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, 15 December 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 15 December 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

Bio-inspired optimization algorithms are used to find good (preferably best) solutions to a given optimization problem. They mimic the behavior of biological agents. A prominent example is the genetical algorithm (GA) [1] that uses a simplified model of natural evolution to improve the solutions for an optimization problem. Another example is the particle swarm optimization (PSO) [2] that is based on the movement of bird flocks.

Most optimization problems are NP-hard. It is not feasible to explore all possible solutions as this would take to much time. Bio-inspired optimization algorithms can't change this fundamental issue, but in many cases their approximation techniques provide "good enough" solutions to a problem in an acceptable time.

Parallelization techniques can be used to further reduce the computational time or to search through more candidate solutions in the same amount of time. There are different approaches to parallelization on current computer architectures. Some of them focus on the exploitation of processing units that are local to a given machine like SIMD, multi core or specialized hardware (GPU, FPGA, ASIC). This local approaches work well in many cases, but the increase of computational power is limited to the amount of resources that single machine can handle.

Further performance increases can only be achieved by distributing the work to several machines, forming a cluster. The downside of this approach is the increased complexity. On the one side, the machines and their components (e.g. hard drives) are unreliable and may fail. This must be detected and handled in an appropriate way. On the other side, the machines need to communicate to each other through the network to exchange work tasks and results.

Apache Hadoop [3] is an open source software project that provides management tools for clusters and libraries to build distributed applications. It is often referred to as an operating system for clusters. It assumes that cluster hardware is inherently unreliable and provides mechanisms to automatically detect and handle failures. This allows distributed applications to focus on the implementation rather than on cluster management concerns. Hadoop also provides the mechanisms to start, stop and monitor the distributed applications, automatically restarting them if needed.

Hadoop versions prior to 2.0 were restricted to the MapReduce [4] computational model. This restriction made it difficult to implement e.g. iterative algorithms, like the bio-inspired optimization techniques. The release of Hadoop 2.0 changed the resource management implementation to YARN [5] which makes no assumptions about the executed application.

One drawback of YARN is its lack of support for application specific communication. This restriction comes by design, as YARNs purpose is to manage the cluster, its resources and the running applications.

Different projects try to improve Hadoop and solve the communication issue. Apache Storm [6] for example implements a stream processing model on top of Hadoop, where cluster nodes are connected to form a graph through which the data flows. Data processing and transformation is performed on the nodes. Apache Spark [7] is an implementation that supports the stream model and a batch processing model on top of Hadoop. In addition, it provides a distributed in-memory store [8].

Biohadoop [9] is another project that aims to simplify the implementation of distributed applications on top of Hadoop. It was developed during this thesis and works based on the master - worker pattern. Biohadoop offers an abstract communication mechanism that makes it easy to distribute work items from the master node to any number of worker nodes. Its focus on the master - worker pattern makes it more lightweight than the previous mentioned solutions, but has the drawback to be restricted to the mentioned pattern.

This thesis introduces Biohadoop and demonstrates its usefulness by implementing two bio-inspired optimization techniques on top of it. It provides additional information about Apache Oozie (a Hadoop workflow tool) [10] and how it was extended to support Biohadoop.

The rest of the document is organized as follows: chapter 2 provides an introduction to bio-inspired optimization algorithms as well as an overview of two common representatives: GA and PSO. Chapter 3 delivers information about Hadoop and Oozie that is needed to understand the functionality of Biohadoop. Chapter 4 explains Biohadoops architecture and the implemented modifications for Oozie (section 4.7). Chapter 6 evaluates the performance of Biohadoop using two different implementations of a GA. The conclusions in chapter 7 summarize the master thesis and the obtained results.

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) [11]. 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 [12].

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

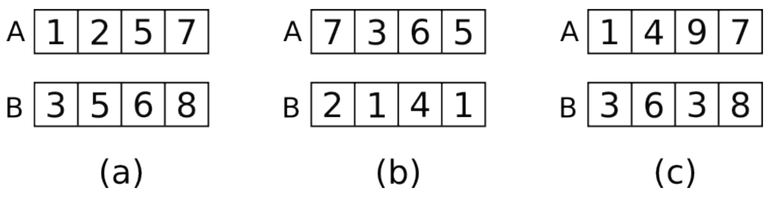

Figure 2.1: Comparing solutions: (a) a is better than b, (b) b is better than a, (c) none is better - a and b are non-dominated

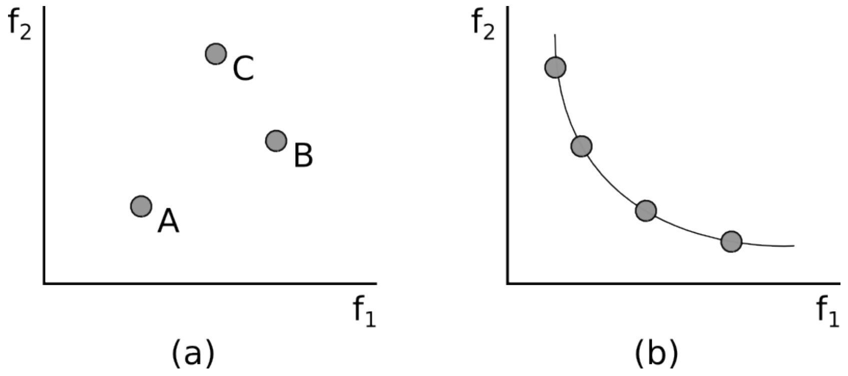

Figure 2.2: Pareto Dominance: (a) A dominates B and C, (b) all points are non-dominated

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.

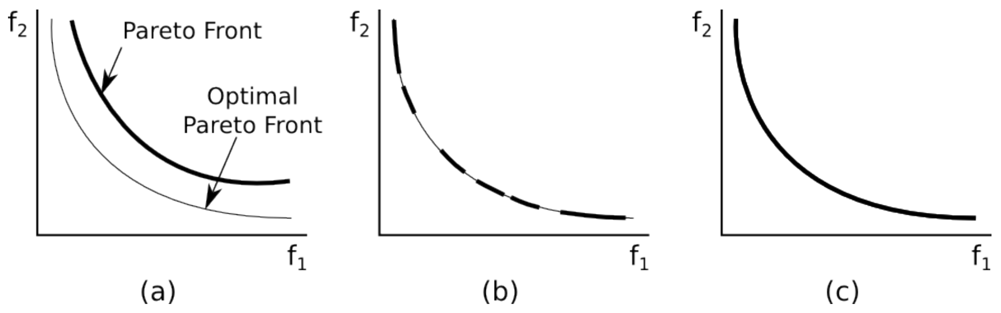

Figure 2.3: Example Pareto Fronts: (a) Pareto Front with bad convergence, (b) Pareto Front with bad divergence, (c) ideal case, where the Pareto Front matches the optimal Pareto Front

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 [13]. 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) [1] imitate the concept of evolution, where only the fittest individuals survive and reproduce. Another example is particle swarm optimization (PSO) [2], that mimics the behavior of a flock of birds. In ant colony optimization (ACO) [14], 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. [1].

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 [15].

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:     $P \leftarrow$ generateInitialPopulation
3:     evaluate($P$)
4:     while !$terminationCriteria$ do
5:         $P'$ $\leftarrow$ recombine($P$)
6:         $P''$ $\leftarrow$ mutate($P'$)
7:         evaluate($P''$)
8:         $P \leftarrow$ 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 [16]. 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.

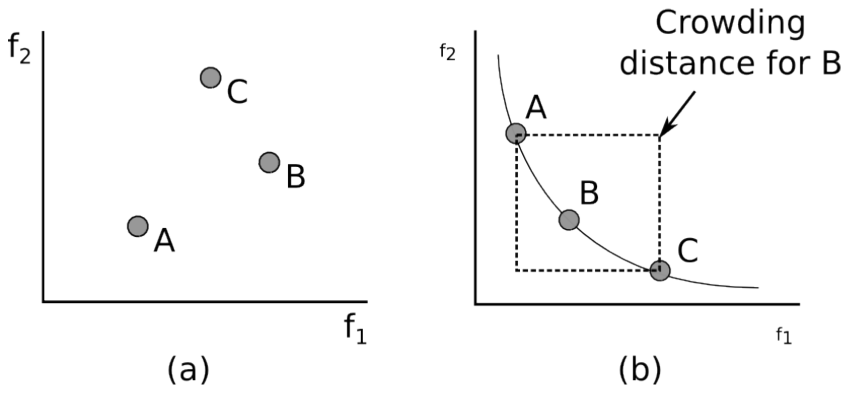

Figure 2.4: Ranking and crowding distance: (a) A has rank 0, B and C rank 1, (b) crowding distance of solution B

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.

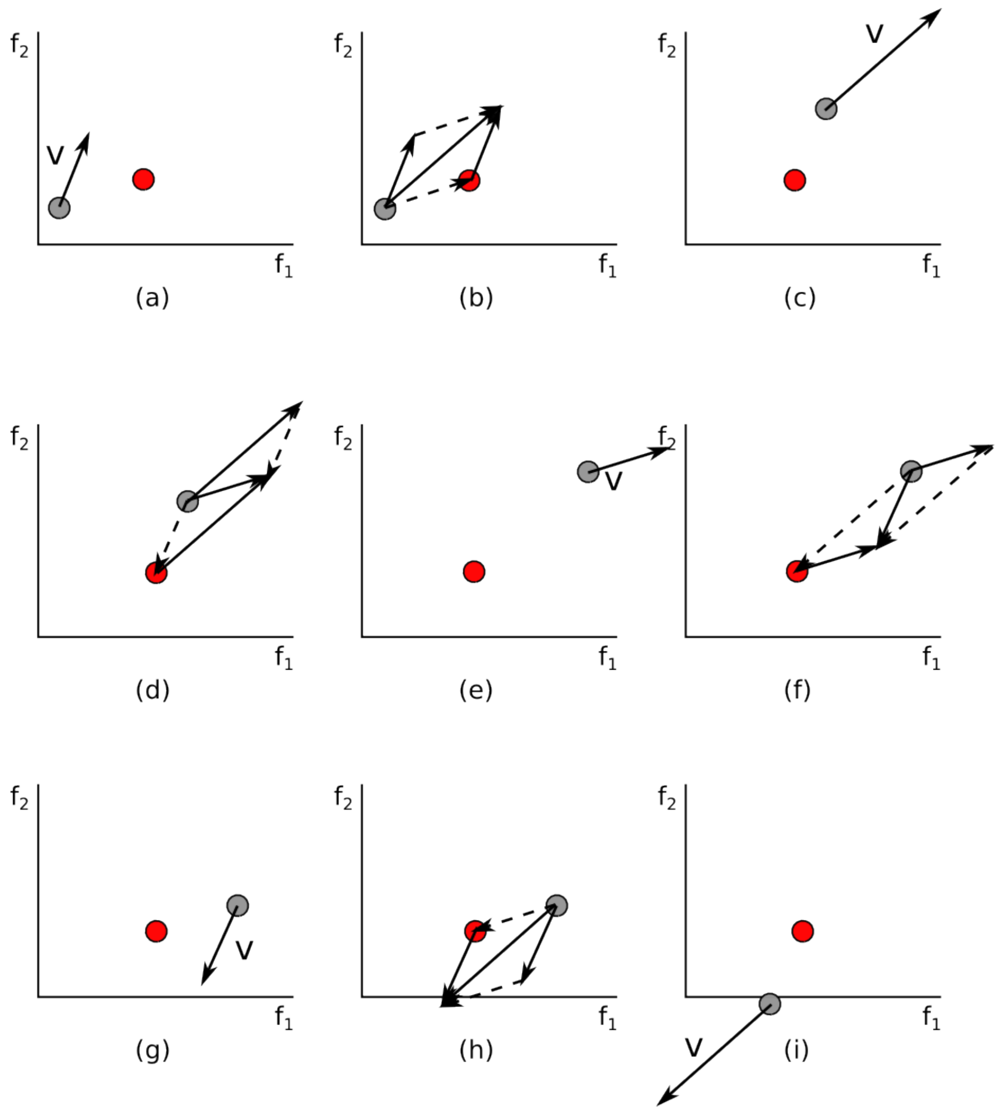

Figure 2.5: Four iterations of PSO for a single particle. The red dot marks gBest, the gray dot the particle. The velocity and position of the particle does rely only on gBest in this example, pBest is not used

Chapter 3

Hadoop

Apache Hadoop [3] is an open source software project for massive data processing and parallel computations in a cluster. It derives from Google's MapReduce [4] and Google File System (GFS) papers [17], 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.

