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, 28 January 2015

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 28 January 2015

Abstract

Problem optimization is a fundamental task encountered everywhere, from everydays 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 mimic 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.

This master thesis presents a new framework, Biohadoop, to facilitate the implementation and execution of parallelized bio-inspired optimization techniques on Apache Hadoop. Its usefulness is demonstrated by the implementation and performance evaluation of two bio-inspired optimization algorithms. Finally, an extension to the workflow scheduler Apache Oozie is presented that simplifies the usage of Biohadoop as part of a larger workflow.

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 genetic algorithm (GA) [1] that uses a simplified model of natural evolution to improve the solutions towards an optimum. Another example is the particle swarm optimization (PSO) [2], which is based on the movement of bird flocks.

Most optimization problems are NP-hard. It is not feasible to explore all possible solutions, since this would simply take too much time. Bio-inspired optimization algorithms can't change this fundamental issue, however, for many problems their approximation techniques provide solutions which are "good enough" in an acceptable time.

Parallelization techniques can be applied to further reduce computational time by parallelizing a single sample evaluation or by evaluating more samples in parallel. 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[1], multi core[2] or specialized hardware (GPU[3], FPGA[4], ASIC[5]). Local approaches work well in many cases, but the increase of computational power is limited by the resources of a single computer.

Further performance increases can be achieved by distributing the work among several computers, forming a cluster. The downside of this approach is the increased overall complexity. On the one side, hardware components are unreliable and may fail (e.g. hard drives), which must be detected and handled in an appropriate way. On the other side, the computers need to communicate with each other through the network to distribute work tasks and data.

[1] `http://en.wikipedia.org/wiki/SIMD` last access: 07.01.2015

[2] `https://en.wikipedia.org/wiki/Multi-core_processor` last access: 07.01.2015

[3] `http://en.wikipedia.org/wiki/Graphics_processing_unit` last access: 07.01.2015

[4] `https://en.wikipedia.org/wiki/Field-programmable_gate_array` last access: 07.01.2015

[5] `https://en.wikipedia.org/wiki/Application-specific_integrated_circuit` last access: 07.01.2015

Apache Hadoop[6] 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 enables distributed applications to focus on the implementation rather than on cluster management concerns. Hadoop also provides the mechanisms to start, stop and monitor distributed applications and automatically restart them if needed.

Hadoop versions prior to 2.0 were restricted to the MapReduce [3] computational model. This restriction made it difficult to implement, e.g., iterative algorithms, like the bio-inspired optimization techniques. With the release of Hadoop 2.0 the resource management implementation has been changed to YARN [4] which makes no assumptions about the application being executed .

One drawback of YARN, however, is the missing 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 software projects try to improve Hadoop and solve the communication issue. Apache Storm[7] 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[8] supports both the stream model and a batch processing model on top of Hadoop. In addition, it provides a distributed in-memory store [5].

Biohadoop[9] is another project whose goal is 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. The focus on the master-worker pattern makes it more lightweight than the previous solutions mentioned above, provided that the problem fits into the master-worker pattern.

In this thesis, Biohadoop is introduced and its usefulness is demonstrated with two bio-inspired optimization techniques. The thesis provides additional information about Apache Oozie[10] (a Hadoop workflow tool) 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

[6] `http://hadoop.apache.org/` last access: 07.01.2015
[7] `https://storm.apache.org/` last access: 01.10.2014
[8] `https://spark.apache.org/` last access: 01.10.2014
[9] `https://github.com/gappc/biohadoop/` last access: 15.12.2014
[10] `https://oozie.apache.org/` last access: 01.10.2014

common representatives: GA and PSO. Chapter 3 provides information about Hadoop and Oozie needed to understand the functionality of Biohadoop. Chapter 4 explains Biohadoops architecture and the modifications implemented for Oozie (section 4.6). Chapter 5 evaluates the performance of Biohadoop using two different implementations of a GA. The conclusions in chapter 6 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) [6]. In TSP, a salesman needs to visit a set 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, the objective is 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* or *search space*
- $f : S \mapsto \mathbb{R}$ is the *objective function* that we want to minimize or maximize

The process of finding the global optimum is called global optimization. The global minimum optimization for the problem stated above is given by

$$s' \in S \mid (f(s') \leq f(s) \; \forall \; s \in S) \tag{2.1}$$

Here, s' is the solution within region S which minimizes the objective function f.

2.2. Multi Objective Optimization

In a *multi objective optimization problem* (MOP) we have several conflicting objectives that we want to optimize at the same time. Non conflicting objectives can be combined leading to a SOP.

Definition (MOP): a MOP is defined by $P = (S, \mathbf{F})$, where

- S is the set of possible solutions also called *solution space* or *search space*
- $\mathbf{F} = (f_1, ..., f_k) : \ S \mapsto \mathrm{I\!R}^{\mathrm{k}}$ are the k objective functions that we want to minimize or maximize

To optimize a MOP, a vector $\mathbf{x} = [x_1, ..., x_k] \in S$ needs to be found 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 $\mathbf{F}$. It should be noted that $g_i(\mathbf{x}) \geq 0$ and $h_i(\mathbf{x}) = 0$ represent constraints that must be satisfied for a feasible solution $\mathbf{x}$.

If we extend the TSP example, two objectives have to be found, a) the shortest path, with b) the lowest costs. The shortest path includes driving on the highway (causing higher costs due to toll fee), the cheapest path forces the salesman to drive on a normal street (longer distance). Therefore, we have to find a compromise between the two conflicting goals which means that there isn't a single best solution, but a set of optimal solutions. Some solutions may result in a shorter path, where other ones may result in lower costs. When optimizing a MOP, the task is to find the 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 figure 2.1(a), solution A is better than solution B, since each component in A is smaller than in B. In figure 2.1(b), B is better than A for the same reason. The situation in figure 2.1(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. Although, more components in A are smaller than in B it is unclear which solution is better, since 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 shorter distance or the lower gas consumption? This can not be answered in general and depends on the users preference at a given moment of time. To find a set of solutions for a MOP, the Pareto Optimality theory is used [7].

The Pareto Optimality theory defines the concept of Pareto Dominance, that can be applied 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 smaller than $\mathbf{v}$, i.e., $\forall i \in \{1, ..., k\}, u_i \leq v_i \wedge \exists i \in \{1, ..., k\} : u_i < v_i$.

