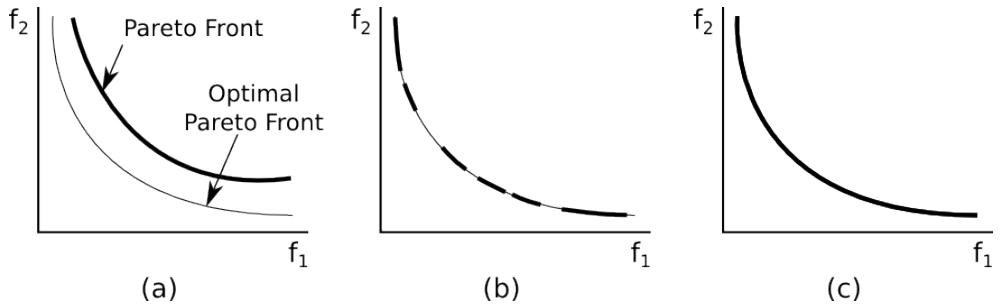

Figure 2.3.: (a) Pareto Front with bad convergence, (b) Pareto Front with bad divergence, (c) ideal case, where the Pareto Front matches the optimal Pareto Front

2.3. Complexity considerations

Optimization is usually a computationally intensive task. For the TSP example, we can compute how many different solutions exist for a problem with n cities, using formula (2.2). Already a small number of cities entails a large number of

solutions. For example, if we have 15 cities, we have over 43 billion solutions, that we need to evaluate to get the best solution. This number grows quickly if new cities are added and soon it becomes impossible to compute the optimal solution.

$$(n-1)!/2, \text{ where } n \text{ is the number of cities} \tag{2.2}$$

Often there is no need to find the best solution to a problem, instead it is sufficient to find a good enough solution in reasonable time. Approximation techniques can be used for this purpose. They don't guarantee to find the exact optimal solution, as they don't search through the entire solution space, but have the advantage, that they deliver solutions in a fast way. The solutions are usually near-optimal, although they can also be arbitrarily bad.

One well known family of approximation techniques are the metaheuristics [35]. A metaheuristic defines an abstract sequence of steps that leads to the optimization of a problem. As such, they are not problem specific and can be applied to a broad range of optimization problems.

2.4. Optimization, inspired by nature

Bio-inspired optimization techniques (BIO) are a sub family of the metaheuristics. The name derives from the fact, that they mimic behaviors observed in nature. For example, genetic algorithms (GA) [36] imitate the concept of evolution, where only the fittest individuals survive and reproduce. Another example is particle swarm optimization (PSO) [37], that mimics the behavior of a flock of birds. In ant colony optimization (ACO) [38], as the name already says, a digital colony of ants is used for optimization.

Bio-inspired optimization techniques are used today in many areas, like mechanical and electrical engineering, image processing, machine learning, network optimization, data mining etc. [36].

2.4.1. Genetic algorithm

Genetic algorithms (GA) are a population-based optimization strategy. The population consists of n individuals, each one representing a solution of the optimization problem. Every individual gets assigned a fitness value, that is computed according to the optimization objective.

The idea behind genetic algorithms (GA) is to iteratively evolve the population towards an optimal solution, by applying selection, recombination (crossover) and mutation to it.

The recombination is performed by selecting two or more individuals out of the whole population. Those individuals are called the parent individuals. Note that fitter individuals are preferred for the recombination, as they have a high chance to produce fit offsprings. The parents are recombined using some crossover operator. An example of a crossover operator is the computation of the mean value between the parents. The crossover results in one ore several child individuals. A mutation operator is then applied to the children, resulting in slightly mutated child individuals. Typically, in each iteration, a population of size n generates n child individuals, but there are also other strategies, e.g. a steady state reproduction strategy [39].

At the end of each iteration, the fitness of all individuals (parents and children) is computed and the n fittest individuals are selected to form the base population of the next iteration. This process is known as natural selection or survival of the fittest and leads intuitively towards fitter and better results.

Algorithm 1 shows the pseudo code for a GA.

Algorithm 1 Genetic algorithm

```
1: procedure GA
2:     P ← generateInitialPopulation
3:     evaluate(P)
4:     while !terminationCriteria do
5:         P' ← recombine(P)
6:         P" ← mutate(P')
7:         evaluate(P")
8:         P ← select(P, P")
       return best solution found so far
```

The GA algorithm can be used to solve SOP and MOP. In the case of MOP, the GA must be modified in order to produce a set of solutions, that form a Pareto Front. A typical implementation of such a modification is NSGA-II [40]. In NSGA-II, the selection is performed based on ranking and crowding distance.

The ranks of the individuals are computed by iteratively finding the non-dominated solutions in the set of yet not ranked individuals, and assigning the current rank to them. The rank starts at 0, after each iteration, it increases by one. The iteration stops after all individuals are ranked.

If two individuals have the same rank, the crowding distance is used to find the better result. The crowding distance measures, how distant neighboring solutions are to a given individual. Higher crowding distances are preferred, as this leads to better diversity in the final solution.

As an example, look at figure 2.4. In picture (a) we have a population of three individuals, namely A, B and C. As one can see from the picture, A is

non-dominated, but dominates B and C. So, if we want to apply the ranking algorithm, rank 0 is assigned to A (because A is non-dominated). Then, the rank is increased by one. Now we have two individuals that are yet not ranked, B and C. Those individuals are again non-dominated, because we don't consider A anymore, which is already ranked. The rank of 1 is applied to A and B. After this iteration, the ranking algorithm stops, because all individuals have a rank.

Picture (b) of figure 2.4 gives a graphical representation of crowding distance, that is used to select the better solution if two individuals have the same rank.

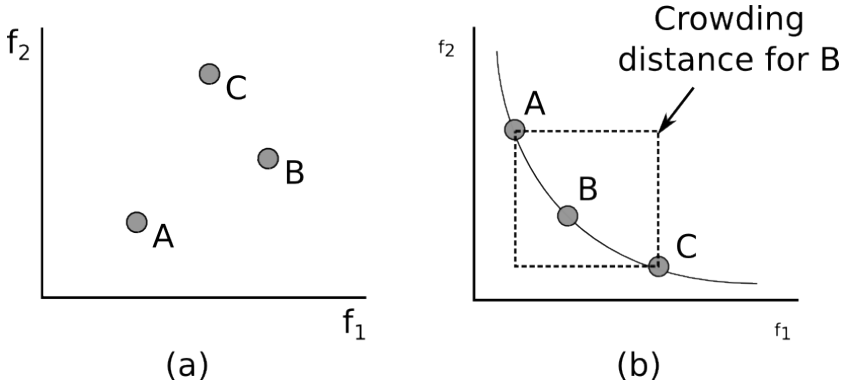

Figure 2.4.: (a) A has rank 0, B and C rank 1, (b) crowding distance of solution B

2.4.2. Particle swarm optimization

The particle swarm optimization algorithm (PSO) mimics the behavior of a flock of birds. The birds in a flock usually follow a highlighted bird, for example the first one. If the highlighted bird changes its direction, the other birds will also adjust their direction. This principle can be used for optimization.

In PSO, the population consists of a number of particles (birds). Each particle has a position, a velocity and a fitness value. Each particle has knowledge of the global best solution found so far (gBest), and its personal best solution (pBest), found so far. What a particle now does, is to move in the direction of gBest. For this, in each iteration the particle adjusts its velocity according to its current velocity and the distance to gBest and pBest. Then it moves according to the adjusted velocity. This simple behavior lets the particle converge towards gBest.

The PSO can be implemented using a simple vector operation, given in formula (2.3).

$$v(t) = w * v(t-1) + c_1 * r_1 * (pBest - x(t-1)) + c_2 * r_2 * (gBest - x(t-1)) \quad (2.3)$$

This vector operation is performed for each iteration on all particles. $v(t)$ is the velocity of the particle at time t, $v(t-1)$ is the velocity in the previous iteration and $x(t-1)$ defines the position of the particle, also in the previous iteration. w is called the inertia weight, that defines the influence of the current speed on the new velocity. The parameter c_1 is called the cognitive acceleration coefficient and defines how much the particle is influenced by its personal best solution. c_2 is called the social acceleration coefficient and defines how much the particle is influenced by the global best solution. The parameters r_1 and r_2 are random values between 0 and 1 and are used as a source of diversity.