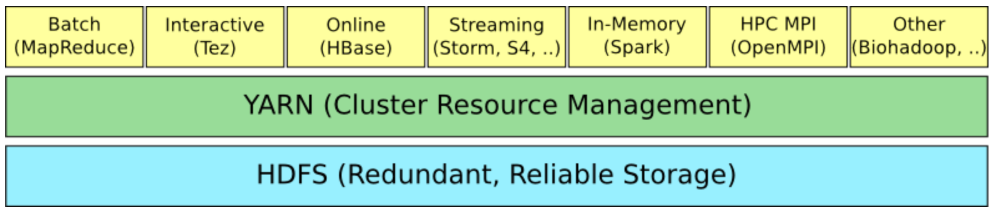

Figure 3.1: Hadoop layers: HDFS forms the base and provides data services, YARN builds on it and provides resource management. The applications run on top of YARN.

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 [18][19].

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.

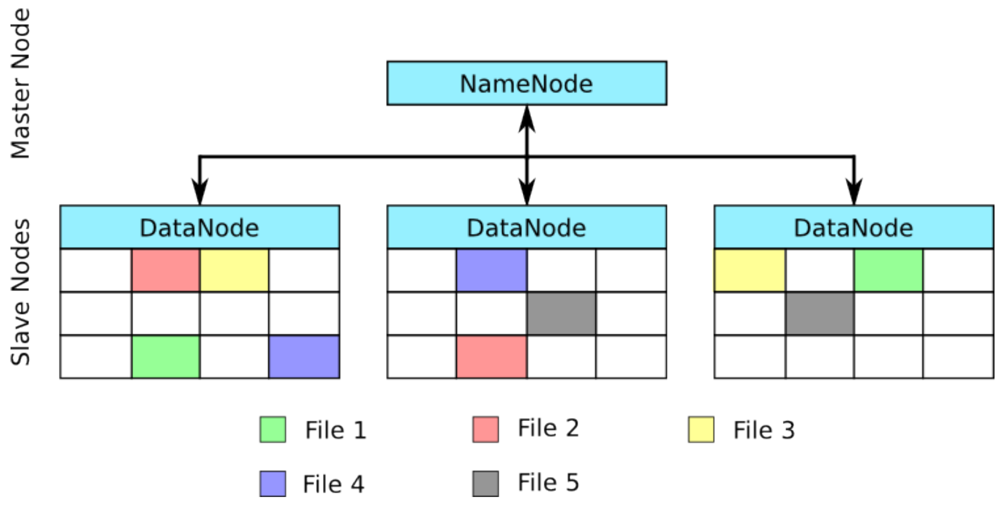

Figure 3.2: HDFS file storage, each file is replicated twice.

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 [20].

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 4.4.

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.

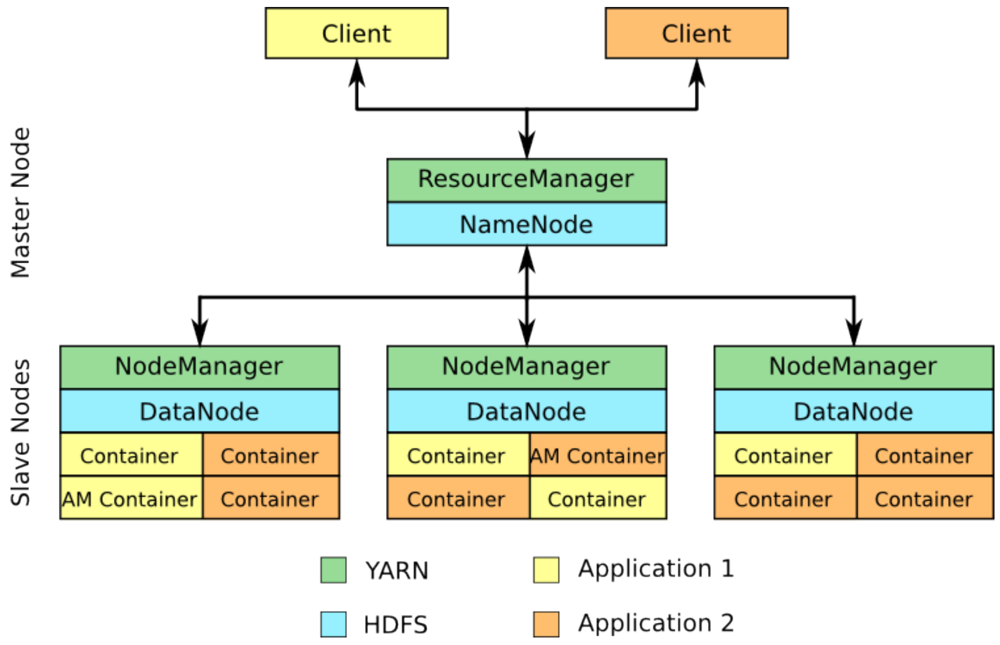

Figure 3.3: Example occupation of a Hadoop cluster for two applications. The NameNode (HDFS) and ResourceManager (YARN) run on the master node. DataNodes (HDFS) and NodeManagers (YARN) run on the slave nodes and communicate with the corresponding services on the master node. The applications, including 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 4.7.

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.

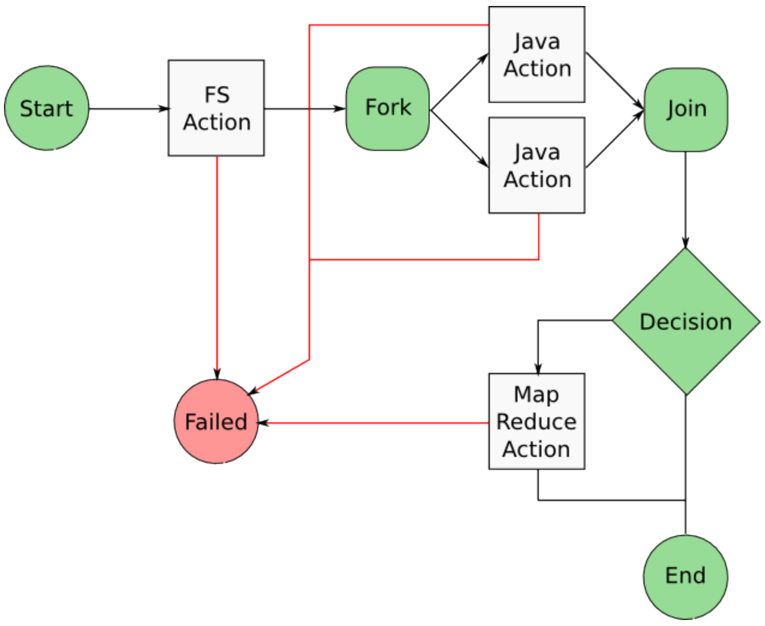

Figure 3.4: Oozie workflow transitions. If any action returns with an error, the workflow transitions to the final state “Failed”, else the workflow advances according to the defined sequence.

Chapter 4

Implementation

4.1 System architecture

Biohadoop is a framework to parallelize algorithms on Apache Hadoop. It works according to the master - worker principle, where one master commands several workers. The master runs an algorithm whose compute intensive parts are executed by the workers in parallel. Time consuming sections of an algorithm, that can be parallelized, are good candidates to run on the workers.

The genetic algorithm (GA) from chapter 2.4.1 is used in the following sections as an example of how to parallelize an algorithm for the master - worker scheme. A GA is an iterative procedure, each iteration creates a set of new individuals (offsprings), evaluates their fitness and selects the best individuals for the next iteration, based on the fitness of all GA individuals. The creation of an offspring and its subsequent fitness evaluation can be combined into a single function. This function consumes most of the time in a GA, but it can be executed in parallel, as it depends only on its input values (the parents) and has no side effects. It is therefore a good example of a parallel task that can be performed by workers, while the rest of the GA algorithm runs on the master (see figure 4.1).

The master - worker architecture was chosen for Biohadoop, because many algorithms can be parallelized by this approach, like the GA and the PSO presented in chapter 2. Another reason is that the master - worker scheme maps very well to YARN (see chapter 3.2).

Biohadoop is written to run as a YARN application and provides features, that otherwise need to be implemented manually, such as:

- support for the execution of several algorithms at the same time in a single Biohadoop instance. This has the advantage that workers, and therefore resources, can be shared by the algorithms. It guarantees also that the algorithms run at the same time. This guarantee can't be given when several Biohadoop instances are used, as YARN decides when it executes a scheduled application

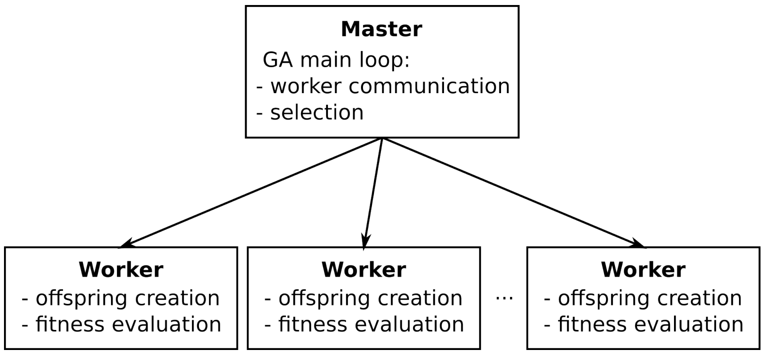

Figure 4.1: Master - worker principle for a GA, one master commands several workers. The master runs the main loop, where it communicates with the workers to get new individuals. Then it selects the best individuals to become the population of the next iteration.

- asynchronous communication between master and workers using the task system (section 4.3). YARN doesn't provide a default communication facility between its containers
- storage and load of arbitrary data sets to and from a file system (section 4.5.1)
- support for the island model, a high level parallelization that can be used to improve the optimization performance of bio-inspired optimization techniques by exchanging results between multiple instances of an algorithm (section 4.5.2)
- support for Apache Oozie through a custom action (section 4.7). This custom action can be used to schedule one or many Biohadoop instances. It also allows the usage of Biohadoop in a larger workflow

Every YARN application needs a client that submits the application to YARN. The YARN client in Biohadoop 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.

Biohadoops ApplicationMaster starts the configured workers using 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 ApplicationMasters main class is `BiohadoopApplicationMaster`. In a local (development) environment, it acts as the main entrance point to run Biohadoop without YARN (more on how to run Biohadoop can be found in 5.1).

The workers are started in additional YARN containers, each worker resides in a dedicated container.

Figure 4.2 shows how Biohadoops architecture maps to the architecture of a typical YARN application. It also gives a first impression of the task system that hides the technical details of the master - worker scheme from the algorithm. The task system consists of a task broker, one ore more endpoints and one or more workers.