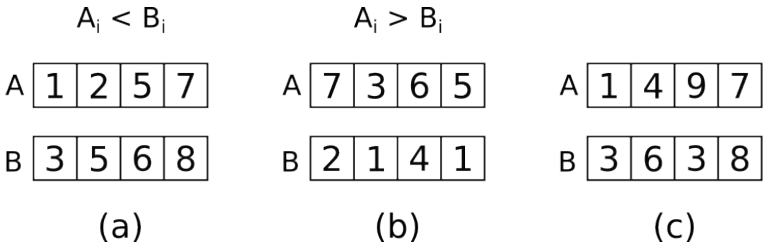

Figure 2.1.: Solution comparison: (a) A is better than B, (b) B is better than A, (c) none is better — A and B are non-dominated

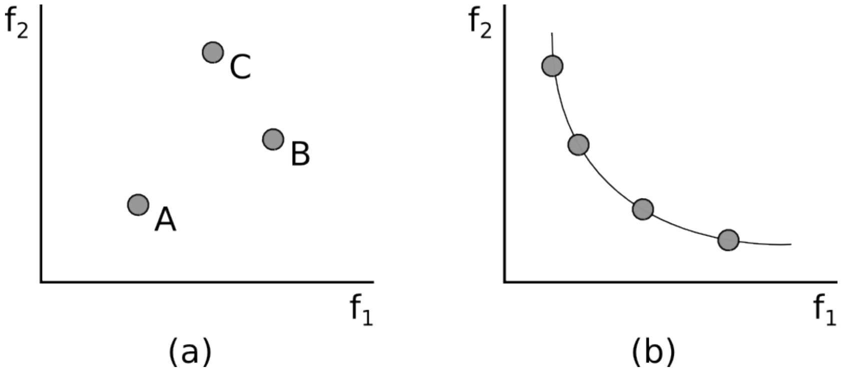

Figure 2.2.: Pareto Dominance: (a) A dominates B and C, (b) all points are non-dominated

In figure 2.2(a) A dominates the solutions B and C because

- The values for f_1 and f_2 of A are equal or smaller than the corresponding values for B and C.

- At least one of the values for A is smaller than the corresponding values for B and C.

In figure 2.2(b) no solution dominates another solution.

Using the concept of dominance, it is possible to define one optimal solution among other solutions. This is known as Pareto Optimality.

Definition (Pareto Optimality): a solution $\mathbf{x}$ is Pareto Optimal, if there is no $\mathbf{x}' \in S$ 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 $\mathcal{P}^* = \{\mathbf{x} \in S | \nexists \mathbf{x}' \in S, \mathbf{F}(\mathbf{x}') \preceq \mathbf{F}(\mathbf{x})\}$

Its correspondence in the objective space (that is, the space where the results of the objective functions are located) 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 $\mathcal{PF}^* = \{\mathbf{F}(\mathbf{x}) | \mathbf{x} \in S\}$

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

- has good convergence to the optimal Pareto Front, i.e., it is as close to the optimal Pareto Front as possible

- has good diversity, i.e., the solutions are well distributed throughout the Pareto Front

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

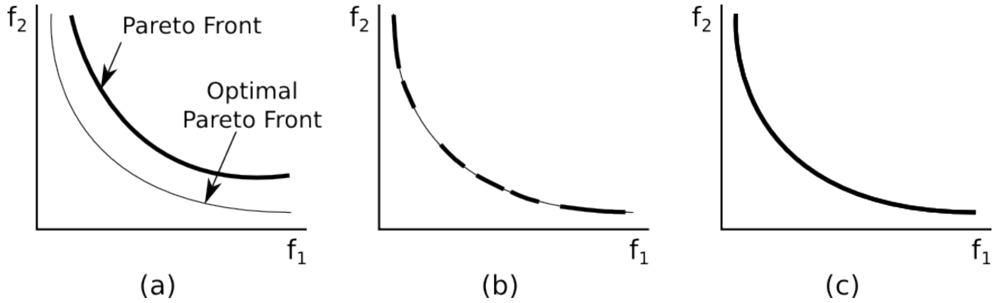

Figure 2.3.: Pareto Front examples: (a) Pareto Front with bad convergence, (b) Pareto Front with bad divergence, (c) ideal case, where the Pareto Front matches the optimal Pareto Front

2.3. Complexity Considerations

Optimization is usually a computationally intensive task. For the TSP example, we get $N = (n - 1)!/2$ different solutions. Here, n is the number of cities. Already a small number of cities results in a large number of solutions. For example, if we have 15 cities, we have a search space of over 43×10^9 solutions that we need to evaluate to get the best solution (assuming that we have no better suited technique to solve the problem). This number grows extensively with a growing number of cities and soon it becomes impossible to compute the optimal solution.

Often there is no need to find the best solution to a problem, or it is even impossible, e.g., if there are to many candidate solutions. 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, since they don't evaluate all solutions of the entire solution space. The advantage of approximation techniques is that they deliver solutions quite quickly. The results are usually near-optimal, although they can also be arbitrarily bad.

One well known family of approximation techniques are the metaheuristics [8]. 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 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.

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 Algorithms

Genetic algorithms (GAs) are population-based optimization strategies. The population consists of n individuals each representing a sample for the optimization problem. Every individual gets assigned a fitness value which represents its adoption to the environment. The fitness value is computed according to the optimization objective.

The idea behind genetic algorithms is to iteratively evolve the population towards individuals with better adoption to the environment, ultimately leading to solutions of higher quality. The evolution is performed by applying selection, recombination and mutation to the population.

Recombination is performed by selecting two or more individuals out of the whole population. Those individuals are called the parent individuals. Fitter individuals are preferred for the recombination, as they have a high chance to produce fit offsprings. The parents are recombined using an operator called crossover. 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.

At the end of each iteration, the fitness of all individuals (parents and children) is computed and the 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.

The GA algorithm can be used to solve SOPs and MOPs. In the case of MOPs, 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 [9]. 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 not yet ranked individuals, and assigning the current rank to them. The rank starts at 0 and after each iteration it increases by one. The iteration stops after all individuals are ranked.