After the particles velocity is computed, its current position is updated using formula (2.4)

$$x(t) = v(t) + x(t-1) \tag{2.4}$$

An example of four iterations for a single particle can be found in figure 2.5. Picture (a) shows the initial situation, the red dot highlights gBest, the gray dot the particle. Picture (b) shows how the current velocity and position of the particle, as well as the position of gBest, influence the velocity and position of the particle in the next iteration, which is shown in picture (c). Pictures (d) to (i) show additional iterations, all conforming to the same principle. The value of pBest is not considered in this example.

2.4.3. Ant colony optimization

The ant colony optimization (ACO) is an optimization technique for combinatorial problems, like the TSP. It is inspired by ant colonies, that are very effective in finding shortest paths from their nest to a source of food.

To find this paths, ants deposit a pheromone on the track they are using. The pheromone evaporates over time, such that paths that are frequently used (for example because they are shorter), have more pheromones applied, than seldom used paths. If an ant must decide which path to take, it follows with a higher probability the path that has the most pheromones applied. While using this path, it applies again pheromones, which increases the probability that other ants will follow this path.

Individually, the ants take only simple decisions, based on the amount of pheromones on a given path. Collectively, they work on solving an optimization problem.

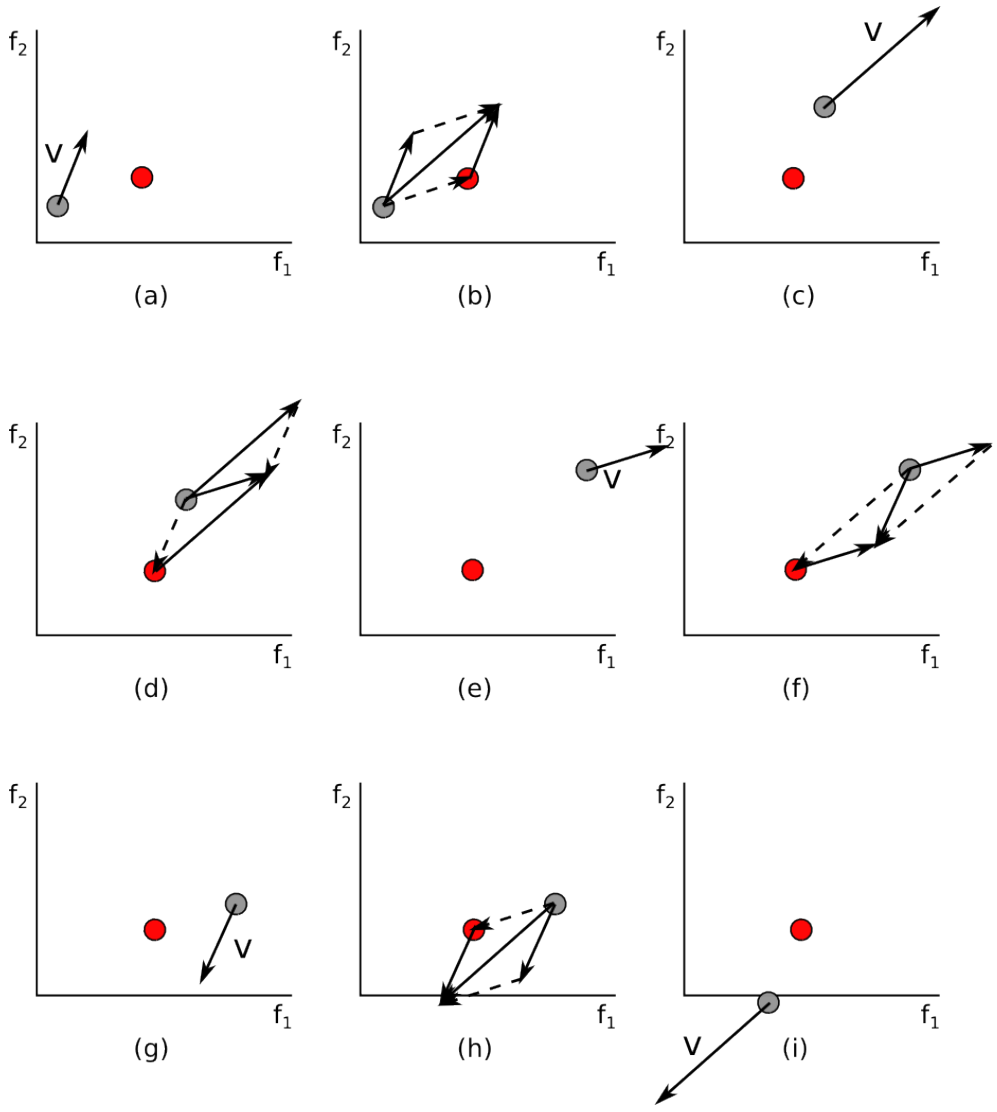

Figure 2.5.: Four iterations of PSO for a single particle. The red dot marks gBest, the gray dot the particle. The velocity and position of the particle does rely only on gBest in this example, pBest is not used

Chapter 3.

Hadoop

Apache Hadoop [6] is an open source software project for massive data processing and parallel computations in a cluster. It derives from Google's MapReduce [7] and Google File System (GFS) papers [41], but has undergone quite some changes due to its active development, for example the introduction of YARN.

Hadoop is designed to run on inexpensive commodity hardware and provides a high degree of fault tolerance, implemented in software. It scales from a single server up to thousands of machines. Each machine stores data and is used for computation.

Following are the key properties of Hadoop:

- Scalability: Hadoop is able to scale horizontally. New nodes can be added at runtime if there is demand, unnecessary nodes can be shut down.
- Cost effectiveness: As Hadoop runs on commodity hardware, there is no need for exclusive, proprietary hardware, which can be a huge cost saving factor. It is possible to run Hadoop on an already existing infrastructure. Hadoop runs also in the cloud, different offerings exist from Amazon, Google, Cloudera, etc. This can further reduce costs, for example, when big data capabilities are seldom needed.
- Flexibility: Hadoop can handle any kind of data. Custom applications running on Hadoop enable the transformation of, and computation on arbitrary data.
- Fault tolerance: Hadoop expects hardware failures, it is designed from the ground up with fault tolerance in mind. If a machine fails, Hadoop automatically redirects the computations and data of this machine to another machine.

Hadoop consists of two main components, HDFS (Hadoop Distributed File System) and YARN (Yet Another Resource Manager). HDFS is a scalable and reliable file system. YARN assigns resources (CPU, memory, and storage) to

applications running on a Hadoop cluster and is part of Hadoop since version 2.0. It replaced the resource management capabilities, that were bundled in MapReduce prior to Hadoop version 2.0. Since this version, MapReduce uses YARN for the resource management. YARN on the other hand offers the possibility to host computational models that are different from MapReduce, like Biohadoop, which wasn't possible before.

Figure 3.1 shows the architecture of Hadoop. HDFS provides data services as the base layer, YARN builds on it and manages the resources of the cluster. The different applications, like MapReduce, Storm, Spark, Biohadoop etc. run on top of YARN.

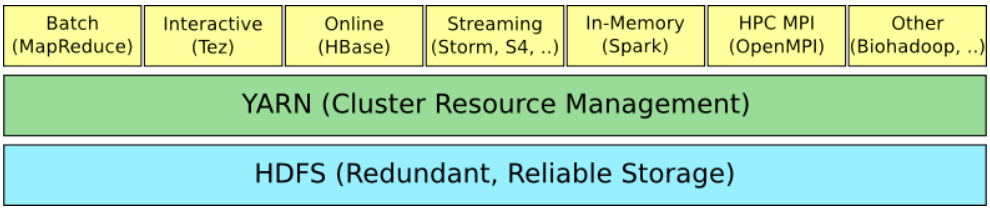

Figure 3.1.: Hadoop layers: HDFS forms the base and provides data services, YARN builds on it and provides resource management. The applications run on top of YARN.

3.1. HDFS

HDFS is the distributed, scalable and reliable file system of Hadoop, written in Java. A HDFS cluster consists of a single NameNode and several DataNodes. The NameNode manages the file system namespace, regulates the client access to files and commands the DataNodes. A DataNode stores the assigned data on the storage that is attached to its cluster node (usually there is one DataNode running on each cluster node). HDFS performs file transfer and storage only between clients and DataNodes or among DataNodes. The NameNode never comes directly in touch with the user data. This way, HDFS can scale by adding additional DataNodes.