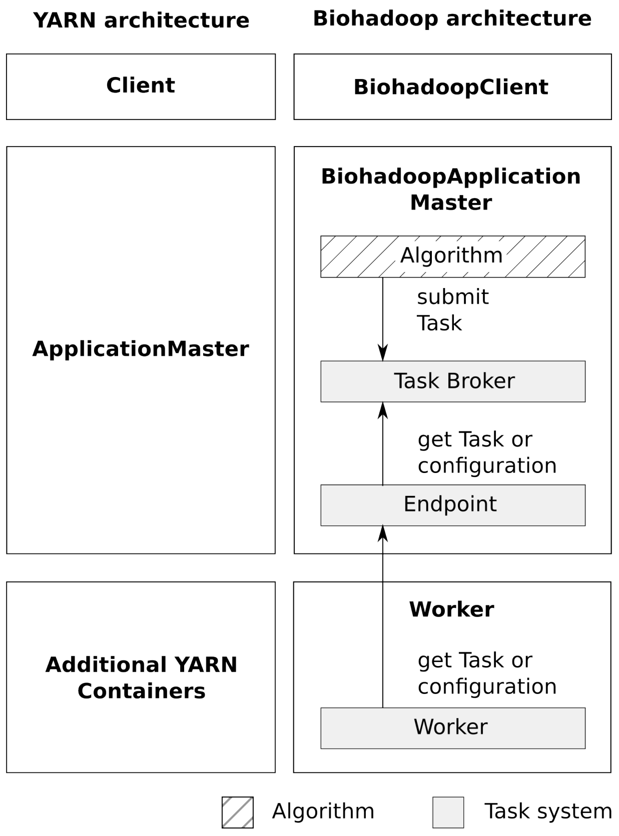

Figure 4.2: The figure shows, how the Biohadoop architecture maps to the architecture of a typical YARN application

An algorithm uses the task system to submit work items, from here on called tasks, to the workers and wait for the results. A task contains data and a reference to a task configuration that defines how to compute the result for the task. In the GA example, a task would be to create individuals and compute their fitness. In this case, the task data represents the parent individuals that are used to create an offspring. The configuration defines the method for offspring creation and fitness computation.

Submitted tasks are queued by the task broker. Endpoints get tasks from the broker 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 broker and from there to the algorithms.

A task configuration is usually shared by many tasks, for example each task in the GA uses the same method to compute an offspring and its fitness value. Therefore, a task configuration is send to a worker only the first time it is needed, and cached by the worker otherwise. This reduces the communication amount between the master and the workers.

Figure 4.3 shows how a worker requests tasks or task configurations to compute the result for a task. The worker requests a task from the master in picture 4.3(a), which is delivered in 4.3(b). The worker then recognizes that it doesn't know how to compute the result for the task - it has to request the task configuration in 4.3(c). The task configuration is returned in 4.3(d), the worker can now compute the task result in 4.3(e). The result is returned in 4.3(f), together with the request for the next task. The next task is delivered in 4.3(g) and as it uses the same task configuration as the prior task, the worker can reuse this configuration and directly compute the result in 4.3(h), after which the result is returned together with the request for the next task.

More details about the task system can be found in section 4.3

Persistence is another feature provided by Biohadoop. It can be used to save and load arbitrary data sets to and from the file system. This is convenient in many cases, for example if an algorithm wants to save its current state and reload this state at the next startup. To get more information about the persistence, refer to section 4.5.1.

Biohadoop is capable of running several algorithms at the same time in a single Biohadoop instance. They all run in the same JVM as Biohadoop does, and can be of arbitrary type. For example it is possible to run two GA instances and one PSO instance at the same time. As YARN doesn't guarantee when an application runs, the mentioned capability of launching several algorithms at the same time in the same JVM is a useful enhancement when it comes to high level parallelization using the island model, as it guarantees that the algorithms really run at the same time. If the algorithms are started as separate Biohadoop instances, it is possible that they run in sequential order, instead of running at the same time.

But Biohadoop doesn't restrict the usage of the island model to algorithms that run in the same JVM. By taking advantage of ZooKeeper [21], running algorithms find each other also across the boundaries of different Biohadoop instances. This can lead to higher scalability, as each instance gets its own resources, but entails the aforementioned problem that YARN doesn't guarantee

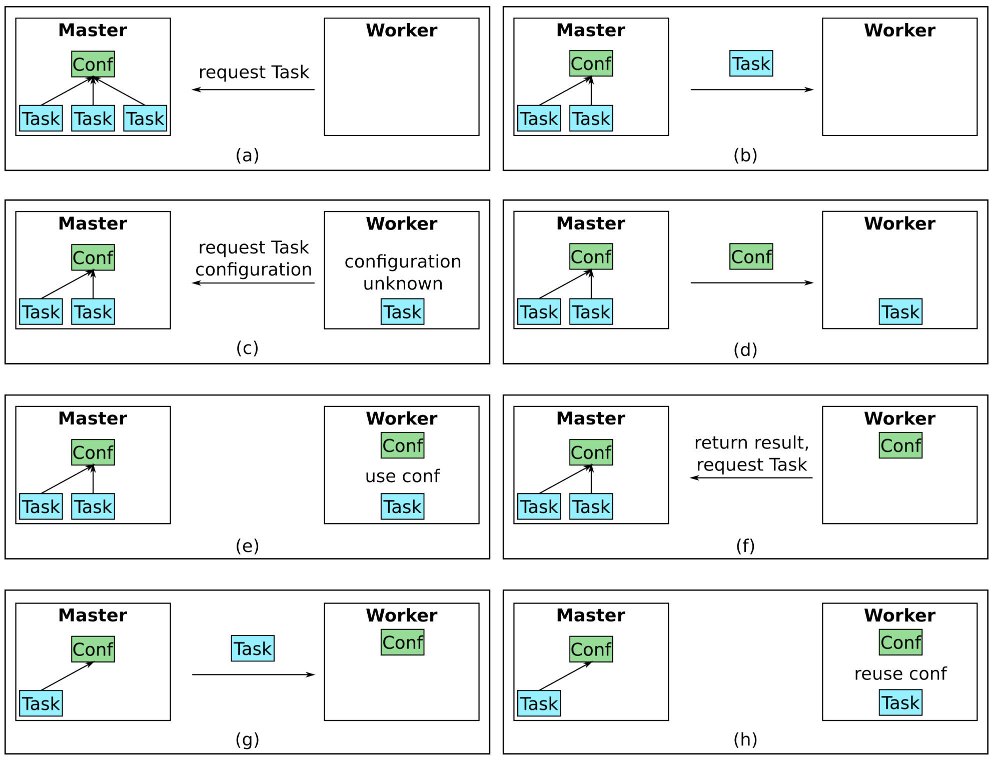

Figure 4.3: A worker requests tasks from the master. The task configuration is send to the worker on demand.

when an application runs, so a trade off has to be made. More information about how to use the island model in Biohadoop can be found in section 4.5.2.

Biooozie (section 4.7) implements a custom action for Apache Oozie and is used to run Biohadoop as part of a larger workflow. The action schedules a single Biohadoop instance or several instances in parallel, depending on its configuration.

The following sections in this chapter continue to describe the details of Biohadoop in more detail, starting with the notion of Algorithm in section 4.2. Section 4.3 talks about the task system and its components, followed by the description of the communication mechanisms in section 4.4. The enhancements section 4.5 provides information about Biohadoops support for persistence and the island model. Section 4.6 explains how to configure Biohadoop. Biooozie is presented in the section 4.7 of this chapter. Information on how to use Biohadoop can be found in chapter 5.

4.2 Algorithm

An algorithm in terms of Biohadoop is the implementation of an abstract problem that should be solved. For example, a genetic algorithm (GA) can be implemented to solve an optimization problem.

Biohadoop supports the programmer with an easy way to parallelize the algorithm by providing the asynchronous task system (see 4.3). The algorithm can submit tasks to the task system and the task system takes care about the distribution and computation of the tasks and promotes the results back to the algorithm. The technical details of the task system are hidden from the algorithm.

Additional mechanisms that are offered to algorithms include persistence and high level parallelization using an island model (see 4.5).

All those capabilities can be used by the algorithm, but Biohadoop does not force their usage. The only thing an algorithm has to do to be run by Biohadoop, is to implement the `Algorithm` interface. This interface defines one method, namely `run`, which is invoked by Biohadoop after the system initialization has completed. The return value of the `run` method is void as there is no return data that could be used in a general way. This may change in future versions.

It is possible to run several algorithms of any kind at the same time in one Biohadoop instance (for example two GA and one PSO), this is just a matter of configuration (see section 4.6).

If there is an error during the execution of an algorithm, the algorithm may throw a `AlgorithmException` at any time. The meaning of a thrown `AlgorithmException` is, that there was an unrecoverable error which prevented the algorithm from progress, but the programmer was aware that such an error could happen, e.g. when a needed configuration argument is missing. Sometimes an algorithm may throw an unchecked exception, like the `NullPointerException`. The difference to the `AlgorithmException` lies in the semantics: unchecked exceptions are considered as the outcome of bugs. At the moment, Biohadoop makes no difference in handling `AlgorithmException` and unchecked exceptions, in both cases, the error is logged and the algorithm is terminated without affecting other running algorithms. But it is possible that this behavior may change in the future. For example, it is thinkable that a custom recovery procedure is invoked in case of an `AlgorithmException`.

4.3 Task system

If a programmer decides to parallelize some parts of an algorithm, it can use Biohadoops task system. The task system takes care of promoting tasks to waiting workers and to return the results to the algorithm, while hiding the details of this process from the algorithm.

The task system consists of a task broker, at least one endpoint and at least one worker. The broker and endpoints are executed on the master, which is the ApplicationMaster in YARN. The workers are executed in additional YARN containers.

An algorithm submits its tasks to the task system, by adding them directly to the broker or by using the `TaskSubmitter`. It is advised to use the the `TaskSubmitter`, as it offers a simple interface for task submission, while the broker offers additional methods that are needed internally by the task system. When an algorithm submits a task, it immediately receives back a `TaskFuture` that works similar to the Java `Future`. The `TaskFuture` represents the result of the task computation. The attempt to read the result of a task computation from a `TaskFuture` has two possible outcomes. In the first case, the result is known and can be read from the `TaskFuture`. In the second case, the result is yet not known (because it still needs to be computed) and the read attempt blocks until the result is available. In addition to the possibly blocking read, the `TaskFuture` provides a non blocking method to check if the `TaskFuture` contains a result.

A queued task is eventually taken out of the broker by an endpoint. Endpoints represent a boundary between the broker on the master side and the workers on the other side. They interact with the broker and communicate with the workers. On worker request, an endpoint takes the next task out of the broker and sends it to the worker. The worker computes the result for the task, using the task data and its related task configuration, and returns the result to an endpoint. The endpoint then returns the result to the task broker, which promotes it back to the algorithm. Figure 4.4 gives a graphical representation of this procedure.

The task system works in an asynchronous manner and doesn't block the algorithm while processing the tasks. However, it provides also mechanisms to block and wait for a result to be computed, if this is preferred.

4.3.1 Task Broker

The task broker is the central feature in the task system, it is used to exchange tasks and their results between the algorithms and the endpoints. The broker combines an internal FIFO queue with additional task bookkeeping (see the concepts below).