Algorithm 1 Genetic algorithm

```
procedure GA
    $P$ ← generateInitialPopulation
    evaluate($P$)
    while !$terminationCriteria$ do
        $P'$ ← recombine($P$)
        $P''$ ← mutate($P'$)
        evaluate($P''$)
        $P$ ← select($P$, $P''$)
    return best solution found so far
```

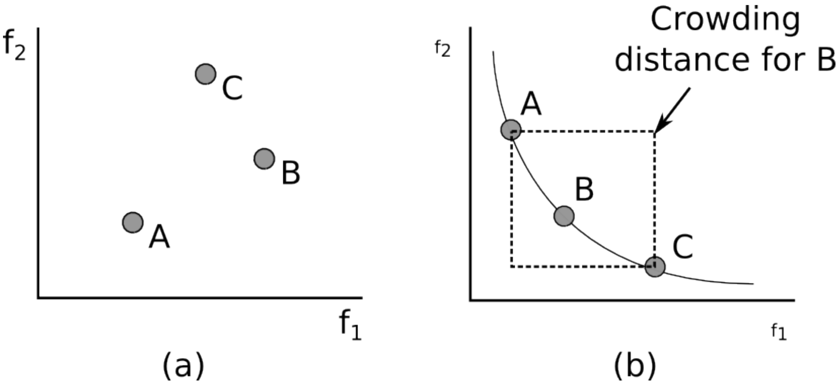

Figure 2.4.: Ranking and crowding distance: (a) A has rank 0, B and C rank 1, (b) crowding distance of solution B.

The crowding distance is used to sort the best individuals of a given rank. This happens when p out of q individuals of a given rank need to be selected (with $p < q$). The crowding distance measures how close neighboring solutions are to a given individual. Higher crowding distances are preferred, since this leads to better diversity in the final solution.

An example can be found at figure 2.4. In figure 2.4(a) we have a population of three individuals, namely A, B and C. It can immediately be seen that A is non-dominated but dominates B and C. Therefore, 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 not yet 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 B and C. After this iteration the ranking algorithm stops, since all individuals have been ranked.

Figure 2.4(b) gives a graphical representation of crowding distance that is used to select the p out of q best solutions for individuals with the same rank.

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 knows about the global best solution found so far (gBest), and its personal best solution (pBest), found so far. A particle moves in the direction of gBest and pBest. 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. The velocity $\mathbf{v}(t)$ of the particle at time t is given by

$$\mathbf{v}(t) = w\mathbf{v}(t-1) + c_1 r_1(\mathbf{pBest} - \mathbf{x}(t-1)) + c_2 r_2(\mathbf{gBest} - \mathbf{x}(t-1)) \quad (2.2)$$

$\mathbf{v}(t-1)$ is the velocity in the previous iteration, $\mathbf{x}(t-1)$ is the according particle and w is called the inertia weight that defines the influence of the current speed on the new velocity. Parameters c_1 and c_2 are the cognitive acceleration coefficient defining the influence of the personal best solution and social acceleration coefficient defining the influence of the global best solution, respectively. Parameters r_1 and r_2 are random values between 0 and 1 used as a source of diversity. This vector operation is performed for each iteration on all particles.

After the particles velocity is computed, its current position is updated using

$$\mathbf{x}(t) = \mathbf{v}(t) + \mathbf{x}(t-1) \quad (2.3)$$

An example of two iterations for a single particle can be found in figure 2.5. Figure 2.5(a) shows the initial situation, the red dot highlights gBest, the gray dot the particle. Figure 2.5(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 figure 2.5(c). Figure 2.5(d) and 2.5(e) show an additional iteration conforming to the same principle. The value of pBest is not considered in this example.

PSO can also be used to compute solutions to a MOP. In this case, the non-dominated solutions are stored in an archive. New solutions are inserted into the archive if they are non-dominated by the elements in the archive. Subsequently

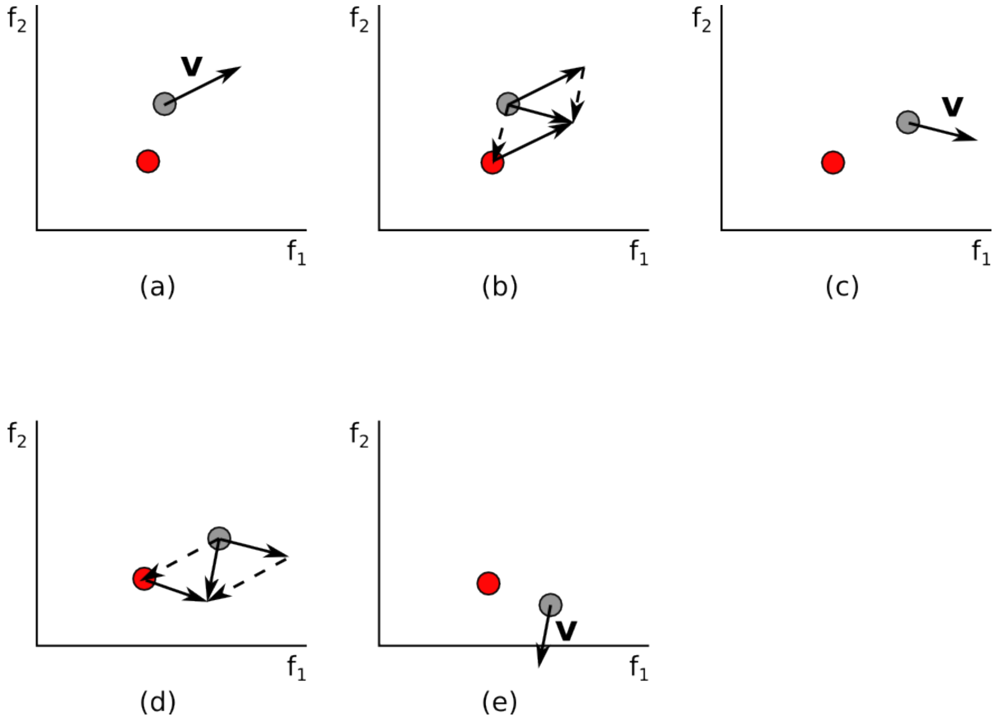

Figure 2.5.: Two iterations of PSO for a single particle. The red dot marks gBest, the gray dot the particle. The velocity and position of the particle does rely only on gBest in this example, pBest is not used

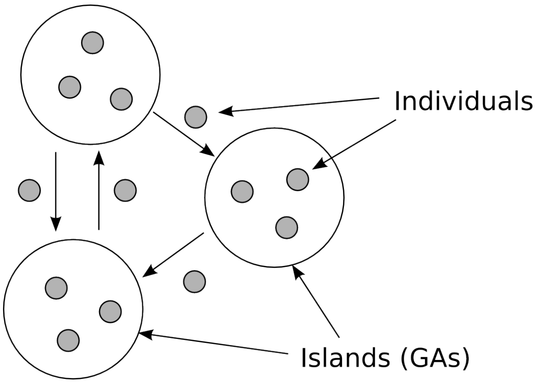

Figure 2.6.: Island model example with 3 GAs exchanging individuals.

the archive is scanned for dominated solutions that can arise from new inserted solutions. The dominated solutions are removed from the archive. The non-dominated solutions to the MOP can be extracted from the archive after PSO terminates.