The files in HDFS are stored in blocks of a configurable maximum size (default 128MB). Reliability arises from the replication of blocks to other available DataNodes. The number of replicas is configurable and has a default value of 3, which means that each block is stored three times in HDFS. If a node goes down, for example because of a hardware failure, HDFS automatically replicates this node's data blocks to other nodes by using the remaining copies, such that the replication factor is satisfied again. Files that are bigger then the maximum

block size are split into several smaller blocks. The resulting blocks are handled as described above.

The restriction of a single NameNode instance makes the NameNode effectively a single point of failure, but there exist solutions for high availability [42][43].

Figure 3.2 shows how HDFS organizes the data. In this example, each file is replicated twice (replication factor of 2). Note how HDFS tries to store block replicas on different nodes. For the sake of simplicity, all files in this example are smaller than the HDFS maximum block size, thus fitting in a single block.

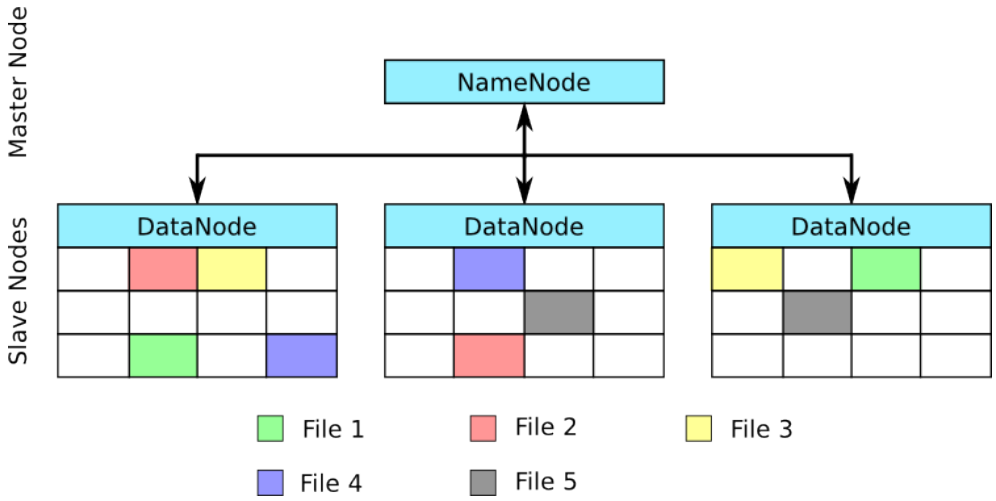

Figure 3.2.: HDFS file storage, each file is replicated twice.

HDFS provides rack awareness, which allows applications to consider the physical location of a machine when data has to be stored or moved. Rack awareness allows a variety of advanced features. For example, in the case of computations it is best to move the computation to the nodes that already contain the necessary data, instead of moving the data to the computation. This minimizes possibly slow data traffic. Another example is HDFS itself, that uses its rack awareness for its performant replication process, while considering the physical location of the replicas.

3.2. YARN

YARN is the resource manager of Hadoop. It schedules and satisfies resource requests of applications, and monitors running applications. Resources are granted in form of resource Containers, that consist of CPU, RAM, storage etc.

The functionality of YARN is provided by one ResourceManager and several NodeManagers (similar to HDFS). This architecture offers the needed scalability. Like in HDFS, the ResourceManager is a single point of failure, but solutions for high availability exist here, too [44].

The ResourceManager has two components, the Scheduler and the ApplicationsManager. The Scheduler allocates resources to running applications, but performs no monitoring of running applications. This is the job of the ApplicationsManager. The ApplicationsManager is responsible for accepting new application submissions, negotiating the first Container of an application and monitoring of running applications. The monitoring aspect allows the ApplicationsManager to automatically restart failed applications.

The NodeManager (usually one per node) is responsible for the containers, that run on its node. It monitors their resource usage and reports this information back to the ResourceManager.

A typical YARN application consists of three components:

- Client: this is the starting point of every YARN application. It submits the application to the ResourceManager, which allocates a free Container in the cluster and starts the ApplicationMaster inside this container.

- ApplicationMaster: the ApplicationMaster (don't confuse with ApplicationsManager) communicates with the ResourceManager. The ApplicationMaster can request additional Containers, for example if it wants to distribute some of its work. It can return already allocated containers, if they are not needed. And it provides information about its current status in form of a heart beat. The heartbeat is used by the ApplicationsManager to determine, if the application is still alive, or needs to be restarted.

- Additional Containers: the additional Containers are not mandatory, but can be useful, as they provide additional resources. The Containers can be used to offload work to them, for example in a Master - Worker scheme, like implemented in Biohadoop. The Master runs on the ApplicationMaster and starts several Containers. Each Container runs a Worker.

There is no defined way for data exchange between an ApplicationMaster and its additional Containers. If data exchange is a requirement, it must be implemented separately. To see how this is done in Biohadoop, please take a look at chapter **??**.

The automatic restart capabilities of YARN are not extended to additional Containers, as there is no general valid solution for their restart. For example, some containers need to maintain state, which must be reflected on a restart.

Other containers may work in a stateless fashion. What YARN does, is to provide information about the states of its additional Containers to the ApplicationMaster. This way, the ApplicationMaster can implement the restart of failed Containers on its own.

YARN runs applications at the same time, as long as there are enough cluster resources available. Figure 3.3 shows an example for the occupation of a Hadoop cluster with one master node and three slave nodes. The NameNode (HDFS) and ResourceManager (YARN) run on the master node, the DataNodes (HDFS) and NodeManagers (YARN) run on the slave nodes and communicate with the corresponding services on the master node. The applications are started by the clients, that request an initial Container for their ApplicationMaster from the ResourceManager. After the ApplicationMasters are started, they in turn request additional Containers from the ResourceManager. In this example, two applications are started, where Application 1 occupies five Containers and Application 2 occupies seven Containers.

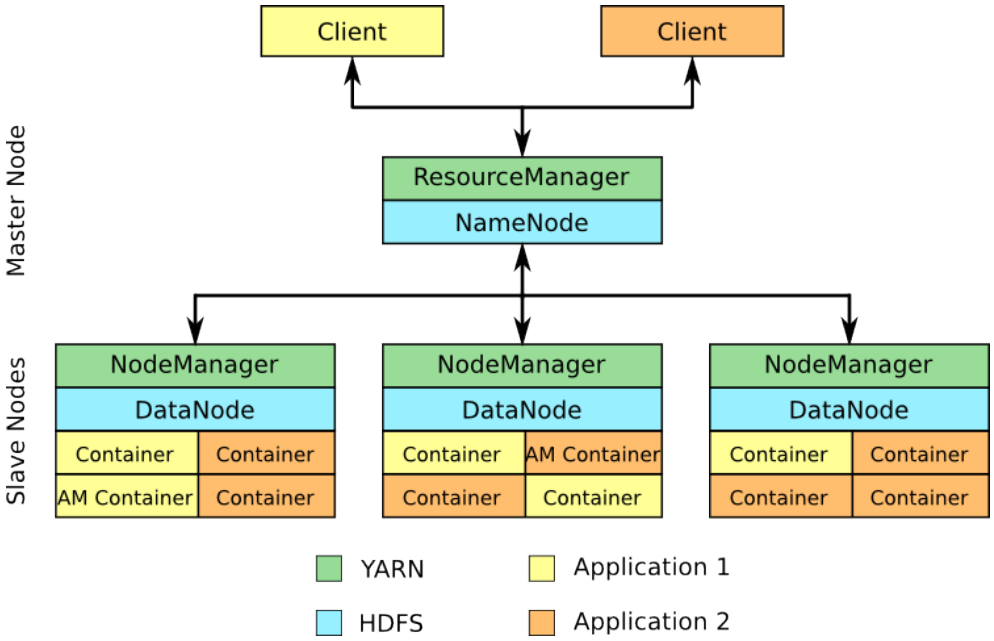

Figure 3.3.: Example occupation of a Hadoop cluster for two applications. The NameNode (HDFS) and ResourceManager (YARN) run on the master node. DataNodes (HDFS) and NodeManagers (YARN) run on the slave nodes and communicate with the corresponding services on the master node. The applications, including the ApplicationMasters, run on the slave nodes inside containers.