All submitted tasks reference a task configuration that is used by the workers to compute the task result. This configuration can be shared by any number of tasks. The configuration reference for a given task is stored in the task

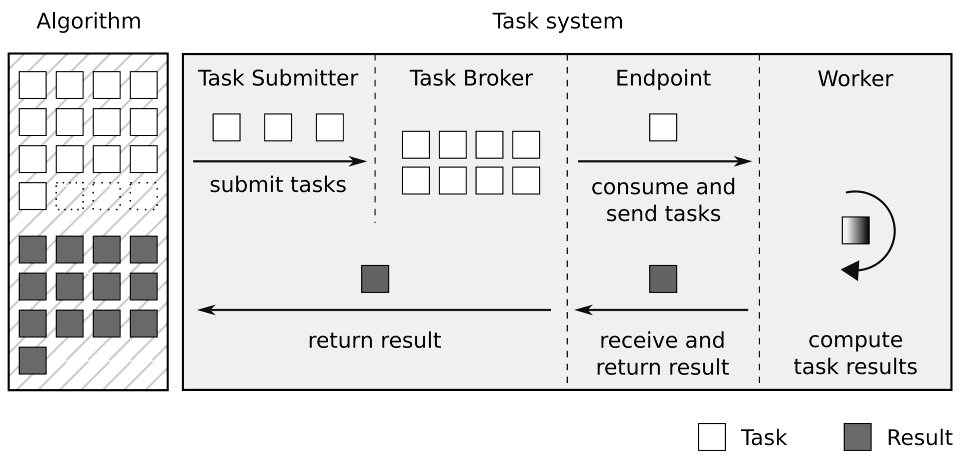

Figure 4.4: Submitting tasks to the task system. The task system consists of a task broker and any number of endpoints and workers. For the sake of simplicity, the figure shows only one endpoint and one worker. In this figure, an optional task submitter is used for task submission.

broker. The configuration itself is send to the workers on demand. A worker requests a specific task configuration, if it encounters a task whose configuration is unknown to the worker. After the configuration was received, it is cached by the worker for later reuse. This is appropriate as usually many tasks use the same configuration. In the GA example, where the workers generate offsprings and compute their fitness values, a single task configuration is enough for all tasks, as offspring generation and fitness evaluation is done in the same way for all tasks. Therefore, a worker needs to obtain the task configuration only once, reducing the communication overhead. It is of course also possible to assign each task an individual task configuration. The result would be, that the worker has to request the configuration for each task individually, the communication overhead would be much higher than in the previous example.

To work properly, the task broker uses two concepts that are needed to perform its work.

The first concept is its internal first-in first-out (FIFO) queue, that stores the submitted tasks. The FIFO queue is thread safe, to allow multiple producers (algorithms) and consumers (endpoints) to interact with the queue at the same time. As an example, lets suppose that two GAs are running in one Biohadoop instance. Both algorithms can submit new tasks, while the endpoints consume tasks from the broker. By using a thread safe FIFO queue, all of this can happen at the same time, without causing problems.

The second concept is a map, that connects submitted tasks to their configuration and their `TaskFuture`. This must be done, because a task loses its references once it is taken out of the FIFO queue and send to a worker. The configuration for the task would become unknown and the result of the task could no longer be associated with the corresponding `TaskFuture`. The mentioned map resolves this problem.

The two concepts are depicted in figure 4.5. In the first step, a task and its configuration is submitted to the broker. The broker inserts the task in its internal FIFO queue and adds the task, the configuration and a new `TaskFuture` to its internal map. The `TaskFuture` is immediately returned to the algorithm as result of the task submission. At this stage, any attempt to access the result of the `TaskFuture` would block, as the result of the task computation is unknown yet. At a certain point, step 2 is performed, where the queued task T_N is consumed by an endpoint and send to a waiting worker. If the worker doesn't know about the referenced configuration TC_N for task T_N, it asks the endpoint for the configuration, which gets the configuration from the broker in step 3. The configuration for the task T_N can only be retrieved, because the broker kept a reference to it in its internal map. In step 4, the worker returns the computed result to the endpoint, which forwards it to the broker. The broker associates the result for task T_N with the according `TaskFuture` TF_N, again using its internal map. After the result for the `TaskFuture` is set, the algorithm can access the result for task T_N without blocking.

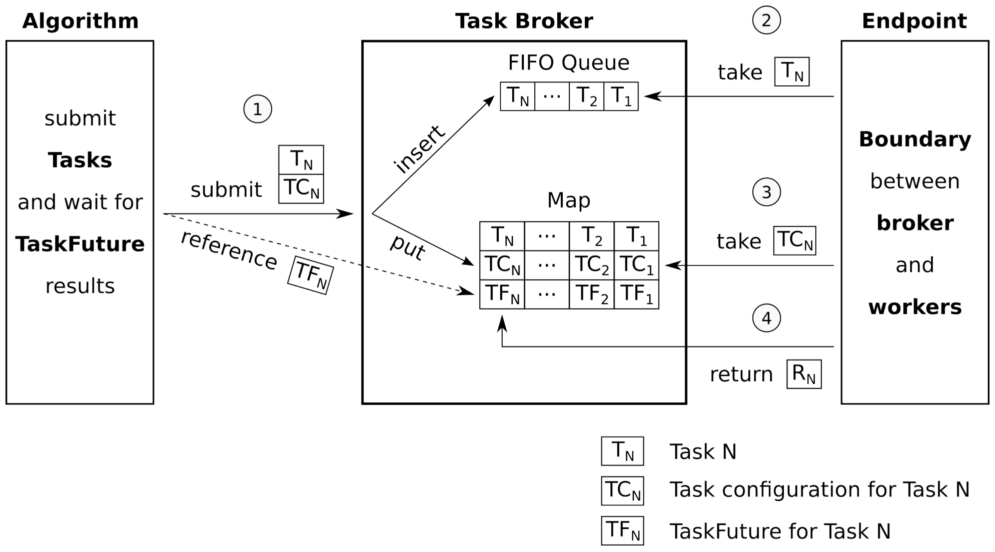

Figure 4.5: Internal structure of the task broker

In addition to task and result exchange, the task broker provides methods to resubmit a task. This is needed in the case of a failure during the computation of the task result.

4.3.2 Endpoint

An endpoint is a boundary between the task broker and the workers. It's main purpose is the communication with the workers. By hiding the technical details of the communication from the algorithm and the task broker, any type of communication facility between the master and the workers can be implemented. This property lead to the implementation of two different communication protocols for Biohadoop. Section 4.4 talks in greater detail about the communication.

Communication is not the only purpose of an endpoint. It interacts also with the task broker, taking tasks and task configurations out of it or returning results that were received from the workers. The attempt of an endpoint to get a task from the broker blocks, if the brokers internal FIFO queue is empty. As soon as new tasks are available, the endpoint continues to work. A blocked endpoint blocks also its waiting workers.

An endpoint is allowed to resubmit a task to the task broker, if there is the need to. For example, lets suppose a task is taken from the task broker and send to a worker. This worker encounters a problem and terminates before returning the result. The endpoint can detect the issue with different methods (e.g. connection closed, heartbeat, time out, etc.) and resubmit the task to the task broker. This way, no task gets lost.

It is possible to run an arbitrary amount of endpoints, but usually this is not necessary. Nevertheless, it can be useful to run different endpoints for different communication protocols (see section 4.4).

The endpoints run in the YARN ApplicationMaster and are started and stopped automatically by Biohadoop. It is configurable which endpoints should be started, section 4.6 gives details about the configuration aspects.

4.3.3 Worker

A worker computes the result for a given task using the configuration that is referenced by the task. The task itself contains the data needed for the computation. The configuration contains information about how to compute the result for the task. The "how" is specified by a Java class that implements the `AsyncComputable` interface. In addition, the task configuration can contain an immutable data set, shared by all tasks that reference the same configuration. The immutable data set is called "initial data".

In the GA example, a task generates and evaluates an offspring. The task data consists of the parent individuals and the task configuration points to a class that implements offspring generation and fitness evaluation. The "initial data" contains static parameters for offspring creation and fitness evaluation. The worker uses the configuration to instantiate the given `AsyncComputable` class. The class and the "initial data" are then used to compute the result for the task, which is a new offspring and its related fitness value. The result is then send to the endpoint.

Tasks are send to the workers without task configuration. The task configuration is delivered only on demand and is afterwards cached by the worker. This can be done, as most tasks will share a common configuration. Sending the configuration only on demand, reduces the amount of transmitted data and increases therefore the performance of the whole system. Figure 4.6 shows how tasks and task configurations are handled by the workers.

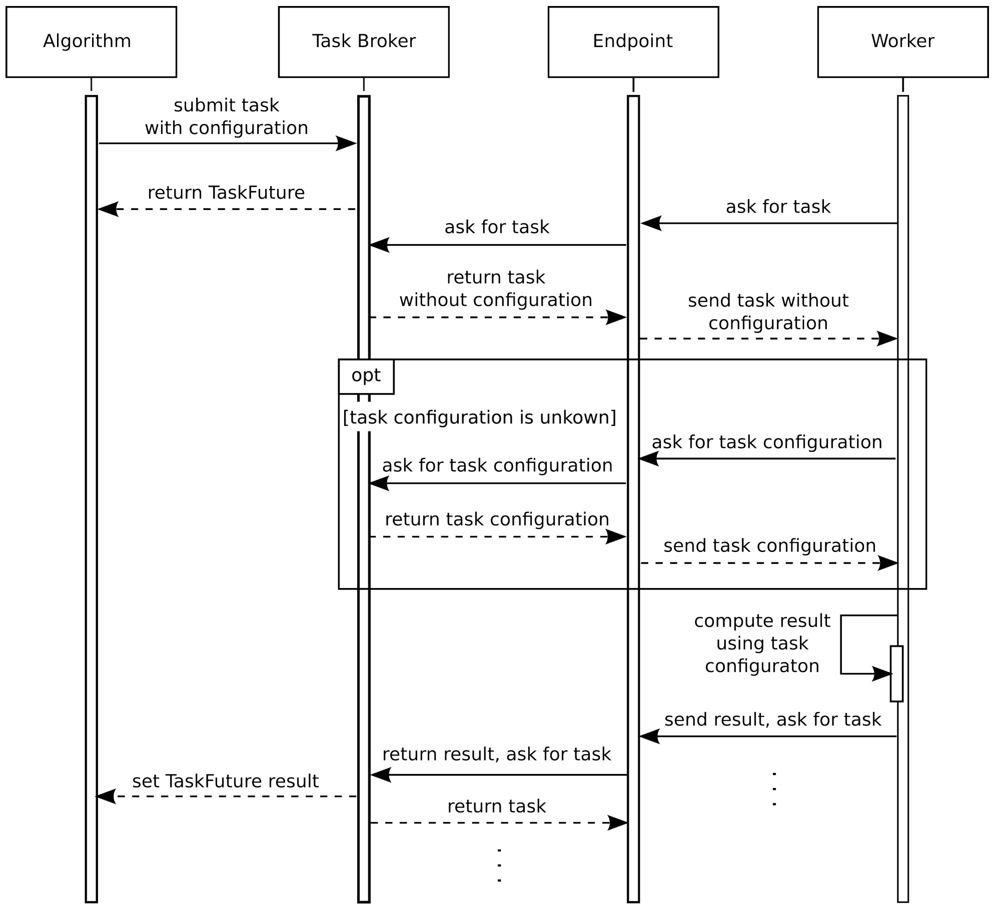
Algorithm
Task Broker
Endpoint
Worker
submit task with configuration
return TaskFuture
ask for task
ask for task
return task without configuration
send task without configuration
opt
[task configuration is unkown]
ask for task configuration
ask for task configuration
return task configuration
send task configuration
compute result using task configuraton
send result, ask for task
return result, ask for task
set TaskFuture result
return task

Figure 4.6: Life cycle of a task, the result is computed by a worker using an AsyncComputable

A worker needs to communicate with an endpoint to get tasks. If the endpoint has currently no work to offer, for example because the running algorithms have not submitted any tasks to the task system, then the worker waits until new work is available. Of course the worker need some resources during the waiting times too, like CPU, RAM and storage or, in YARN terms, containers. One should consider and measure how many workers are really needed.

Biohadoop supports two different kinds of workers: the ones that run under the control of the Apache Hadoop system (from now on called embedded workers) and the ones that run outside of this system (from now on called external workers).

Embedded workers must be configured in Biohadoop, which controls their life cycle. In contrast, external workers can not be configured by Biohadoop, their whole life cycle must be controlled in some other way that is not part of Biohadoop. This can pose some problems, as external workers need to know e.g. when and where Biohadoop is running. The solution to this problems is outside of the scope of this thesis.