2.5. Parallelization using the Island Model

The island model is a high level parallelization model that is sometimes used in optimization problems. In the island model several optimization 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 to the others and each one may have a different solution at a given 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 2.6 shows an example island model with 3 GAs that exchange individuals.

The island model enhances the exploratory behavior of optimization algorithms, which often results in better overall solutions. Since the data exchange happens only at certain iterations, 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 — the framework implemented during this thesis — provides the capabilities to execute an island model. More information can be found in section 4.4.2.

Chapter 3.

Hadoop

Apache Hadoop is an open source software project for massive data processing and parallel computing in a cluster. It derives from Google's publications about MapReduce [3] and Google File System (GFS) [10], but has undergone quite some changes due to its active development, e.g., the introduction of YARN (see section 3.2).

Hadoop is designed to run on 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.

In the following key properties of Hadoop are listed:

Scalability : Hadoop is able to scale horizontally, which means that the performance scales with the number of cluster machines. The performance can be improved by dynamically adding nodes at runtime if required. Furthermore, unnecessary nodes can be shut down if their additional performance is not needed.

Cost effectiveness : Since Hadoop runs on commodity hardware there is no need for exclusive, specialized proprietary hardware. It is possible to run Hadoop on an existing infrastructure. Hadoop runs also in the cloud. Different offers from Amazon, Google, Cloudera, etc. exist. This can further reduce costs, as the customer has to pay only for the resources used and doesn't need to maintain its own data center.

Flexibility : Hadoop can handle any kind of data. Custom applications running on Hadoop enable the transformation of, and computation on arbitrary data. For example, Hadoop can be used to analyze log files. It can equally well be used by physicists to find patterns in their huge datasets.

Fault tolerance : Hadoop expects hardware failures and is designed from ground up with fault tolerance in mind. If a machine fails, Hadoop automatically redirects the computations and data of this machine to another machine.

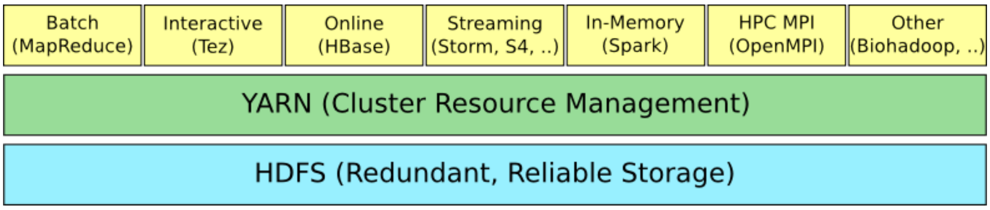

Figure 3.1.: Hadoop layers: HDFS forms the base and provides data services, YARN builds on it and provides resource management. The applications run on top of YARN.

Hadoop consists of two main components, HDFS (Hadoop Distributed File System) and YARN (Yet Another Resource Manager). HDFS is a distributed, 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.

YARNs concept of resources is more flexible than the MapReduce resource management model used for Hadoop prior to version 2.0. The old resource management is tailored to MapReduce tasks and divides the cluster resources into separate map and reduce slots. Map tasks and reduce tasks are only executed on map slots and reduce slots, respectively. The number of map and reduce slots are statically configured and established during the startup of Hadoop. This can lead to poor resource usage if the MapReduce applications don't fit to the configured number of slots. The situation gets even worse for applications with a different computational model like Biohadoop, which provides a framework for iterative computations. With the old resource management model iterative tasks run either on the map slots or on the reduce slots while unused slots idle. Therefore, resources are not fully exploited. YARN, on the other hand, is able to host arbitrary computational models, as it provides the resources in an application agnostic way. More details about YARNs resource management can be found in section 3.2.

Figure 3.1 shows the current architecture of Hadoop. HDFS provides data services as the basic layer. YARN builds on top of HDFS and manages the resources of the cluster. The different applications, like MapReduce, Storm, Spark, Biohadoop, etc. 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, commands the DataNodes and decides where to store the data. 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 more 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 decision where to store data replicas is made by the NameNode, the exchange of replicas is performed directly between the 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 than 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 that make use of a Quorum Journal manager[1] or NFS.[2]

Figure 3.2 shows how HDFS organizes the data. Here, 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.

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 for a variety of advanced features, e.g., it is faster to execute computations on the nodes where the required data is already available instead of first moving the data to another node. This minimizes possibly slow data traffic. Another example is HDFS itself, that uses its rack awareness for its

[1] `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html` last access: 12.11.2014

[2] `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html` last access: 12.11.2014

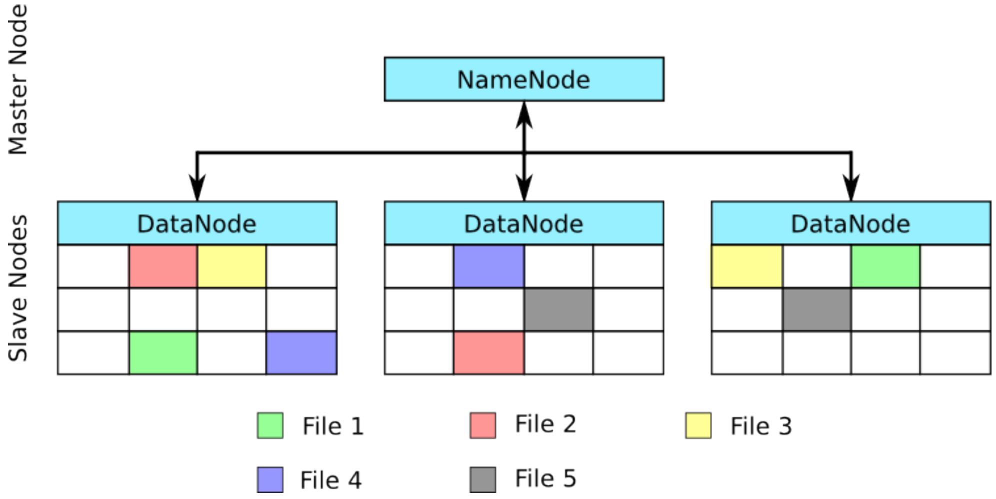

Figure 3.2.: HDFS file storage, each file is replicated twice.