3.3. Oozie

Apache Oozie is a workflow tool, that runs on Hadoop. The workflow jobs are Directed Acyclic Graphs (DAGs) of actions, written in XML. Each action has two possible outcomes: "ok" if the action completed without an error, and "error" in the case of an error. The action outcome influences the next transition in the DAG.

An Oozie workflow consists of two types of nodes, the control flow nodes and the action nodes. Control flow nodes define the sequence of actions in a workflow (start, end, failure, decision, fork, join). Action nodes are used to execute a program or computation.

Oozie provides a number of default actions, like the Java action, that can be used to start a YARN application. Other actions include MapReduce actions, HDFS file system actions (move, delete and mkdir), Email, etc. Oozie allows also to implement new custom actions, that expose different behaviors. In the course of the work on this master thesis, a custom action was implemented to invoke Biohadoop from Oozie. More details about this custom action and its configuration can be found in chapter **??**.

Figure 3.4 shows a workflow example. It starts at the control flow node labeled "Start". Then it executes the "FS action". The transition to the next step depends on the outcome of this action. If an error happened, the next (and final) action is the "Failed" action. If no error happened, the "Fork" action will be invoked, that starts two Java actions in parallel. Actions performed inside a "Fork" action have a special error semantic. If an error happens in any of the parallel actions inside a fork node, all of the parallel action are considered as failed. If no error happened, the "Join" node waits for the two actions to finish. Then, a "Decision" action is invoked, that transitions to the "MapReduce" Action if the needed conditions are given. If the whole workflow had no errors, it terminates at the "End" node.

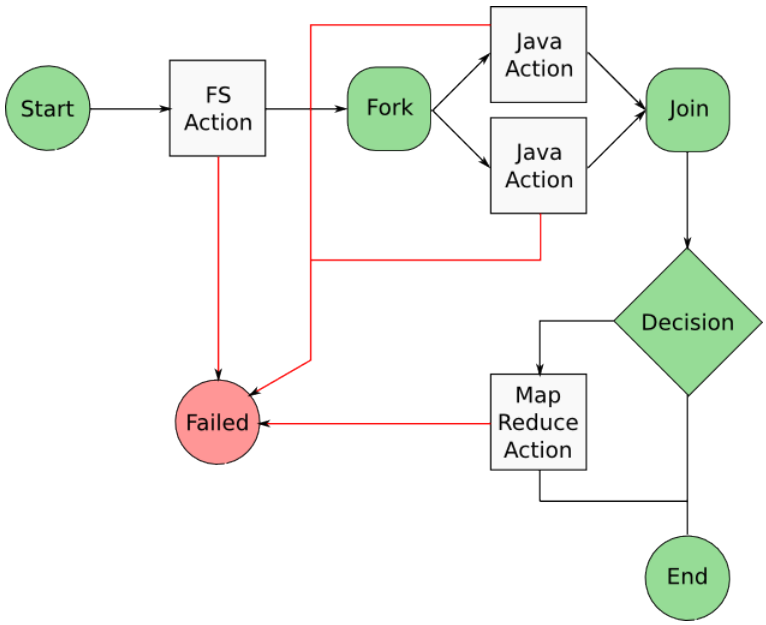

Figure 3.4.: Oozie workflow transitions. If any action returns with an error, the workflow transitions to the final state "Failed", else the workflow advances according to the defined sequence.

Chapter 4.

Implementation

4.1. System architecture

Biohadoop is a framework for running parallel algorithms on Apache Hadoop. It works according to the master - worker principle, where one master commands several workers. The master typically implements an algorithm, whose compute intense parts are executed by the workers in parallel.

The framework relies on YARN (chapter 3.2) and provides features, that otherwise need to be implemented manually, such as:

1. support for running several algorithms in a single Biohadoop instance. This has the advantage that workers, and therefor resources, can be shared by the algorithms

2. asynchronous communication between master and workers using a simple task system (section **??**)

3. storage and load of arbitrary data to a file system (section **??**)

4. support for the island model, a high level parallelization that can be used to improve the optimization performance of bio-inspired optimization techniques (section **??**)

5. support by Apache Oozie through a custom action (section **??**). This custom action can be used to start Biohadoop as a single instance, or to start several instances in parallel

Every YARN application needs a client that submits the application to YARN. In Biohadoop, the client is implemented in the class `BiohadoopClient`. It is the main entrance point to run Biohadoop in a Hadoop environment and responsible to start the ApplicationMaster under the control of Hadoop.

The ApplicationMaster's main class is `BiohadoopApplicationMaster`. In a local (development) environment, it acts as the main entrance point to run

Biohadoop (more on how to run Biohadoop can be found in 5.2). The ApplicationMaster is responsible for the start of the configured workers in additional YARN containers, and as it is the master in the master - worker scheme, it executes the configured algorithms and communicates with the workers.

The communication between the master and its workers is hidden from the algorithm (and therefor the user) by the task system. It provides a simple interface through the `TaskSubmitter` class. This interface can be used to submit work items, the so called tasks, to the task system, along with details of how the computation should be performed. The tasks are stored in a queue, the `TaskQueue`, until their result is known. Endpoints take the tasks and their configurations out of the queue and send them to waiting workers. The workers execute the configured computations on the tasks and return the results to the endpoints, that promote the results back to the `TaskQueue` and from there to the algorithms. Figure 4.1 gives a graphical representation of this concepts, more details about the task system can be found in section **??**