If there are no special needs, embedded workers are just fine. There are although some good reasons to use external workers:

- external worker don't necessarily depend on the Hadoop ecosystem, they may run wherever they want, as long as they are able to communicate to at least one endpoint. It is possible to develop external workers that run in completely different environments, for example on mobile phones.
- there is no restriction on the program language for an external worker, as long as it knows how to talk to an endpoint. For example it is possible to implement a worker in JavaScript [22] or Python [23]. In contrast to this, embedded workers have to be written in Java.
- there is no limit to the number of external workers that may run. On the other side, the number of embedded workers is limited by the Hadoop environment on which Biohadoop runs.

Biohadoop provides the possibility to run different endpoints to support external workers. The endpoints may implement arbitrary communication protocols, e.g. WebSockets with JSON serialization for external workers written in JavaScript.

4.4 Communication

The communication between the algorithms, the broker and the endpoints is not difficult, as they run in the YARN ApplicationMaster and therefore in the same JVM. It is just a matter of sharing variables between the different threads, possibly protected by concurrency protocols. For example, the task broker contains a FIFO queue based on the Java `LinkedBlockingQueue`, which is a thread safe queue that supports concurrent writers and readers.

Communicating between endpoints and workers is more complicated, as the endpoints and workers may run in different processes or even on different machines, so variable sharing is not that easy. A more sophisticated communication method must be used.

Biohadoop uses Netty [24] for all communication purposes that span different processes or machines. Netty is a high performance framework for network applications that hides the underlying socket implementation from the user and provides a useful and well tested API to build distributed applications. Netty provides TCP and UDP support, only TCP is used for Biohadoop. All provided endpoints and workers use Netty as their communication base.

The communication protocols on top of Netty can be of arbitrary type. Biohadoop provides two implementations for endpoints and workers that use sockets (hidden by Netty) or the WebSocket protocol. The socket protocol uses Kryo [25] for object serialization, while the WebSocket protocol relies on JSON serialization. The two protocols are discussed in more detail in section 4.4.1.

The reason for the support of different protocols lies in their different use cases. While the performance of sockets with Kryo serialization is higher than for WebSockets, WebSockets have the advantage of great compatibility and broad support in different languages. This is important for external workers as they don't have to be implemented in Java.

On top of the communication protocols, Biohadoop establishes its own application protocol for task, configuration and result exchange between endpoints and workers. This protocol defines a communication flow further described in section 4.4.2. Figure 4.7 shows the different layers that Biohadoop uses for data exchange.

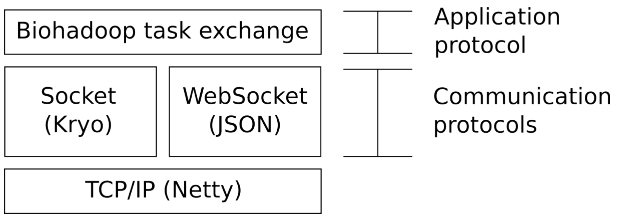

Figure 4.7: Biohadoops communication layers

Biohadoop enables the use of custom protocols, by implementing the appropriate parts of `Endpoint` and `Worker` interfaces. Corresponding endpoints and workers have to agree about the communication and application protocol.

4.4.1 Protocols

In this section, the two provided communication protocols of Biohadoop are described, each one has its advantages and disadvantages.

Sockets

The socket communication between endpoints and workers is implemented on top of Netty that provides an abstraction of the underlying raw TCP/IP socket. The advantage of using Netty over raw sockets is, that Netty provides a simple to use API for non-blocking asynchronous communication. A manual implementation would have been more error prone.

Kryo [25] is used for object serialization. It is a library for high speed serialization of Java objects and usually faster than the build-in serialization features of Java [26].

A disadvantage of Kryo is, that its object serialization is not a standard and therefore in broad usage. This restricts the worker implementations to be written in Java, which may not be a problem at all, especially if only embedded workers are used. But if external workers should be used, they have to be written in Java, or need to provide a custom Kryo implementation.

If the data exchanged between endpoints and workers is huge, the Kryo buffer sizes must be increased. This can be done by setting the according configuration options in the configuration file (see section 4.6).

WebSocket

The WebSocket implementation also uses Netty to hide the underlying TCP/IP socket. In contrast to the socket communication, it adds the WebSocket protocol on top of it. JSON is used for object serialization.

WebSockets are usually used for the communication between a web application and its application server. They rely on the HTTP protocol for the handshake, during which the communication partners agree to upgrade to the WebSocket protocol. After the upgrade is done, the communication between an endpoint and a worker can be performed using binary or text streams.

WebSockets have a very small overhead when data is exchanged, for example 2 byte for text stream messages. This is a major difference to the HTTP protocol where the HTTP headers are sent on each request and response. It improves the communication performance, especially for the transmission of small amounts of data. Another difference between WebSockets and HTTP is, that the WebSocket communication can be performed in full duplex, HTTP needs a request - response cycle. This can further improve the performance, but has no impact for Biohadoop, as the communication between endpoints and workers is performed in a request - response manner (see section 4.4.2 for more information).

The biggest advantage of WebSockets and JSON lies in their standardization. This is important in combination with external workers, as those workers can be written in any language. Most languages have support for WebSockets and JSON - because they are standards. The biggest disadvantage of WebSockets with JSON is, that they are slower than sockets with Kryo serialization.

4.4.2 Communication flow

The communication protocols presented in the prior section are used as a base to the application protocol, that implements a well defined communication flow between endpoints and workers. This communication flow, depicted in figure 4.8, is the same for both provided communication protocols. Custom protocols don't need to implement this flow, they are free to implement their own communication pattern.

As one can see in figure 4.8, the communication between endpoints and workers is initialized by the worker. After the worker gets a task from an endpoint, it looks if the needed task configuration is already stored in its internal cache. If the configuration is unknown, the worker requests it from the endpoint. The endpoint responds with the configuration, which is cached by the worker, in case that it is needed again. The worker computes the result for the task using the task data, and the corresponding configuration. The result is then transmitted to the endpoint and a new task is requested.

4.5 Enhancements

Biohadoop provides two enhancements for algorithms, beside the task system presented in section 4.3. The first one is for persistence, where it is possible to load and store arbitrary data to and from a file system (see section 4.5.1). The second enhancement provides high level parallelism between parallel running algorithms, called the "island model" (see section 4.5.2).

4.5.1 Persistence

There are a lot of reasons to store algorithm results to a file system. The most important is to save the final result of a computation. Another reason is to

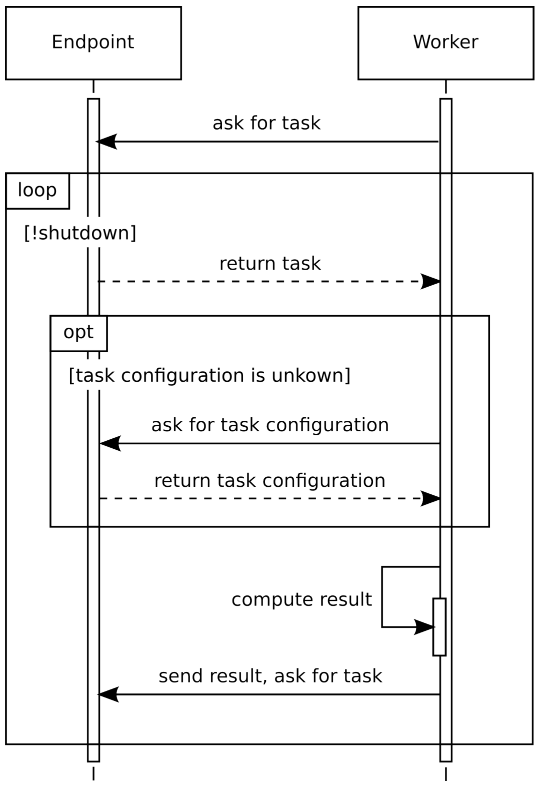
Endpoint
Worker
ask for task
loop
[!shutdown]
return task
opt
[task configuration is unkown]
ask for task configuration
return task configuration
compute result
send result, ask for task

Figure 4.8: Communication flow between endpoints and workers

store intermediate results in case something happens. If this data is reloaded afterwards, the computation can continue from that point on. Intermediate results can also be used for other computations or for visualization.

The conclusion is that some kind of persistence is useful. It should include both the saving and loading of data. Biohadoop provides this kind of service by offering a simple API, accessible through the class `FileUtils`. The API stores provided data in JSON format in a file with a given name. When a file is loaded, the API supposes that the contained data is also in JSON format and tries to deserialize it. An exception is thrown if this is not possible.

The API covers the fundamental persistence use cases, but a programmer is free to use its own mechanism of data storage and retrieval.

4.5.2 The island model

The island model is a high level parallelization model that is sometimes used in optimization problems. In the island model, several algorithms run in parallel, trying to compute the result to the same problem. The parallel running algorithms are called the islands. Each of these algorithms is independent of the others, and each one may have a different solution at a given point of time. By exchanging their data after some intervals, islands may get interesting solutions from other islands that can be integrated in their own computation to enhance their solution. If we take the GA as an example, the islands would consist of independently running GAs that exchange individuals to improve the solution. Figure 4.9 shows an example island model with 3 GAs that exchange individuals.

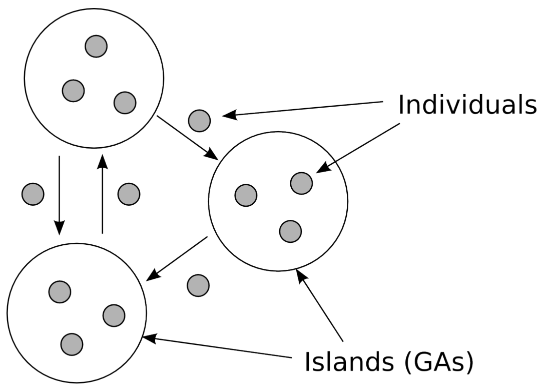

Figure 4.9: Example island model with 3 GAs, exchanging individuals

The island model enhances the exploratory behavior of optimization algorithms, which often results in better overall solutions. As the data exchange is only done at certain points of time, the islands have the chance to exploit their own solution. When they get stuck in a local optima, they get the chance to escape this optima by considering solutions from other islands.

Biohadoop provides an API, implemented in the class `IslandModel`, that can be used to build an island model between any number of algorithms. It provides methods to publish the own solutions to other islands or to merge remote solutions with the own solutions. Data merging can be configured by implementing the interfaces `RemoteResultGetter` and `DataMerger`. The `RemoteResultGetter` defines from which remote island the data should be retrieved. It may take into account different properties, like the number of iterations, the fitness of individuals, etc. The `DataMerger` defines, how two solutions should be merged.

The island model API uses ZooKeeper [21], which is a server that provides distributed configuration and synchronization services and a naming registry. Therefore, a running ZooKeeper instance must be accessible by Biohadoop, if an algorithm wants to use the island model.

By using ZooKeeper as central registry for the island model, it doesn't matter if the algorithms run in the same Biohadoop instance or in different Biohadoop instances. They find each other through their ZooKeeper registrations.

The main advantage of running several algorithms in the same Biohadoop instance is, that it guarantees that they all run at the same time. Scheduling several Biohadoop instances in parallel doesn't guarantee that they also run at the same time, as YARN decides when to launch an application. The island model is useless if the algorithms don't run at the same time.

4.6 Configuration

Biohadoop uses a configuration file in JSON format. The advantage of JSON is, that its understandable for humans and usually smaller in size than e.g. XML.

The path to the configuration file must be given on invocation of Biohadoop as its first parameter (more information on how to run Biohadoop can be found in 5.1). Biohadoop stops immediately with an exception if the path is empty or wrong or the configuration file can not be parsed.

The configuration file itself consists of the following four top-level objects:

- `communicationConfiguration`: defines a list of endpoints and a list of workers that Biohadoop should start. The worker configuration provides additional information about the number of workers that should be started

- `globalProperties`: a map with strings as keys and values. This properties are used for global settings that should be available in the ApplicationMaster. Examples for such global settings are configurations for Kryo and ZooKeeper.