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. It also 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 do also exist here in the form of standby ResourceManagers.[3]

The ResourceManager has two components, the Scheduler and the ApplicationsManager. The Scheduler allocates resources for 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.

[3] `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html` last access: 13.11.2014

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 (not to confuse with the 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 via a heartbeat. 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, confer to section 4.3.

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 simultaneously, as long as there are enough cluster resources available. Applications are queued if the currently available resources are not sufficient. 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

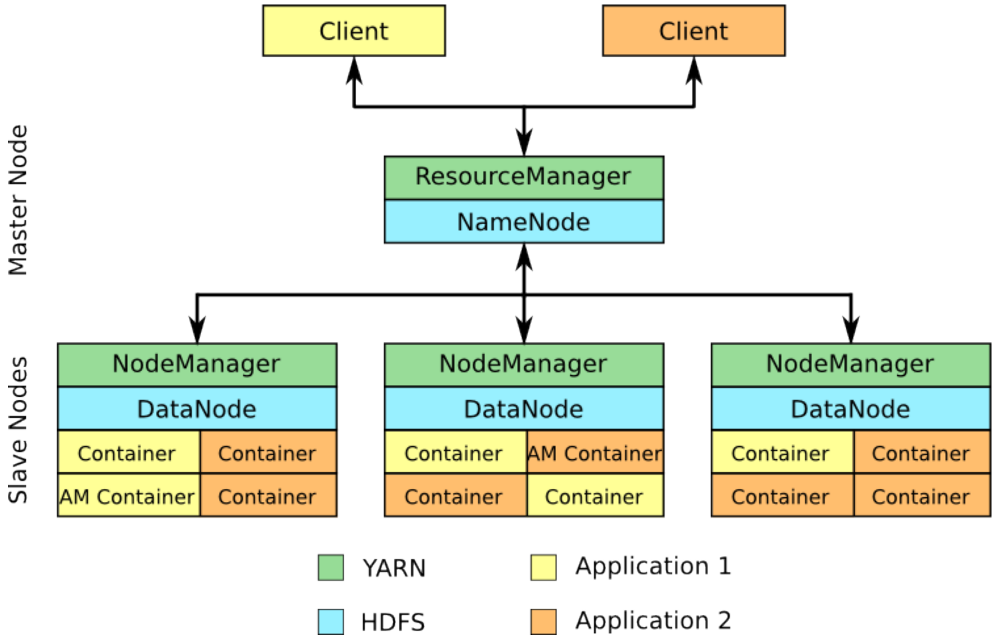

Figure 3.3.: Occupation of a Hadoop cluster for two applications. The NameNode (HDFS) and ResourceManager (YARN) run on the master node. DataNodes (HDFS) and NodeManagers (YARN) run on the slave nodes and communicate with the corresponding services on the master node. The applications, including the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

applications are started. Application 1 occupies five containers and Application 2 occupies seven 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.[4] Oozie uses an internal scheduler to make decisions about the submitted workflows and their progress, the action execution is delegated to Hadoop.

Each Oozie action has two possible outcomes: "ok" if the action completed successfully, 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

[4] `http://www.w3.org/TR/xml/` last access: 09.01.2015

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 this work a custom action was implemented to invoke Biohadoop from Oozie. More details about this custom action and its configuration can be found in chapter 4.6.

Figure 3.4 shows a workflow example. It is started at the control flow node labeled “Start” followed by the “FS action”. The transition to the next step is depending on the result of this action. Actions “Failed” and “Fork” are invoked in case of an error or success, respectively. The “Fork” action starts two Java actions in parallel. Actions performed inside a “Fork” action have a special error semantic. If an error occurs in any of the parallel actions inside a fork node, all parallel actions are considered as failed. If no error happened, the “Join” node waits for two (or possibly more) parallel 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 has no errors it terminates at the “End” node.

It can be difficult and error prone to manually write a workflow XML. Graphical tools like Hue [5] can simplify this process.

[5] http://gethue.com/category/oozie/ last access: 26.12.2014

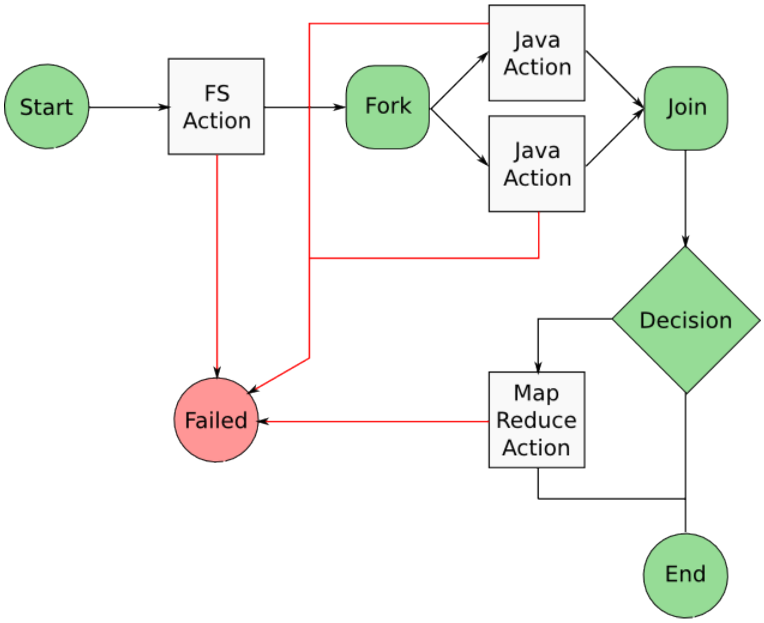

Figure 3.4.: Oozie workflow transitions. If any action returns with an error, the workflow transitions to the final state “Failed”, else the workflow advances according to the defined sequence.

Chapter 4.

Biohadoop 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.

In the following, we are going to show how to parallelize the GA from chapter 2.4.1. A GA is an iterative procedure, each iteration creates a set of new individuals, called 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 and can be executed in parallel since it depends only on its input values (the parents) and has no side effects. Therefore it is a good example for a parallel task that can be performed by workers. The master controls the generational loop and workers create and evaluate individuals (see figure 4.1).