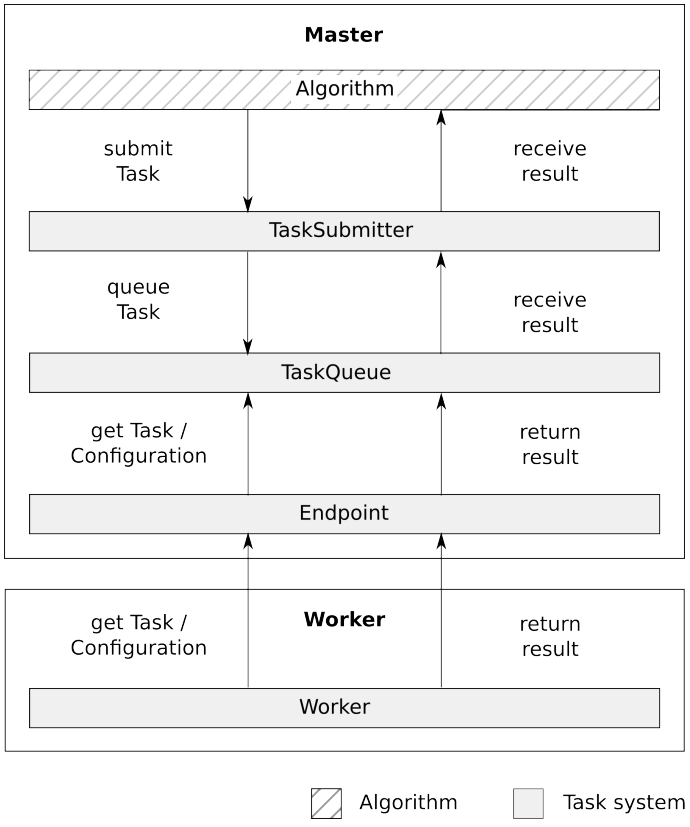

Figure 4.1.: Biohadoop system architecture

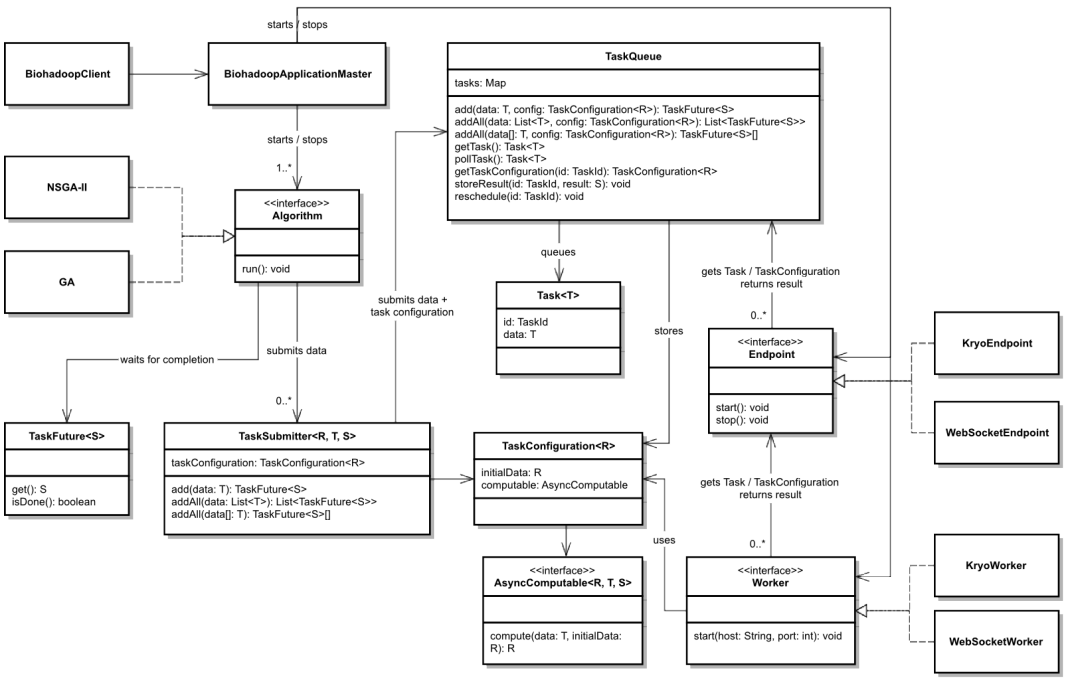

Figure 4.2.: Biohadoop UML Diagram

Chapter 5.

Using Biohadoop

One of the design goals of Biohadoop was to provide a framework for distributed computation on the Hadoop platform, that is easy to use. The result is a simple API, that can be used to implement algorithms. This chapter introduces into the necessary steps to write an algorithm that is capable of being run by Biohadoop on a Hadoop environment. In addition, the presented algorithm uses the capabilities of the task system to distribute parts of its work to Biohadoops workers, to achieve a higher level of parallelism.

5.1. Example algorithm

For the sake of simplicity we reuse the example algorithm `Sum` of chapter 4.1. As mentioned there, it is a simple algorithm, whose only purpose is to sum the values of an integer array. In listing **??** we have already a first implementation, that is capable of being run by Biohadoop. But in that example, no usage of the task system and therefor distribution is taken. Also, all the parameters are fixed. In the next sections, we will create step by step an implementation that is configurable using Biohadoops configuration file, and that is able to distribute its work (see section **??** for more details about the configuration file).

To make it more transparent how and when we use the task system, we modify the algorithm a little bit. It should use a number of integer arrays (the number is called `CHUNKS` from here on), each array should be of a fixed size (the size is called `CHUNK_SIZE` from here on). If we would append all the chunks one after another, we would have one big integer array of size $\texttt{CHUNCKS} * \texttt{CHUNCK\_SIZE}$. If you look at it from the other side, a big integer array would have to be split into several chunks, to make it suitable for parallel computation. So there is no real difference to the original algorithm, it just preserves us from doing some computations to split a big integer array into smaller arrays.

Listing 5.1 shows how the data is created. The method `buildData(int chunks, int chunkSize)` takes `CHUNCKS` and `CHUNCK_SIZE` as input argument and returns `CHUNKS` integer arrays, each of size `CHUNCK_SIZE`. The arrays are

filled with consecutive numbers, starting from 0. The consecutive numbers continue between the boundaries of adjacent arrays. For example, array0=[0,1,2], array1=[3,4,5], ...

Listing 5.1: Building the integer arrays for the `Sum` algorithm

```
private int[][] buildData(int chunks, int chunkSize) {
  int[][] data = new int[chunks][chunkSize];
  for (int i = 0; i < chunks; i++) {
    for (int j = 0; j < chunkSize; j++) {
      data[i][j] = i * chunkSize + j;
    }
  }
  return data;
}
```

5.1.1. Configuring the algorithm

Lets begin with modifying the algorithm in a way, such that it can be configured by Biohadoops configuration file. The configuration file is read by Biohadoop at its startup, and the private parameters for an algorithm instance are provided when the `compute(SolverId solverId, Map<String, String> properties)` method of the algorithm is called. If we use the configuration file of listing **??**, we see that we get two properties:

- `CHUNKS`: the number of integer arrays, that should be produced by the algorithm.
- `CHUNK_SIZE`: the size of each integer array.

Those parameters are of type `String`, so we need to convert them first to integer values, to make them suitable for the `buildData(int, int)` method, mentioned in listing 5.1. After this steps are taken, we have configured our algorithm through a Biohadoop configuration file, and we have initialized the data. Listing 5.2 shows, how the algorithm looks at the current stage. The `buildData(int, int)` method was skipped, to make the code more concise, it can be found in listing 5.1.

Listing 5.2: Configuring the `Sum` algorithm and initializing the data

```
public class SumAlgorithm implements Algorithm {

  public static final String CHUNKS = "CHUNKS";
  public static final String CHUNK_SIZE = "CHUNK_SIZE";

```

```
6   @Override
7   public void compute(SolverId solverId, Map<String, String>
        properties)
8       throws AlgorithmException {
9     // Read properties from configuration file
10    int chunks = getPropertyAsInt(properties, CHUNKS);
11    int chunkSize = getPropertyAsInt(properties, CHUNK_SIZE)
          ;
12
13    // Prepare sample data
14    int[][] data = buildData(chunks, chunkSize);
15
16    // ... more to come in the next sections
17  }
18
19  // Convenience method to parse String to int
20  private int getPropertyAsInt(Map<String, String>
      properties, String key)
21       throws AlgorithmException {
22    String value = null;
23    try {
24      value = properties.get(key);
25      return Integer.parseInt(value);
26    } catch (Exception e) {
27      throw new AlgorithmException("Could not convert
            property " + key
28           + " to long, value was " + value, e);
29    }
30  }
31
32  // ... method for building data
33 }
```

5.1.2. Parallelization using the task system

The parallelization of an algorithm using Biohadoop consists of four steps:

1. provide an implementation of `AsyncComputable`, that can be used to compute the task results by the workers.

2. create a task submitter, that can be used to add new tasks to the task system.

3. submit tasks to the task system, by using the before mentioned submitter.

4. wait for the tasks to complete. It is possible to block during the wait process, but is also possible to do other work in between.

In step 1 we have to implement `AsyncComputable`. The resulting class can be used by the workers to compute the results for the tasks. In our case, the workers get as input arguments an integer array and have to compute the sum over all of its elements. Listing 5.3 shows how this would be done in a class with name `AsyncSumComputation`. We see that this class is typed, the types have to match the task submitter in step 2. The types are `Object` for the `initialData`, but we don't take usage of it in this case. The input data is of type `int[]`, which matches our integer arrays. As output we have a single integer, which is the result of the summation. The content of the method itself is straight forward.

Listing 5.3: Summation of all values in an integer array done in an `AsyncComputable` class that is suitable to be run by workers

```
1  public class AsyncSumComputation implements AsyncComputable<
       Object, int[], Integer> {
2
3    @Override
4    public Integer compute(int[] data, Object initialData)
5        throws ComputeException {
6      int sum = 0;
7      for (int i : data) {
8        sum += i;
9      }
10     return sum;
11   }
12
13 }
```

Step 2 demands the creation of a task submitter, to be able to add tasks to the task system. Lets clarify at this point, what a single task means in our example program: each single task consists of computing the sum of an integer array. The data for this task is the given integer array. This data is send to workers, that use a given `AsynComputable` class to compute the result - in our case the sum of the integer array computed by the `AsyncSumComputation` class. The result is then returned to the algorithm, at which point the task has completed.