- `includePaths`: a list of strings that define the paths where needed libraries can be found. This isn't important for a local running instance of Biohadoop (e.g. during development), as the necessary classpaths must be set when starting Biohadoop. But it is important when Biohadoop runs in the Hadoop environment, as this are the paths to libraries that Hadoop should provide to Biohadoop when running. If the paths to the necessary libraries are wrong when running on Hadoop, Biohadoop won't run correctly.

- `algorithmConfigurations`: a list of algorithms that should be run by Biohadoop. Each element in the list describes the configuration for an algorithm. The configured algorithms are started in parallel in the same Biohadoop instance.

It is not always convenient to write a configuration by hand, although it is possible. Biohadoop provides two builder classes to make it more easy to produce configuration files. The builder in `BiohadoopConfiguration` offers methods to configure the top level elements of a configuration file. Algorithm configuration is simplified by the builder in `AlgorithmConfiguration`. The result of this algorithm configuration can then be handed over to the `BiohadoopConfiguration` builder.

4.7 Biooozie

Biooozie implements a custom action for Apache Oozie (see 3.3) that schedules one or several instances of Biohadoop. The action can be part of a workflow of arbitrary size. The outcome of the custom action is "ok" if no error happened during the execution of the action. This is also true for the case that several Biohadoop instances are defined in one action. If any Biohadoop instance fails, the outcome of the action is "error".

A short example workflow with three stages copies data sets to a HDFS file system, on which some MapReduce action is performed that produces new data sets. Those data sets in turn are the base for a GA computation, performed by Biohadoop (see figure 4.10).

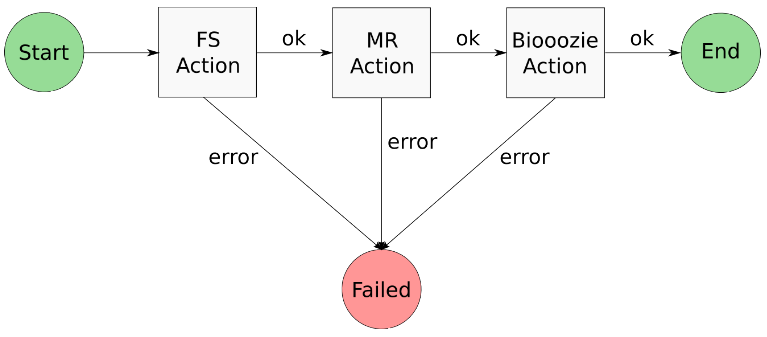

Figure 4.10: Example Oozie workflow, including Biooozie action

The action is configured in an Oozie workflow as an XML element with name `biohadoop`. It contains one `name-node` element that defines the URL of the HDFS NameNode, and one or several `config-file` elements. Each `config-file` element represents a Biohadoop instance that starts with the given configuration file. This way it is possible to schedule several Biohadoop instances in parallel, using a single action. The parallel instances are for example useful for running an island model.

The term “schedule” is used here on purpose, because Hadoop doesn’t guarantee that the instances also run in parallel, this depends on the available Hadoop resources (i.e. on the available YARN containers). It is for example possible that a custom action schedules three instances of Biohadoop, but due to a lack of Hadoop resources, the instances run one after another.

Biohadoop could also be invoked using Oozies `java` action, the advantage of Biooozie is that its tailored to Biohadoop and there is no need to provide all the parameters that a simple `java` action needs.

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 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 4.6 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 [27]. 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 [9]. 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 **??**. 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 simple method can be found, to use a pre-build Hadoop environment. The only dependency this environment has, is Docker [28], which is a simple and lightweight runtime for containers, available on all major operating systems.

5.1.1 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 4.6 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.

5.1.2 Hadoop environment

To start Biohadoop in a Hadoop environment (for example in the one provided in appendix **??**), 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:

```
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

Evaluation

Biohadoops purpose is to facilitate the implementation of parallel algorithms on Hadoop. It is expected that the execution time of an algorithm reduces if it is parallelized (assuming the algorithm is suitable for parallelization). As this cannot be guaranteed without proper measurement, this chapter is devoted to the study of Biohadoops speedup characteristics.

Two bio-inspired optimization algorithms are used as benchmarks. Both use Biohadoop and its task system to solve a test problem. The algorithms and test problems are described in section 6.1.

The algorithms are executed on a Hadoop cluster to study how their execution times change when their problem sizes and the number of workers change. The benchmark details can be found in section 6.2, the results are presented in section 6.2.

6.1 Test problems

Both implemented algorithms are part of the GA family. They differ in the number of objectives that they can handle. While NSGA-II is used to solve the MOP in section 6.1.1, a simple GA is used to solve the SOP in section 6.1.2.

6.1.1 ZDT-3

The first implemented optimization algorithm is NSGA-II that is used to find optimal solutions for the Zitzler–Deb–Thiele's function nr. 3 [29]. ZDT-3 is a well known MOP and therefore suited as a test problem. The task is to find an approximation to the optimal Pareto Front, given in figure 6.1.

The implementation uses Biohadoop workers to create and evaluate the offsprings. Simulated Binary Crossover (SBX) and Parameter based mutation [30] are used for the offspring creation. The fitness is computed using the ZDT-3 function. The selection of the fittest individuals for the next population is based on ranking and crowding distance and is performed on the master.

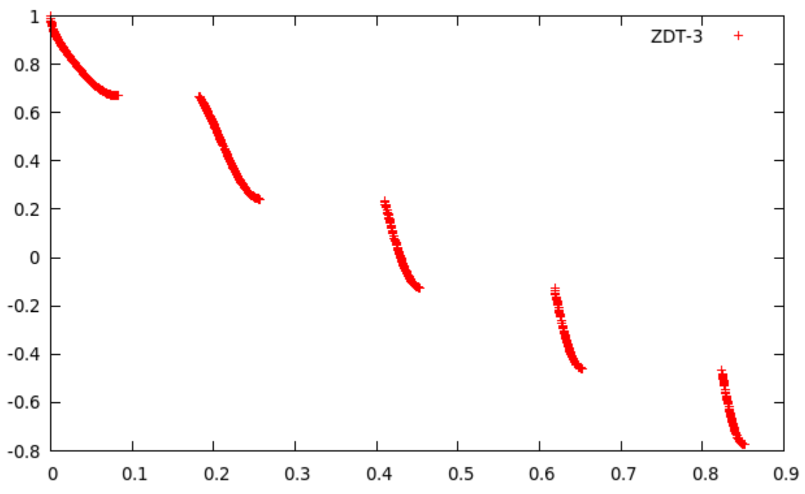

Figure 6.1: Optimal Pareto Front for ZDT-3

6.1.2 Tiled matrix multiplication

The second benchmark implements a GA to solve the SOP of finding optimal tile sizes for the tiled matrix multiplication. The objective is to minimize the execution time for a matrix multiplication.

A matrix multiplication can be performed in different ways. The most obvious one is the standard algorithm:

```
for i = 1 to n
  for j = 1 to m
    for k = 1 to l
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