The master-worker architecture was chosen for Biohadoop, because many bio-inspired algorithms can be parallelized by this approach (like GA and PSO of 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 supports features that are not provided by YARN, 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. The island model (section 2.5) benefits from this feature, as it ensures the simultaneous execution of the islands.

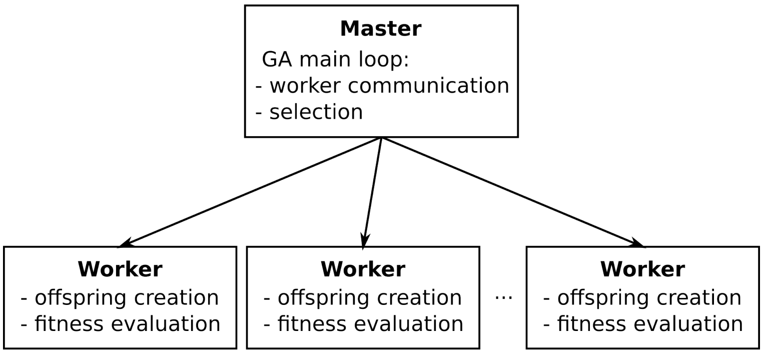

Figure 4.1.: Master-worker principle applied to a GA. One master commands several workers. The master runs the main loop, where it communicates with the workers to get new individuals. Then, it selects the best individuals to create the population of the next iteration.

- Asynchronous communication between master and workers using Biohadoops task system (section 4.2). 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 4.4.1).
- 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 4.4.2).
- Support for Apache Oozie through a custom action (section 4.6). 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, since it is the master in the master-worker scheme, it executes the configured algorithms in this single ApplicationMaster instance 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 Hadoop (more on how to run Biohadoop can be found in appendix A.1).

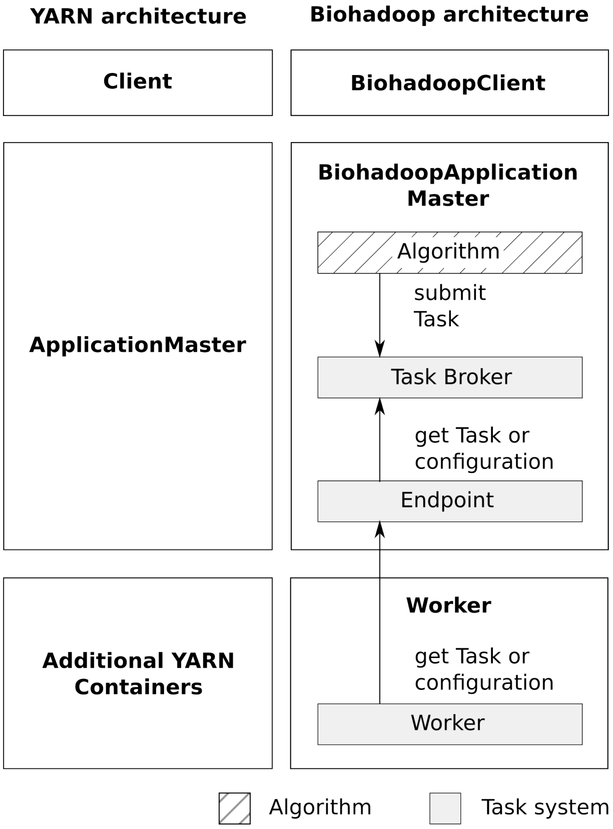

Figure 4.2.: Mapping of Biohadoops architecture to the architecture of a typical YARN application

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.

An algorithm uses the task system to submit work items, from here on called tasks, to the workers and to wait for the results. A task contains data and a reference to a task configuration. A task configuration 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 (fig. 4.3(a)) which is delivered (fig. 4.3(b)). The worker then recognizes that it doesn't know how to compute the result for the task. Therefore, it has to request the task configuration (fig. 4.3(c)). The task configuration is returned (fig. 4.3(d)). The worker can now compute the task result (fig. 4.3(e)). The result is returned (fig. 4.3(f)) together with the request for the next task. The next task is delivered (fig. 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 (fig. 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 4.2.

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, e.g., 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 4.4.1.

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. Since 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. 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. Algorithms that register to ZooKeeper[1] find each other also across the boundaries of different Biohadoop instances. The regis-

[1] `http://zookeeper.apache.org/` last access: 05.09.2014

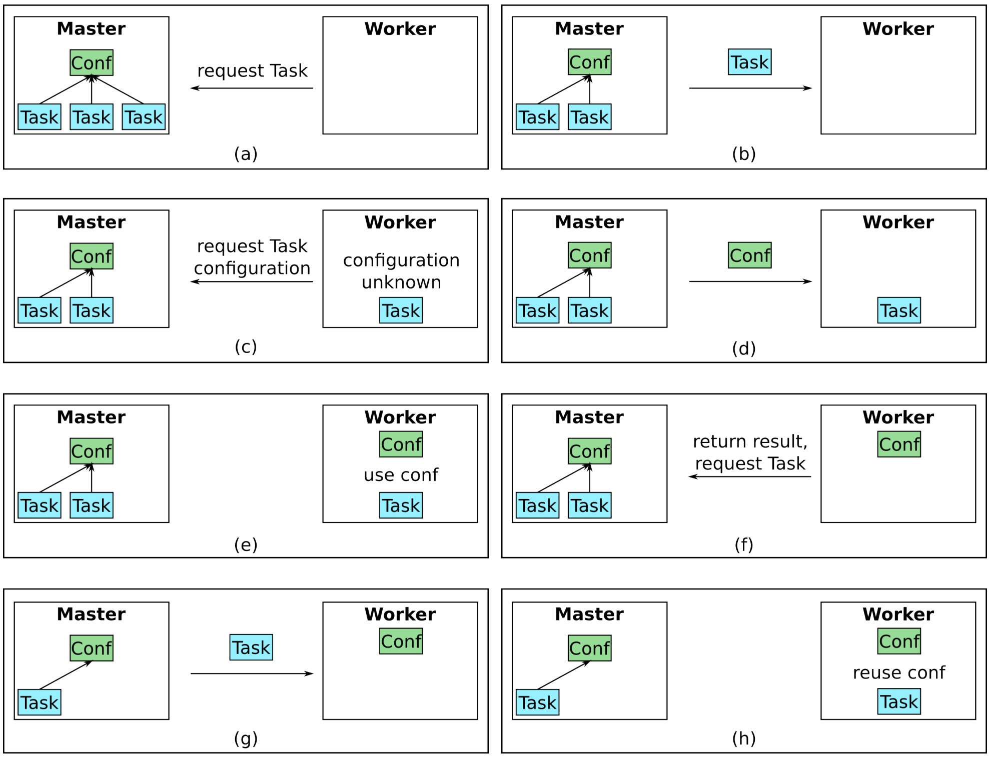

Figure 4.3.: Task requests performed by a worker. The task configuration is send to the worker on demand.

tration and data exchange is hidden behind Biohadoops island model API (see section 4.4.2). The execution of algorithms in different Biohadoop instances can lead to higher scalability, since each instance gets its own resources. However, it entails the aforementioned problem that YARN doesn't guarantee the order of execution, so a compromise has to be made.