The `Sum` example algorithm uses many integer arrays, therefor we have many tasks, one task for every integer array. Tasks are submitted to the task system, by submitting their data through a task submitter. The usage of a task submitter is advised, as it hides some complexity from the algorithm author. But it is possible to communicate to the task system without a submitter. We choose to use the task submitter, and create a new one the following way:

```
TaskSubmitter<int[], Integer> taskSubmitter = new
    SimpleTaskSubmitter<Object, int[], Integer>(
    AsyncSumComputation.class);
```

This task submitter is configured to get an object of type `Object` as `initialData`. As we don't want to use this feature, we don't provide such an `initialData`. The submitter is further configured to receive tasks of type `int[]` and to return `TaskFutures` of type Integer, that can be used by the algorithm to retrieve the results from the task system. The tasks of type `int[]` that we are going to submit are exactly the arrays, that we created in the previous section, using the `buildData(int, int)`. We store all those arrays in a single 2-dimensional array of type `int[][]` and call this array `data`. The only parameter that we provide to the new instance of `SimpleTaskSubmitter` is the `AsyncSumComputation` class. This class is used by the workers to compute the results for the tasks, that are submitted through this task submitter.

In step 3 we start to submit tasks to the task system. As we want to submit a number of arrays, it is best to do it in a loop. We also want to be able to use the results of the different sums to compute the final sum. Therefor we have to store the `TaskFutures`, to be able to retrieve their results:

```
List<TaskFuture<Integer>> taskFutures = new ArrayList<>();
for (int i = 0; i < chunks; i++) {
  TaskFuture<Integer> future = taskSubmitter.add(data[i]);
  taskFutures.add(future);
}
```

Step 4 consists of waiting for the results. The `get()` method of a `TaskFuture` provides us the result, but blocks until a result is available. `TaskFuture` provides also a `isDone()` method, which we can use to check for finished computations in a non-blocking way. If `isDone()` returns true, we can get the results by invoking `get()`, that doesn't block anymore at that time.

We choose the simpler `get()` approach at the cost of blocking the algorithm. As we need all results to proceed, this makes no difference to the non-blocking variant. Because we got several `TaskFuture` objects from our task submissions in step 3, we have to loop on all the `TaskFuture` objects to get all of their results. By summing up the results, we get the final sum over all arrays:

```
int sum = 0;
for (TaskFuture<Integer> future : taskFutures) {
  sum += future.get();
}
```

After this step completes, we have the final sum over all arrays, and the algorithm can terminate.

Something that wasn't mentioned until now is, that the task submission and result retrieval operations may throw checked exceptions. This exceptions must be caught, therefor the submission and retrieval must be surrounded by a `try ... catch` block. In the example code above, this step was omitted to keep the code clear and concise. In the appendix, the whole code can be found, once for the `Algorithm` (see listing A.2), and once for the `AsyncComputable` (see listing A.3). More examples can be found at [45].

5.2. Run Biohadoop

Although the main purpose of Biohadoop is to be run in a Hadoop environment, it can also be run in a local environment. This is for example useful when new algorithms are developed. In this case, the whole process of compilation, deployment to a Hadoop environment and testing can be abbreviated.

To run Biohadoop, three components must be available (four if Biohadoop is started in a Hadoop environment):

- An installation of Java in version 1.7 or higher. From here on it is assumed, that Java is installed and configured and that the `JAVA_HOME` variable points to this installation.
- All necessary libraries must be present and accessible in one or several folders.
- A valid configuration file must be present and accessible.
- The fourth component is only necessary when running Biohadoop in a Hadoop environment. This component is Hadoop itself, which must be available in a version ≥ 2.

If those three components are provided (four for running on Hadoop), Biohadoop can be started. In a local environment, this is done by setting the right class paths when launching the program. In a Hadoop environment, the configuration option `includePaths` must be set, to include the necessary files (see **??** on more information about configuring Biohadoop). Those paths have to point to valid locations of an accessible HDFS file system.

Biohadoop was developed using Maven [46]. Therefor it is rather easy to get all the needed libraries, as they are declared as dependencies. The source code for Biohadoop can be found at [47]. By invoking the following command on the

projects root folder, all dependencies are accumulated and put to the sub folder `target/dependency`:

```
mvn dependency:copy-dependencies
```

From there, they can be directly referenced through Javas `classpath` option when running in a local environment. When running in a Hadoop environment, the libraries need to be copied to an accessible HDFS file system, and this location must be present in the before mentioned `includePaths` configuration option.

For a quick tutorial on how to compile and run Biohadoop from the sources, please refer to appendix A.4. To use Biohadoop in a Hadoop environment, such an environment must be present. It can be a difficult task to configure such a Hadoop environment, therefor in appendix A.5 a simple method can be found, to use a pre-build Hadoop environment. The only dependency this environment has, is Docker [48], which is a simple and lightweight runtime for containers, available on all major operating systems.

5.2.1. Local environment

To start Biohadoop in a local environment, the class paths need to be set to include the necessary libraries. The necessary libraries can be obtained by invoking the Maven command, outlined above. As an additional parameter, the `-Dlocal` option must be provided to Java. This is the only way to tell Biohadoop that it is launched in a local environment. If this parameter is missing, Biohadoop will complain that it can't connect to Hadoop.

Lets assume, that all of the necessary libraries can be found at the location `/home/user/biohadoop/libs`, and the configuration file can be found at `/home/user/biohadoop/configs/simple-config-json`. Then, Biohadoop can be started in local mode by running the following command:

```
java -Dlocal -cp /home/user/biohadoop/libs/* at.ac.uibk.dps.
    biohadoop.hadoop.BiohadoopApplicationMaster /home/user/
    biohadoop/configs/simple-config-json
```

A little bit hidden in the command we find the main class that starts Biohadoop, `at.ac.uibk.dps.biohadoop.hadoop.BiohadoopApplicationMaster`. This class takes care of starting the configured algorithms, adapters and workers. After all of the algorithms have terminated, either because they have finished their computation or because they had errors, Biohadoop shuts down.

When running Biohadoop in the local environment, all workers are started as threads in the same JVM as Biohadoop (in contrast to a Hadoop environment, where the workers are started in separate JVMs on perhaps different machines).

This leads to the fact, that the workers have full access to all (static) objects of Biohadoop, for example the `Environment` object (see **??** for a description of this object). But when those workers are started in a Hadoop environment, this access is not given anymore. So, one has to take care to rely only on the objects and properties, that are provided to the used methods.

5.2.2. Hadoop environment

To start Biohadoop in a Hadoop environment (for example in the one provided in appendix A.5), all needed libraries and configuration files must be present in the HDFS file system. In addition, Biohadoops jar file must be accessible through the local file system, as it is started directly by Hadoop. The location of the libraries and configuration files in the HDFS file system must be configured in the configuration file, that is provided to Biohadoop on startup.

Lets assume, that all of the necessary libraries can be found at the HDFS location `/biohadoop/libs`, the configuration file can be found at the HDFS location `/biohadoop/configs/simple-config-json` and that those paths are part of the configuration file that is provided to Biohadoop at startup. Further more, Biohadoops jar file can be found at `/home/user/biohadoop/biohadoop.jar`. Then, Biohadoop can be started in Hadoop mode by running the following command:

```
1 yarn jar /home/user/biohadoop/biohadoop.jar at.ac.uibk.dps.
    biohadoop.hadoop.BiohadoopClient /biohadoop/configs/
    simple-config-json
```

As one can see, now the command `yarn` is used to launch Biohadoop. This command takes care of setting all the needed Hadoop environment variables, after which it starts the provided main class. The `yarn` command is part of Hadoop since version 2, and should be available if Hadoop is configured in the right way. If it is not available, please contact the system administrator.

In contrast to running Biohadoop in local mode, we have now a different main class, that is launched. This is due to the fact, that Yarn needs a startup class, from where it loads the main program. By looking at the source code of `at.ac.uibk.dps.biohadoop.hadoop.BiohadoopClient`, one will notice that it starts the `BiohadoopApplicationMaster` behind the curtains (`BiohadoopApplicationMaster` is the class that is directly started in local mode).

When running Biohadoop in a Hadoop environment, all workers are started in their own containers, which are under the control of Hadoop. The result is, that the workers can not access the (static) objects of Biohadoop, if one wants

to access some properties of Biohadoop, this must be done through the provided communication facilities of Biohadoop. It is nevertheless possible to implement some different communication facilities, if this is needed.

Chapter 6.

Results

Chapter 7.

Conclusions

- suitable for bio-inspired optimization techniques? - suitable for scientific computing? - suitable for very large scale computing? - split protocol and serialization to make communication more customizable

Appendix A.