The matrix multiplication can be improved by loop tiling [31]. The computation is performed on smaller blocks (tiles) of the matrices:

```
for i0 = 1 to n, step blocksize_i
  for j0 = 1 to m, step blocksize_j
    for k0 = 1 to l, step blocksize_k
      for i = i0 to min(i0 + blocksize_i, n)
        for j = j0 to min(j0 + blocksize_j, m)
          for k = k0 to min(k0 + blocksize_k, l)
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

If the blocks are small enough they fit into the L1 CPU cache which results in a speedup. For example, the average of five consequent test multiplications for two matrices of size 1024x1024 took 18.245 seconds for the simple matrix multiplication and 2.048 seconds for the tiled multiplication with tile sizes $i = 16$,

$j = 16$ and $k = 16$. This numbers show that it is appropriate to use the tiled approach for the matrix multiplication. But the speed of the tiled matrix multiplication depends heavily on the tile sizes, the same tiled multiplication as above with tile sizes of $i = 1$, $j = 1$ and $k = 1$ took 22.690 seconds to finish, an increase of more than 10 times compared to a good tile size. Because of the number of possible tile size combinations and the time it takes to execute a matrix multiplication (e.g. matrix size=1024, 2 seconds for a matrix multiplication: $1024^3 * 2 = 2e9$ seconds), it is not feasible to do an exhaustive search for the optimal tile sizes.

An optimization algorithm can be used to find the (near) optimal tile sizes for the different loops. In this case, the optimization is done using a GA. The implementation uses Biohadoops workers to create and evaluate an offspring. For the offspring creation, Simulated Binary Crossover (SBX) and Parameter based mutation are used. The fitness is computed as the time it takes to multiply two matrices using a given tile size. The selection of the fittest individuals for the next population is performed on the master.

6.2 Benchmarks

The task of the benchmarks is to find the speedup characteristics of Biohadoop. The assumption is that the execution time of an algorithm depends on the problem size and the number of workers. To evaluate this assumption, the algorithms presented in section 6.1 are executed with different problem sizes and different numbers of workers. The execution times are measured and used to calculate the speedup using the formula $S = T_S/T_P$, where S is the speedup, T_S is the time for a serial execution and T_P is the time for parallel execution. The results are discussed in section 6.3.

A benchmark is defined by a given algorithm (e.g. NSGA-II) and its settings (e.g. number of workers, number of iterations, etc.). All performed benchmarks have the following settings in common:

- the number of iterations is set to 250
- the population size is set to 100
- the distribution index n_c for the SBX crossover is set to 20
- the distribution index n_m for the mutation is set to 20
- the mutation probability for each offspring value is set to $1/n$, i.e. on average one offspring value is mutated

The test problems have also exclusive settings that only apply to them.

For ZDT-3 this is the genome size. The genome size corresponds to the number of values of an individual and the dimension of the solution space. Each individual is represented by its genome. ZDT-3 can handle any genome size, changing this number influences two properties of the ZDT-3 benchmark. First, increasing the number of genomes also increases the computation effort for the workers that generate new offsprings and compute their fitness. This results from the fact that the workers generate new individuals using the parent genomes and that the ZDT-3 algorithm, used for the fitness computation, loops over all genomes. Second, the genome size influences the amount of data that has to be transferred between the master and the workers. Each worker repeatedly receives two parent individuals and returns an offspring and its computed fitness. The amount of data send between master and workers is therefore related to the gnome size of each individual.

The exclusive setting of the tiled matrix multiplication is the matrix size. It influences the number of computations that need to be performed for a full matrix multiplication and therefore the execution time. In contrast to the first problem, the matrix size has no influence on the amount of data that has to be transferred between the master and the workers. The matrices are part of the "initial data" (see chapter 4.3.3) and therefore transferred exactly once to every worker. The task data consists of two parent individuals that are transferred from the master to the workers to create a new offspring and compute its fitness. The data transferred from a worker to the master contains the offspring and its computed fitness value. Each individual consists of its tile sizes for i, j and k.

The genome size for different ZDT-3 benchmarks is set to 10, 100, 1000 and 10000. The matrix size for the tiled matrix multiplication is set to 128x128 and 256x256. The execution time for each setting is measured for a number of workers that range from 1 to 15. Each of this benchmarks is repeated five times to improve the reliability of the results, making it 300 benchmark runs for ZDT-3 (4 genome sizes * 15 worker setting * 5 repetitions) and 150 benchmark runs for the tiled matrix multiplication (2 tile sizes * 15 worker settings * 5 repetitions).

All experiments were performed on a Hadoop cluster with 6 identical machines. Each machine has the following specifications:

- Intel Core2 Duo CPU E8200 @ 2.66GHz (2x2,66Ghz, no hyperthreading)
- 6MB shared L2 cache, 32KB L1 data cache, 32KB L1 instruction cache
- 4GB (2x2GB) DDR2 RAM @ 667MHz

The machines are directly connected to the same Switch through a 1Gb (Gigabit) Ethernet network.

6.3 Results

The time a Biohadoop application takes to execute is composed of the time Biohadoop needs to start up and the algorithm execution time. The start up begins with Biohadoops submission to Hadoop and ends when the algorithms `run` method is invoked. The algorithm execution time starts with the invocation of the algorithms `run` method and ends when this method returns.

The distinction between start up time and algorithm execution time is made because the main part of the start up time is spent between the application submission to YARN and the beginning of its execution. It is not possible to predict when an application is executed by Hadoop, it depends on different factors like the available cluster resources. To minimize the impact of this uncertainty, the following measurements are based on the algorithm execution time, without the application start up time. The start up time over all benchmarks range from 2.378s to 6.147s, with a median of 3.864s, a 25% quartile of 3.145s and a 75% quartile of 4.258s. The mean was 3.781s. This start up times are close to each other, because the used cluster was completely dedicated to the benchmarks. The start up may take longer when the cluster usage is higher.

Table 6.1 gives an impression how well the benchmark problems are suited to parallelization by showing the maximum theoretical speedup. The theoretical speedup was calculated using the formula $S = T/(T - t_p)$, where S is the speedup, T the algorithm execution time and t_p is the time spent in code parts that are parallelized using Biohadoops task system. T and t_p were taken from the average benchmark times with one worker.

Test problem	Theoretical speedup
NSGA-II, 10 genomes	7.028
NSGA-II, 100 genomes	7.600
NSGA-II, 1000 genomes	12.008
NSGA-II, 10000 genomes	11.124
128x128 tiled mul	81.359
256x256 tiled mul	212.169

Table 6.1: Theoretical speedups

The algorithm execution time results for the benchmarks can be found in the boxplots in figures 6.2, 6.3, 6.4 and 6.5 for the ZDT-3 benchmarks and figures

6.6 and 6.7 for the tiled matrix multiplication. The number of workers is plotted on the x-axis, the algorithm execution time is plotted on the y-axis.

6.3.1 Influence of YARN container placement

The first thing to note when looking at the figures is that the five benchmark times for a given setting (e.g. NSGA-II, 10 genomes) and one worker are very different. The explanation for this effect can be found in the YARN container placement. If a worker container is executed on the same machine as the master container, they communicate without using the physical network. The result is a huge performance gain, as can be seen for example in figure 6.3. In the single worker benchmarks, 4 out of 5 benchmarks executed with both the master and worker container running on the same machine, resulting in a 50% better performance (9,761s average) compared to the fifth benchmark (14.164s) where the master and worker were executed on different machines.

The number of worker containers running on the same machine as the master can also have a negative effect on the execution times. This is especially true if the master is already at the limit of the machines resources and must share them with the workers. An example for this can be found in figure 6.2 for 8 workers. Two worker containers were executed on the same machine as the master during 2 out of 5 benchmarks. The execution time results were 87.977s and 88.014s. In the remaining 3 benchmarks, only one worker container was executed on the same machine as the master, leaving more resources to the master. This results in execution times of 68.423s on average, a difference of more than 20%.

So it depends on the available resources of a machine if the execution of worker containers on the same machine as the master container provides benefits or drawbacks. If a resource like CPU or network is already at its limit, additional worker containers slow the whole Biohadoop execution down. If there are enough resources available, the execution of worker containers on the same machine as the master provides benefits, as the communication between the master and the workers can be performed without network usage.

The location of the YARN containers can currently not be influenced, but discussions by the YARN developers suggest that future versions of YARN will support this feature.

6.3.2 ZDT-3

The next thing to notice are the speedups for the ZDT-3 benchmarks. ZDT-3 is not well suited for parallelization as can be seen from the theoretical speedups in table 6.1, but the results are even worse than expected, with maximum speedups of 1.619 for 10 genomes, 1.513 for 100 genomes, 2.498 for 1000 and 2.479 for

10000 genomes. Figure 6.8 shows the speedup results of the ZDT-3 benchmarks together with the speedups for the tiled matrix multiplication.

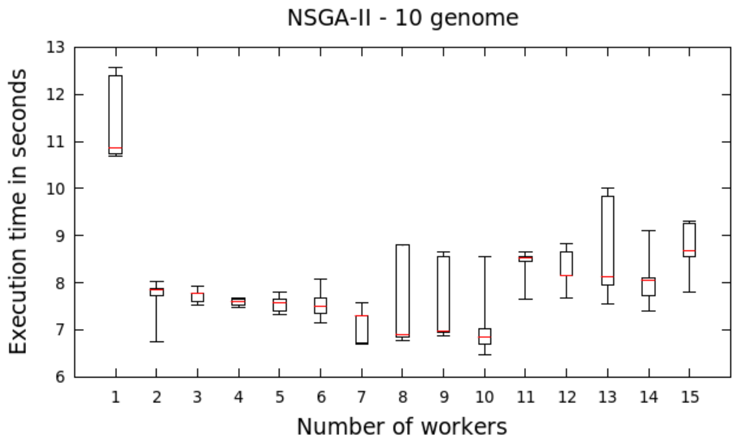

Figure 6.2: ZDT-3 execution times for a genome size of 10

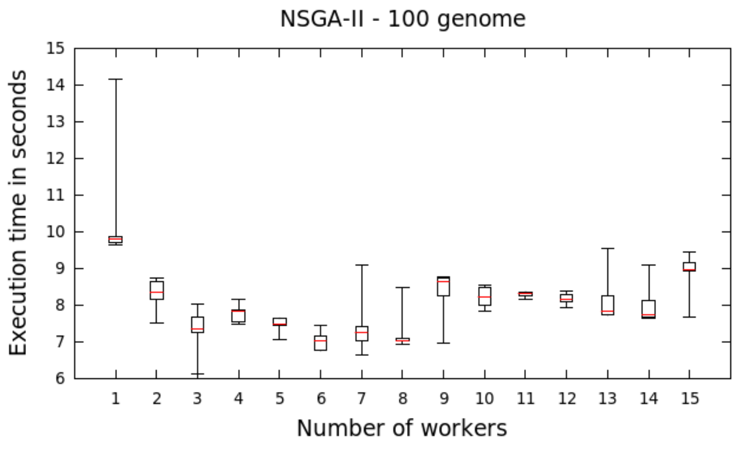

Figure 6.3: ZDT-3 execution times for a genome size of 100

The ZDT-3 benchmarks seem to suffer from the lack of one or more resources (bound by the resources), which prohibits further speedup increases. The investigations show that the ZDT-3 benchmarks are not bound by memory, i.e. memory issues don't slow the execution down. All benchmarks start with 256MB of Java heap memory, which is enough for the containers to execute without causing excessive garbage collections. This was established by using the tool jvisualvm (delivered with Java) for the ZDT-3 benchmark with 10000 genomes. The memory usage for the master container is at about 100MB to 150MB, the

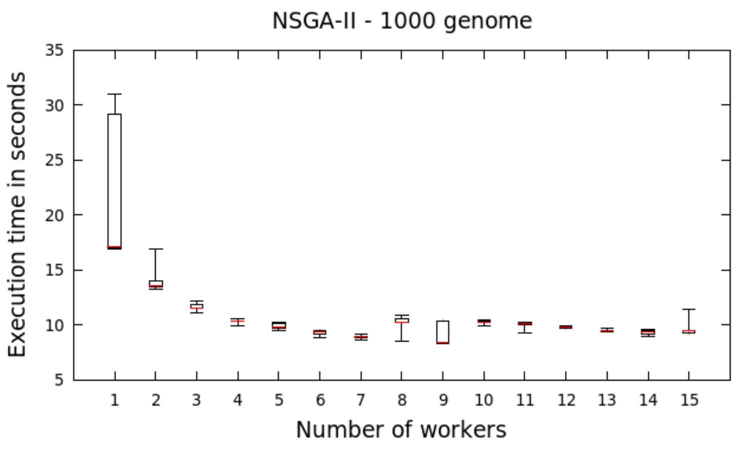

Figure 6.4: ZDT-3 execution times for a genome size of 1000

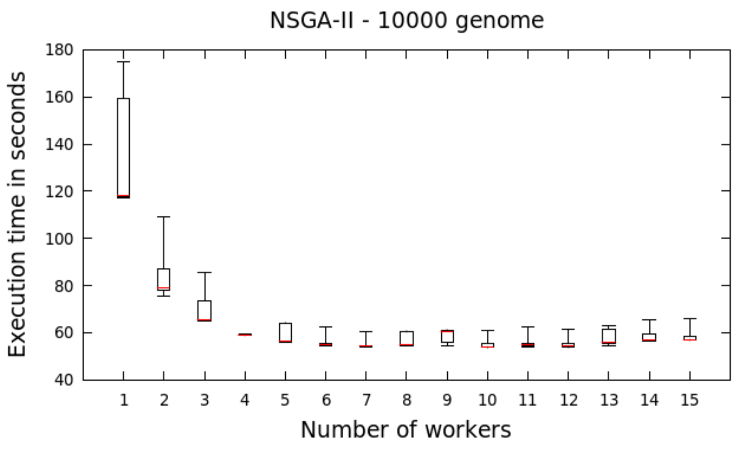

Figure 6.5: ZDT-3 execution times for a genome size of 10000

GC activity, a good indicator for memory problems, ranges from 3% to 5.5% of the CPU time, with an average of 3.8%. The memory usage for a worker is even lower and lies in the range of 5MB to 30MB. This numbers show no significant memory problems.

The next step is to investigate the network performance. Calculations give a first hint to understand if the bad speedups can be explained with the saturation of the 1Gb network (a small "b" denotes bits, a big "B" denotes bytes, e.g. 1Gb = 1 gigabit, 1GB = 1 gigabyte). In each benchmark, 250 iterations on 100 individuals are performed, resulting in 25000 tasks. The tasks are send from the master to the workers and the workers return the results. The task data send from the master to the worker contains two individuals. Each individual