Biooozie (section 4.6) implements a custom action for Apache Oozie and is used to run Biohadoop as part of a larger workflow. As an example, a workflow imports data from a set of sensors into HDFS, performs a MapReduce task on this data to extract the relevant information and executes Biohadoop to find a solution for a given problem.

In the following sections of this chapter Biohadoop is described in more detail, starting with the notion of Algorithm in section 4.1. Section 4.2 presents the task system and its components, followed by the description of the communication mechanisms in section 4.3. The extensions section 4.4 provides information about Biohadoops support for persistence and the island model. Section 4.5 explains how to configure Biohadoop. Finally, Biooozie is presented in the section 4.6.

4.1. 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 provides an easy way to parallelize the algorithm using its asynchronous task system (see section 4.2). The algorithm can submit tasks to the task system that 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 section 4.4).

All those capabilities can be used by the algorithm, which must 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.

It is possible to run several algorithms of any kind simultaneously in one Biohadoop instance (e.g. two GA and one PSO). This is just a matter of configuration (see section 4.5).

If an error occurred during the execution of an algorithm, the algorithm may throw a `AlgorithmException`. The meaning of a thrown `AlgorithmException` is, that there was an unrecoverable error which prevented the algorithm to continue, 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. It is, however, 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`.

4.2. 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 to 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 `TaskSubmitter`, due to the simple interface. On the other hand, 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 not yet 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

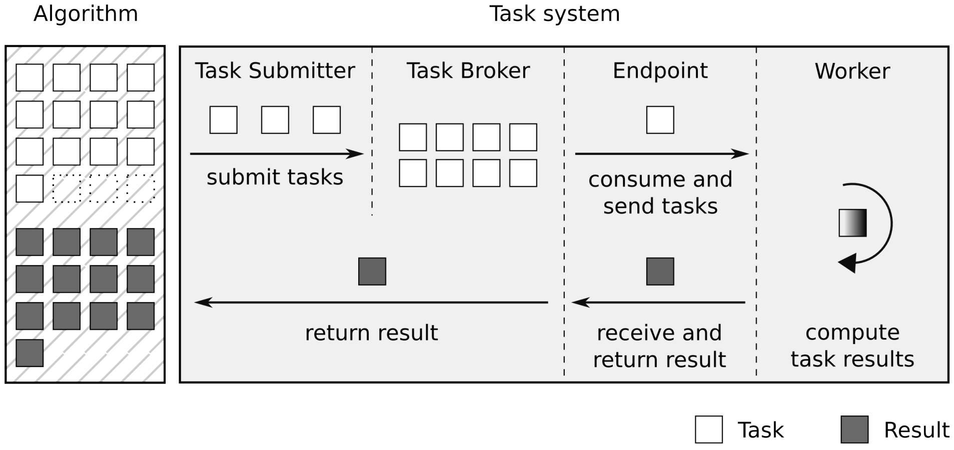

Figure 4.4.: Task submission to the task system. The task system consists of a task broker and any number of endpoints and workers. For the sake of simplicity, the figure shows only one endpoint and one worker. Here, an optional task submitter is used for task submission.

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 by using the task data and its related task configuration, and returns the result to an endpoint. Finally, the endpoint 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. If preferred, however, it provides also mechanisms to block and wait for a result to be computed.

4.2.1. 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 broker. The configuration itself is sent 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, since usually many tasks use the same configuration. The cache is only limited by the amount of available memory.

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. In this case, however, the communication overhead would increase, since task configurations have to be transmitted individually.

To work properly, the task broker uses two concepts that are needed to perform its work. The first concept is an 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 simultaneously, without interfering with each other.

The second concept is a map connecting submitted tasks to their configuration and their `TaskFuture`. This is necessary, because a task loses its references once it is taken out of the FIFO queue and sent 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 usage of a map resolves this problem.

The two concepts are depicted in figure 4.5. In the first step, a task T_N and its configuration TC_N 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` TF_N 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 sent 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 R_N to the endpoint, which forwards it to the broker. The broker associates the result for task T_N with the according `TaskFuture` TF_N using its internal map. After the result for the `TaskFuture` is set, the algorithm can access the result R_N for task T_N without blocking.

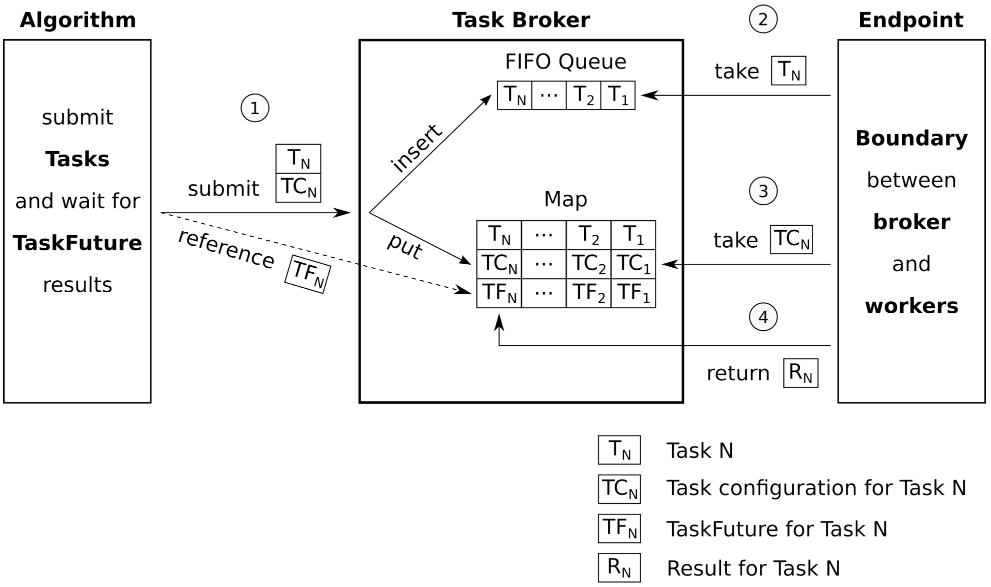

Figure 4.5.: Internal structure of the task broker

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.