A.1. JSON Schema for Biohadoops configuration file

Listing A.1: JSON Schema [49] for the Biohadoop configuration file

```
{
  "$schema": "http://json-schema.org/schema#",
  "title": "Biohadoop configuration schema",
  "type": "object",
  "properties": {
    "communicationConfiguration": {
      "description": "Configuration for adapters and workers
         , that are started by Biohadoop",
      "type": "object",
      "properties": {
        "dedicatedAdapters": {
          "description": "A list of adapters of dedicated
             pipelelines, that should be started by
             Biohadoop. The adapters of the default
             pipeline are started automatically  (pre-
             configured)",
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "adapter": {
                "description": "Canonical name of a class
                   that implements the interface at.ac.uibk
                   .dps.biohadoop.tasksystem.adapter.
                   Adapter",
                "type": "string"
              },
              "pipelineName": {
                "description": "The name of the dedicated
                   pipeline, this adapter belongs to",
                "type": "string"
              }
```

```
            },
            "required": ["adapter", "pipelineName"]
          }
        },
        "workerConfigurations": {
          "description": "A list of workers, that should be
              started by Biohadoop",
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "worker": {
                "description": "Canonical name of a class
                    that implements the interface at.ac.uibk
                    .dps.biohadoop.tasksystem.worker.Worker"
                    ,
                "type": "string"
              },
              "pipelineName": {
                "description": "The name of the dedicated
                    pipeline, this worker belongs to",
                "type": "string"
              },
              "count": {
                "description": "Number of worker instances
                    that should be launched",
                "type": "integer"
              }
            },
            "required": ["worker", "pipelineName", "count"]
          }
        }
      },
      "required": ["dedicatedAdapters", "
          workerConfigurations"]
    },
    "globalProperties": {
      "description": "A map with String as key and value.
          Contains properties that are global to Biohadoop",
      "type": "object",
      "patternProperties": {
        ".": {
          "type": "string"
        }
      }
```

```
61      },
62      "includePaths": {
63        "description": "The library paths to include when
             running on a Hadoop cluster",
64        "type": "array",
65        "items": {
66          "type": "string"
67        }
68      },
69      "solverConfigurations": {
70        "description": "Configuration information for
             algorithms, that Biohadoop should run",
71        "type": "array",
72        "items": {
73          "type": "object",
74          "properties": {
75            "algorithm": {
76              "description": "Canonical name of a class that
                   implements the interface at.ac.uibk.dps.
                   biohadoop.algorithm.Algorithm",
77              "type": "string"
78            },
79            "name": {
80              "description": "Identification for the algorithm
                   . Can be found e.g. in the log files",
81              "type": "string"
82            },
83            "properties": {
84              "description": "A map with String as key and
                   value. Contains properties specific to this
                   algorithm. This properties are passed as
                   arguments to the algorithm.",
85              "type": "object",
86              "patternProperties": {
87                ".": {
88                  "type": "string"
89                }
90              }
91            }
92          },
93          "required": ["algorithm", "name", "properties"]
94        }
95      }
96    },
```

```
97   "required": ["communicationConfiguration", "
        globalProperties", "includePaths", "
        solverConfigurations"]
98 }
```

A.2. Example algorithm: Sum

Listing A.2: Source code for the example `Sum` algorithm elaborated in chapter 5

```
1  /**
2   * Simple example algorithm, that sums the values of integer
        array. It uses
3   * Biohadoops task system to distribute chunks of the
       integer array to waiting
4   * workers, where they are summed up, after which the result
        is returned
5   *
6   * @author Christian Gapp
7   *
8   */
9  public class SumAlgorithm implements Algorithm {
10
11   public static final Logger LOG = LoggerFactory
12       .getLogger(SumAlgorithm.class);
13
14   public static final String CHUNKS = "CHUNKS";
15   public static final String CHUNK_SIZE = "CHUNK_SIZE";
16
17   @Override
18   public void compute(SolverId solverId, Map<String, String>
         properties)
19       throws AlgorithmException {
20     // Read properties from configuration file
21     int chunks = getPropertyAsInt(properties, CHUNKS);
22     int chunkSize = getPropertyAsInt(properties, CHUNK_SIZE)
          ;
23
24     // Prepare sample data
25     int[][] data = buildData(chunks, chunkSize);
26
27     // Get a task submitter for default pipeline. Declare
28     // AsyncSumComputation as class that should be run by
          the workers to
```

```
29      // compute the results
30      TaskSubmitter<int[], Integer> taskSubmitter = new
            SimpleTaskSubmitter<Object, int[], Integer>(
31          AsyncSumComputation.class);
32
33      try {
34        List<TaskFuture<Integer>> taskFutures = new ArrayList
              <>();
35        // Submit the sample data to the task system for
              asynchronous
36        // computation. Each submission defines one task
37        for (int i = 0; i < chunks; i++) {
38          TaskFuture<Integer> future = taskSubmitter.add(data[
                i]);
39          taskFutures.add(future);
40        }
41
42        int sum = 0;
43        // Wait for all tasks to finish and sum up the result.
              The get()
44        // method blocks until the result for the TaskFuture
              is available
45        for (TaskFuture<Integer> future : taskFutures) {
46          sum += future.get();
47        }
48        LOG.info("The computation result is {}", sum);
49      } catch (TaskException e) {
50        throw new AlgorithmException("Could not submit data");
51      }
52    }
53
54    /**
55     * Convenience method to parse the input arguments. Throws
            an
56     * AlgorithmException if there was an error during parsing
57     *
58     * @param properties
59     *            contain the configuration for this algorithm
          , as defined in
60     *            the configuration file
61     * @param key
62     *            for which the value should be retrieved from
          the map
63     * @return
64     * @throws AlgorithmException
```

```
65      *             if there was an exception during the
            parsing
66      */
67    private int getPropertyAsInt(Map<String, String>
          properties, String key)
68        throws AlgorithmException {
69      String value = null;
70      try {
71        value = properties.get(key);
72        return Integer.parseInt(value);
73      } catch (Exception e) {
74        throw new AlgorithmException("Could not convert
              property " + key
75            + " to long, value was " + value, e);
76      }
77    }
78
79    /**
80     * Builds <tt>chunks</tt> number of integer arrays, each
            one of size
81     * <tt>chunkSize</tt>. The arrays are filled with
            consecutive numbers,
82     * starting from 0. The consecutive numbers continue
            between the boundaries
83     * of adjacent arrays. For example, array0=[0,1,2], array1
            =[3,4,5], ...
84     *
85     * @param chunks
86     *            number of integer arrays
87     * @param chunkSize
88     *            size of each integer array
89     * @return <tt>chunk</tt> number of integer arrays, each
            one of size
90     *         <tt>chunkSize</tt>. The arrays are filled with
            consecutive
91     *         numbers, starting from 0. The consecutive
            numbers continue
92     *         between the boundaries of adjacent arrays
93     */
94    private int[][] buildData(int chunks, int chunkSize) {
95      int[][] data = new int[chunks][chunkSize];
96      for (int i = 0; i < chunks; i++) {
97        for (int j = 0; j < chunkSize; j++) {
98          data[i][j] = i * chunkSize + j;
99        }
```

```
100     }
101     return data;
102   }
103 }
```

A.3. Example algorithm: Sum - AsyncComputable

Listing A.3: Source code for the `AsyncSumComputation` part of the `Sum` algorithm elaborated in chapter 5

```
1  public class AsyncSumComputation implements AsyncComputable<
       Object, int[], Integer> {
2
3    @Override
4    public Integer compute(int[] data, Object initialData)
5        throws ComputeException {
6      int sum = 0;
7      for (int i : data) {
8        sum += i;
9      }
10     return sum;
11   }
12
13 }
```

A.4. Biohadoop quickstart

The quickstart uses the pre-configured Hadoop environment, that can be found in appendix A.5. This environment is installed during the following steps. In addition, some example algorithms are installed, that can be found at [45].

The requirements for this quickstart are Docker $\geq$ 1.0, Maven with MVN_HOME set to the correct path and an installed gnome-terminal (provided by default in Ubuntu). The quickstart takes usage of some scripts, all of them are provided by the used sources.