genomes	data (Mb)	theoretical transfer time (s)	fastest algorithm execution time (s)
10	32	0.032	7.072
100	320	0.32	7.031
1000	3200	3.2	8.910
10000	32000	32	55.475

Table 6.2: Amount of network data sent from master to workers, theoretical transfer time and fastest algorithm execution

consists of its genome, where each value in the genome is of type `double` (8 bytes). For a genome size of 10, this makes 2 (parents) * 10 (genomes) * 8 (bytes) * 25000 (tasks) = 4000000 bytes (4MB) or 32Mb of data that needs to be transferred from the master to the workers during the benchmark. The size of the results send from the workers to the master is about the half, as it consists of an individual (the offspring) and its fitness (the fitness is composed of two `double` values). This fact allows to use the outgoing data amount as upper bound for the network usage: if the outgoing data rate doesn't exceed the network bandwidth, this will be true also for the incoming data. Table 6.2 shows the results for all genome sizes together with the best algorithm execution times. One can see from the table that the benchmark data can be transferred on the 1Gb Ethernet network during the according fastest algorithm execution time.

Additional experiments were performed to improve the confidence in the calculations and to establish the true achievable data rate for the network, given different message sizes. The experiments measure the peak network bandwidth using a small Java program and iftop [32]. The Java program uses the same communication techniques as Biohadoop (Netty + Kryo) and performs repeated request / response cycles between a master and several workers. The exchanged messages consist of 20, 200, 2000 or 20000 `double` values, corresponding to two parent individuals in the according ZDT-3 benchmarks. The resulting peak bandwidth was 134Mb/s for 20, 489Mb/s for 200, 901Mb/s for 2000 and 552Mb/s for 20000 `double` values. The CPU on the master was the limiting factor for 20, 200 and 20000 `double` values. For 2000 `double` values, the network was saturated at 901Mb/s and therefore the limiting factor. No studies were performed to explain why the experiments delivered the best results with 2000 values as this lies out of the scope of this thesis.

One phenomena regarding the network bandwidth needs further investigation. The above measurements show a peak data rate of 552Mb/s for the case of 20000 `double` values. If this data rate is taken as base for the ZDT-3 benchmark with

10000 genomes, one can calculate that more than 55s are needed to exchange 32Gb of data over the network between the master and its workers (32000Mb / 552Mb/s = 57.97s). The explanation can be found once more in the YARN container placement. The 552Mb/s peak bandwidth is the data rate that is send through the Ethernet port to the cluster, but worker containers that run on the same machine as the master don't use this port for communication. Instead, they communicate through the local interface. iftop showed an additional combined data transfer rate of 400Mb/s (send and receive data rates are added) on the local interface when workers were running on the same machine as the master. This gives an aggregated peak data rate of 700Mb/s - 800Mb/s for the outgoing traffic, which is fast enough to transmit 32Gb of data in less than 55s. The execution times were higher in cases where no workers executed on the same machine as the master.

The calculations and additional experiments show that the network is fast enough to transfer the ZDT-3 benchmark data. The reason for the bad ZDT-3 speedup results lie elsewhere.

This leads to the assumption that the benchmarks are CPU bound which was confirmed through observations of the CPU usage of the master. In the case of 10 and 100 genomes the CPU limit was reached by the master with two workers, for 1000 and 10000 genomes the limit was reached with four workers.

The high CPU utilization is caused by two effects: the first one is the object serialization/deserialization overhead that ranges between 30% to 40% for genome sizes of 10 and goes up to 60% to 70% for a genome size of 10000. Small genome sizes mean a high rate of exchanged messages and serializations/deserializations. Large genome sizes reduce the rate of exchanged messages but increase the amount of work for a single serialization/deserialization.

The second effect is a direct consequence of computational small worker tasks like in the case of 10 to 100 genomes: the master performs (beside the communication aspects) the algorithms for ranking and crowding distance. The workers return their results fast as the computation is not intense. The master has to compute therefore the ranking and crowding distance at short intervals. This results in a CPU utilization of about 25% to 30% only for this computations.

In conclusion, the ZDT-3 benchmarks are CPU bound by the master due to the small computational effort on the workers and the resulting fast exchange of many small messages. Increased genome sizes provide better speedup results, but are again limited by the CPU of the master, as they have higher demands for object serialization/deserialization. The performance of the 1Gb network and the available memory are sufficient to not slow down the ZDT-3 benchmarks.

6.3.3 Tiled matrix multiplication

The theoretical speedups for the tiled matrix multiplication promise better results (see table 6.1) as matrix multiplications are compute intense and clearly dominate the algorithm execution time. Figure 6.6 and 6.7 show the execution times. One can see that the execution times decrease with the number of workers. This scales until 12 workers, after which the execution times remain the same or increase slightly. The reason for this is that the cluster offers 12 CPU cores in total. When all cores are fully utilized, which happens with 12 workers, additional workers have to share CPU resources. This negatively impacts the execution times. So the tiled matrix multiplication is CPU bound by the workers.

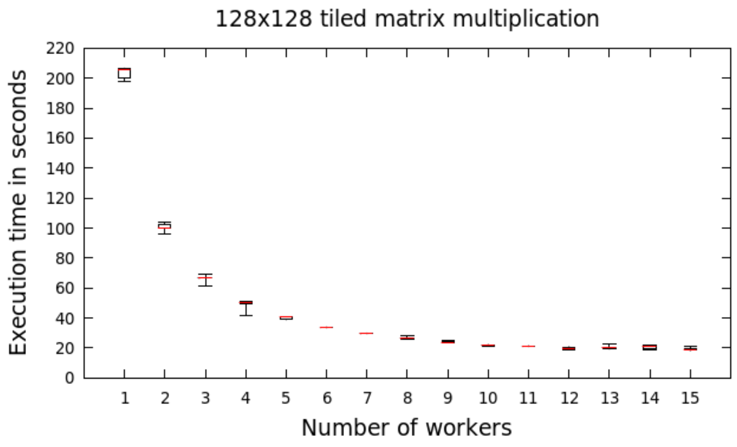

Figure 6.6: Tiled matrix multiplication execution times for a matrix size of 128x128

An additional advantage of the tiled matrix multiplication benchmarks over the ZDT-3 benchmarks is the small amount of data that needs to be transmitted. Like in the ZDT-3 benchmarks, each task data consists of two parents that are sent from the master to the worker, the result is an offspring with its fitness value. In contrast to ZDT-3, where an individual consists of a number of `double` values according to its genome size, a tiled matrix multiplication individual consists of the tile sizes for the i, j and k loop. Each of them is a single `integer` with 4 bytes. The total amount of data that needs to be transmitted from the master to the workers is therefore $2 * 3 * 4 * 25000 = 600000$ bytes or 4.8Mb. Together with the computational intense tasks of matrix multiplication on the workers (leads to lower network usage) and the absence of time consuming ranking and crowding distance algorithms on the master, this provides speedups of up to 10.507 for 128x128 matrices and 7.961 for 256x256 matrices.

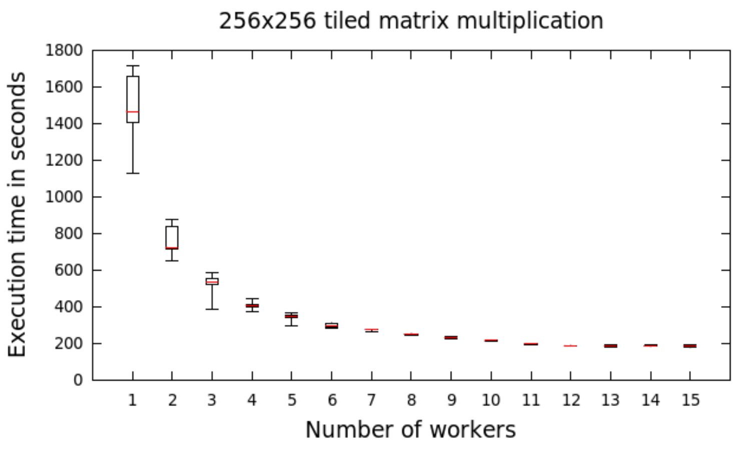

Figure 6.7: Tiled matrix multiplication execution times for a matrix size of 256x256

The reason for the better performance of the 128x128 benchmark over the 256x256 benchmark is unknown. A possible explanation is that the tile sizes are taken from a bigger range (256 instead if 128) which makes it more likely that bad tile sizes are chosen. This is although pure speculation.

6.3.4 Speedups

Figure 6.8 depicts the speedups for all test problems with respect to increasing worker sizes. The ZDT-3 benchmarks show poor results. This is not surprising as the maximum theoretical speedups of this problems are small (see table 6.1) and the communication overhead is bigger compared to the tiled matrix multiplication. The only unexpected outcome was that the benchmarks scale very bad with a maximum speedup of 2.498 for 1000 genomes. The reason is that the ZDT-3 benchmarks are CPU bound by the master, as the investigations in section 6.3.2 show.

The tiled matrix multiplications demonstrate better results, the maximum speedup was 10.507 for a matrix size of 128x128. In this case, the speedup grows near linear or even slightly better than linear with the number of workers. That a speedup is better than linear is usually suspicious but can be explained by the fact that each benchmark was repeated five times and the average times of this five executions were taken to compute the speedups. Five executions seem to be to small to get smooth results, especially when taking into account that the YARN container placement has big influences on the execution times.

The speedups for the 128x128 tiled matrix multiplication increase until a worker size of 12 is reached. At this point, no more improvements are achieved. The reason for this is the limited number of CPUs in the cluster.

The speedup for the 256x256 tiled matrix multiplication benchmark is worse compared to the 128x128 tiled matrix multiplication, although it also grows nearly linear until 12 workers.

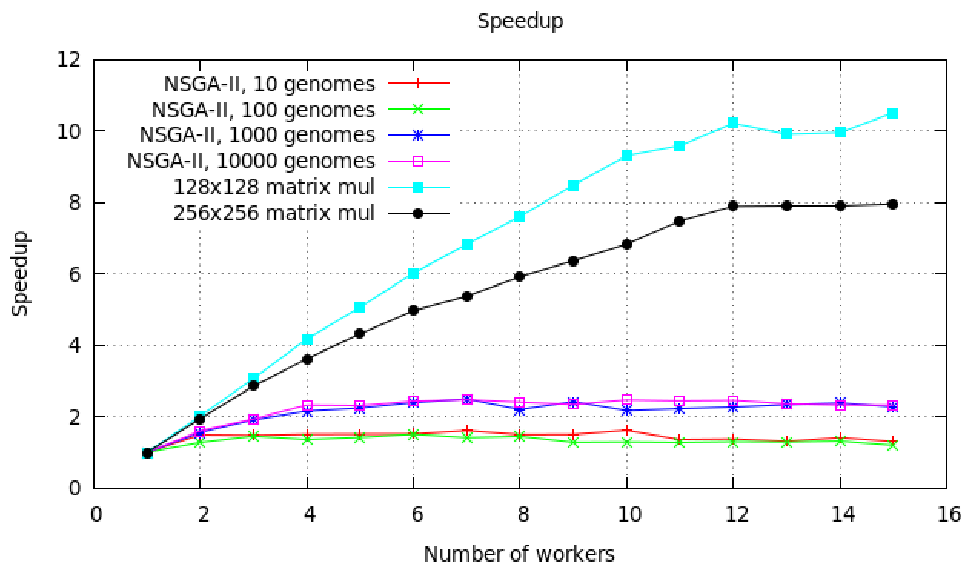

Figure 6.8: Speedups for ZDT-3 and tiled matrix multiplications

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

List of Figures

Bibliography

[1] S. Sivanandam and S. Deepa, *Genetic Algorithm Optimization Problems.* Springer, 2008.

[2] J. Kennedy, "Particle swarm optimization," in *Encyclopedia of Machine Learning*, pp. 760–766, Springer, 2010.

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

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013.

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

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

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*, pp. 2–2, USENIX Association, 2012.

[9] "Biohadoop." `https://github.com/gappc/biohadoop`. last access: 15.12.2014.

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

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

[12] M. Ehrgott, *Multicriteria optimization*, vol. 2. Springer, 2005.

[13] X.-S. Yang, *Nature-inspired metaheuristic algorithms.* Luniver press, 2010.

[14] M. Dorigo and M. Birattari, "Ant colony optimization," in *Encyclopedia of Machine Learning*, pp. 36–39, Springer, 2010.

[15] J. J. Durillo, A. J. Nebro, F. Luna, and E. Alba, "A study of master-slave approaches to parallelize nsga-ii," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, IEEE, 2008.

[16] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, 2002.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 29–43, ACM, 2003.

[18] "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.

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

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

[21] "Zookeeper." `http://zookeeper.apache.org/`. last access: 05.09.2014.

[22] "Example worker for biohadoop, written in javascript." `https://github.com/gappc/bioworker-browser`. last access: 22.09.2014.

[23] "Example worker for biohadoop, written in python." `https://github.com/gappc/bioworker-python`. last access: 22.09.2014.

[24] "Netty." `http://netty.io/`. last access: 19.11.2014.

[25] "Kryo." `https://github.com/EsotericSoftware/kryo`. last access: 04.08.2014.

[26] "Comparison of jvm serializers." `https://github.com/eishay/jvm-serializers/wiki`. last access: 27.11.2014.

[27] "Maven." `http://maven.apache.org/`. last access: 11.09.2014.

[28] "Docker." `https://www.docker.com/`. last access: 11.09.2014.

[29] E. Zitzler, K. Deb, and L. Thiele, “Comparison of multiobjective evolutionary algorithms: Empirical results,” *Evolutionary computation*, vol. 8, no. 2, pp. 173–195, 2000.

[30] K. Deb, “An efficient constraint handling method for genetic algorithms,” *Computer methods in applied mechanics and engineering*, vol. 186, no. 2, pp. 311–338, 2000.

[31] M. Wolfe, “More iteration space tiling,” in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pp. 655–664, ACM, 1989.

[32] “iftop.” `http://www.ex-parrot.com/pdw/iftop/`. last access: 08.12.2014.