A.4.1. Build and start the Hadoop environment

The first step is to build and start a Hadoop environment with one master and two slave nodes. The master node is started inside a red gnome-terminal window, at the end it prints the password for the root user:

```
git clone https://github.com/gappc/docker-biohadoop.git
sudo docker build -t="docker-biohadoop" ./docker-biohadoop
chmod +x ./docker-biohadoop/scripts/*.sh
./docker-biohadoop/scripts/docker-run-hadoop.sh 2
```

A.4.2. Build Biohadoop and copy it to the Hadoop environment

The second step is to build and copy Biohadoop to the Hadoop environment:

```
git clone https://github.com/gappc/biohadoop.git
chmod +x ./biohadoop/scripts/*.sh
./biohadoop/scripts/copy-files.sh
```

A.4.3. Build example algorithms and copy them to the Hadoop environment

The third step is to build and copy the example algorithms to the Hadoop environment:

```
git clone https://github.com/gappc/biohadoop-algorithms.git
chmod +x ./biohadoop-algorithms/scripts/*.sh
./biohadoop-algorithms/scripts/copy-algorithms.sh
```

A.4.4. Run Biohadoop in the Hadoop environment

To run the Echo example in Hadoop , the following command must be issued in the red terminal - if there is no red terminal, please check [50]. This example assumes that the jar `biohadoop-0.3.0-SNAPSHOT.jar` is used:

```
yarn jar /tmp/lib/biohadoop-0.3.0-SNAPSHOT.jar at.ac.uibk.
    dps.biohadoop.hadoop.BiohadoopClient /biohadoop/conf/
    biohadoop-echo.json
```

Other examples of Biohadoop programs can be found at [45]. You can use them to try out Biohadoop and as a template for your own experiments. Good luck and have fun :)

A.5. Pre-build Hadoop environment using Docker

This section provides a way to run a pre-configured Haddop environment using Docker [48]. Docker uses its so called Dockerfiles for its configuration. The here presented solution installs Apache Hadoop 2.5.0, Apache Oozie 4.0.1 and

Apache ZooKeeper 3.4.6 as a cluster environment. The solution is based on `sequenceiq/hadoop-docker` [51].

A.5.1. Build the Hadoop environment

The following commands are used to clone the repository to the current local directory and to build the Docker image. Be aware, that only during the cloning process about 400MB of data is transferred. This is due to Oozie, which is delivered in a compiled form.

```
1 git clone https://github.com/gappc/docker-biohadoop.git
2 cd docker-biohadoop
3 sudo docker build -t="docker-biohadoop" .
```

The project docker-biohadoop provides two scripts, located in the scripts directory, that can be used to start and stop docker-biohadoop instances. Those scripts need to be executable:

```
1 chmod +x scripts/*.sh
```

A.5.2. Run Hadoop

After that, we are able to start Hadoop instances by using the first script: `docker-run-hadoop.sh`. This script starts a number of Hadoop instances. It takes the number of slaves (nr-of-slaves) as argument. The script starts one Docker container as Hadoop master and additional nr-of-slaves Docker containers as Hadoop slaves. For example, one Hadoop master instance and two slaves are started by the following command:

```
1 scripts/docker-run-hadoop.sh 2
```

Note that also the master is used for computational purposes. Therefor, Hadoop has 3 machines for computation with the settings above.

After invoking `docker-run-hadoop.sh`, a gnome-terminal is started for every Docker container. The master containers terminal has a red color, the slaves terminals are yellow. The master container starts the Hadoop environment, which may take some time (depending on the hardware and the number of slaves). After this initialization, the Hadoop cluster is ready for usage. Try to invoke the command `jps` on all running containers to look if Hadoop is running:

```
1 jps
```

On the master node, it should output:

- DataNode

- JobHistoryServer
- NodeManager
- NameNode
- QuorumPeerMain
- ResourceManager
- SecondaryNameNode

On the slave nodes it should output:

- DataNode
- NodeManager

A.5.3. Stopping Hadoop

By using the following command, all running Docker containers are forcefully stopped and their interfaces are removed from the host. It is no problem to forcefully stop Docker containers, as they don't keep any state by default.

```
1 scripts/docker-stop-all.sh
```

A.5.4. SSH access

The master node is accessible with user root, a password is generated on each startup and printed on the terminal. Consider adding your SSH key to the Dockerfile if you are going to use docker-biohadoop often.

List of Figures

List of Tables

Listings

Bibliography

[1] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[2] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[3] The message passing interface (mpi) standard. `http://www.mcs.anl.gov/research/projects/mpi/`. last access: 08.10.2014.

[4] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. Global arrays: A portable shared-memory programming model for distributed memory computers. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349. IEEE Computer Society Press, 1994.

[5] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47. ACM, 2005.

[6] Apache hadoop. `http://hadoop.apache.org/`. last access: 01.10.2014.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[9] Michael Stonebraker, Daniel Abadi, David J DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[10] Avrilia Floratou, Nikhil Teletia, David J DeWitt, Jignesh M Patel, and Donghui Zhang. Can the elephants handle the nosql onslaught? *Proceedings of the VLDB Endowment*, 5(12):1712–1723, 2012.

[11] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[12] Xueyuan Su and Garret Swart. Oracle in-database hadoop: when mapreduce meets rdbms. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 779–790. ACM, 2012.

[13] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[14] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[15] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analyses. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 277–284. IEEE, 2008.

[16] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[17] Joshua Rosen, Neoklis Polyzotis, Vinayak Borkar, Yingyi Bu, Michael J Carey, Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. Iterative mapreduce for large scale machine learning. *arXiv preprint arXiv:1303.3517*, 2013.

[18] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. Stochastic gradient boosted distributed decision trees. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 2061–2064. ACM, 2009.

[19] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[20] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[21] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[22] Apache hadoop 2.2.0. `http://hadoop.apache.org/docs/r2.2.0/`. last access: 01.10.2014.

[23] Apache spark. `https://spark.apache.org/`. last access: 01.10.2014.

[24] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[25] Apache mahout. `http://mahout.apache.org/`. last access: 01.10.2014.

[26] Apache storm. `https://storm.apache.org/`. last access: 01.10.2014.

[27] Apache tez. `https://tez.apache.org/`. last access: 08.10.2014.

[28] Apache samza. `https://samza.incubator.apache.org/`. last access: 08.10.2014.

[29] Folding@home. `http://folding.stanford.edu/home/`. last access: 01.10.2014.

[30] Seti@home. `http://setiathome.ssl.berkeley.edu/`. last access: 01.10.2014.

[31] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 4. ACM, 2012.

[32] Apache oozie. `http://oozie.apache.org/`. last access: 01.10.2014.

[33] S Alexander. On the history of combinatorial optimization (till 1960). *Handbooks in Operations Research and Management Science: Discrete Optimization*, 12:1, 2005.

[34] Matthias Ehrgott. *Multicriteria optimization*, volume 2. Springer, 2005.

[35] Xin-She Yang. *Nature-inspired metaheuristic algorithms*. Luniver press, 2010.

[36] SN Sivanandam and SN Deepa. *Genetic Algorithm Optimization Problems*. Springer, 2008.

[37] James Kennedy. Particle swarm optimization. In *Encyclopedia of Machine Learning*, pages 760–766. Springer, 2010.

[38] Marco Dorigo and Mauro Birattari. Ant colony optimization. In *Encyclopedia of Machine Learning*, pages 36–39. Springer, 2010.

[39] Juan José Durillo, Antonio J Nebro, Francisco Luna, and Enrique Alba. A study of master-slave approaches to parallelize nsga-ii. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[40] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[41] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[42] Hdfs high availability using the quorum journal manager. `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html`. last access: 12.11.2014.

[43] Hdfs high availability using nfs. `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html`. last access: 12.11.2014.

[44] Yarn high availability. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html. last access: 13.11.2014.

[45] Biohadoop example algorithms. https://github.com/gappc/biohadoop-algorithms. last access: 21.08.2014.

[46] Maven. http://maven.apache.org/. last access: 11.09.2014.

[47] Biohadoop. https://github.com/gappc/biohadoop. last access: 11.09.2014.

[48] Docker. https://www.docker.com/. last access: 11.09.2014.

[49] Json schema. http://json-schema.org/. last access: 28.07.2014.

[50] Hadoop environment using docker. https://github.com/gappc/docker-biohadoop. last access: 11.09.2014.

[51] Docker. https://index.docker.io/u/sequenceiq/hadoop-docker/. last access: 03.06.2014.