4.2.2. 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. Details about the communication can be found in section 4.3.

Communication is not the only purpose of an endpoint. It interacts also with the task broker by 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 required. For example, lets suppose a task is taken from the task broker and sent 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 resubmits 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 4.3).

The endpoints run in the YARN ApplicationMaster and are started and stopped automatically by Biohadoop. It is configurable which endpoints should be started, section 4.5 gives details about the configuration aspects.

4.2.3. 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 sent to the endpoint.

Tasks are sent 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.

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.

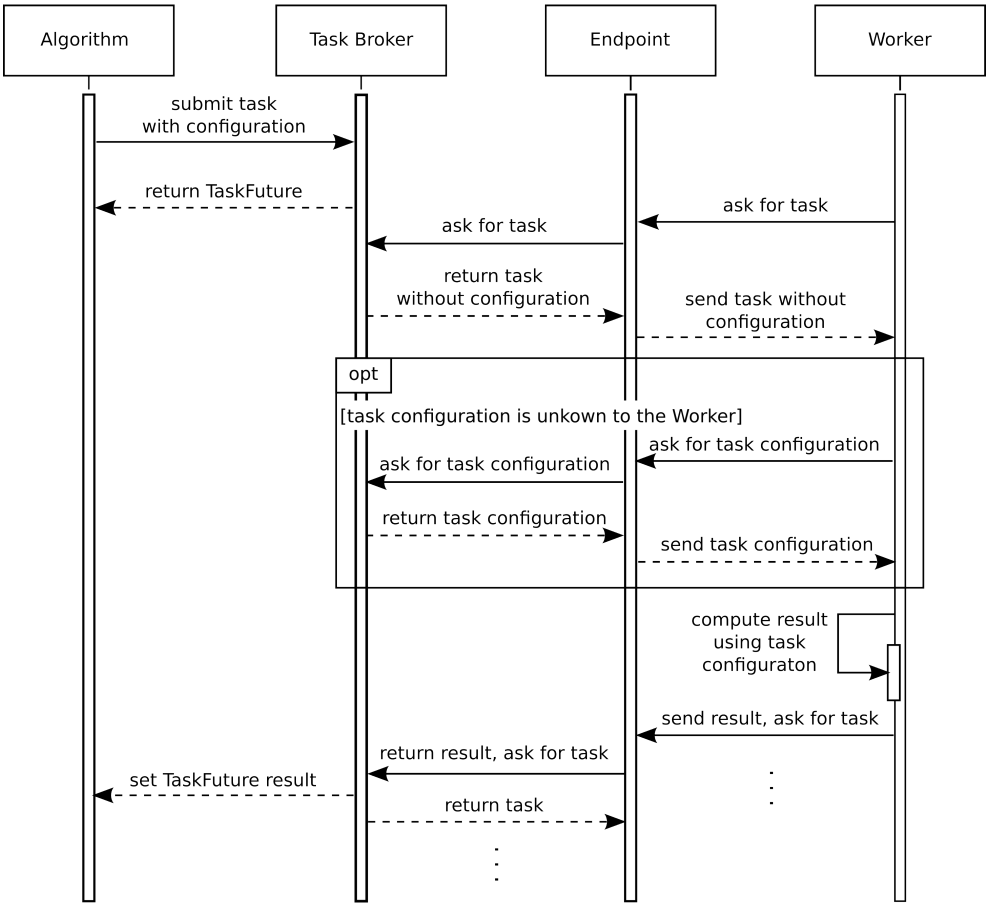

Figure 4.6.: Life cycle of a task. The result is computed by a worker using an `AsyncComputable`.

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.

For most applications embedded workers are just fine. There are, however, 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 programming 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 [2] or Python.[3] In contrast, 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.

4.3. 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.

[2] `https://github.com/gappc/bioworker-browser` last access: 07.01.2015

[3] `https://github.com/gappc/bioworker-python` last access: 07.01.2015

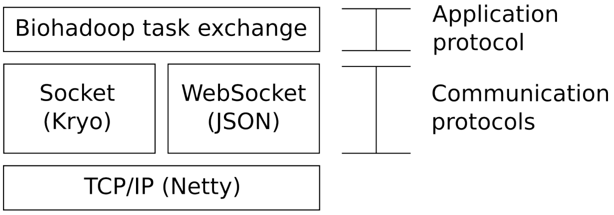

Figure 4.7.: Biohadoops communication layers

Communicating between endpoints and workers is more complicated, as the endpoints and workers may run in different processes or even on different machines, which makes variable sharing more difficult. A more sophisticated communication method must be used.

Biohadoop uses Netty[4] 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. Although, 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[5] for object serialization, while the WebSocket protocol relies on JSON serialization. The two protocols are discussed in more detail in section 4.3.1.

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 4.3.2. Figure 4.7 shows the different layers used by Biohadoop for data exchange.

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] `http://netty.io/` last access: 19.11.2014
[5] `https://github.com/EsotericSoftware/kryo` last access: 04.08.2014

4.3.1. 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 is typically more error prone.

Kryo is used for object serialization. It is a library for high speed serialization of Java objects and is usually faster than the build-in serialization features of Java.[6]

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. However, 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 4.5).

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 streams or text streams.

WebSockets have a very small overhead when data is exchanged, e.g., 2 Bytes for text stream messages. This is a major difference to the HTTP protocol where the larger 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

[6] `https://github.com/eishay/jvm-serializers/wiki` last access: 27.11.2014

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 4.3.2 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, since they are standardized. The biggest disadvantage of WebSockets with JSON is, that they are slower than sockets with Kryo serialization.

4.3.2. 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 communication protocols available. 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 for further reuse. 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.

4.4. Extensions

Biohadoop provides two extensions for algorithms, beside the task system presented in section 4.2. The first one is for persistence, where it is possible to load and store arbitrary data to and from a file system (see section 4.4.1). The second extension provides high level parallelism between parallel running algorithms, called the “island model” (see section 4.4.2).

4.4.1. 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 store intermediate results in case something happens. If this data is reloaded

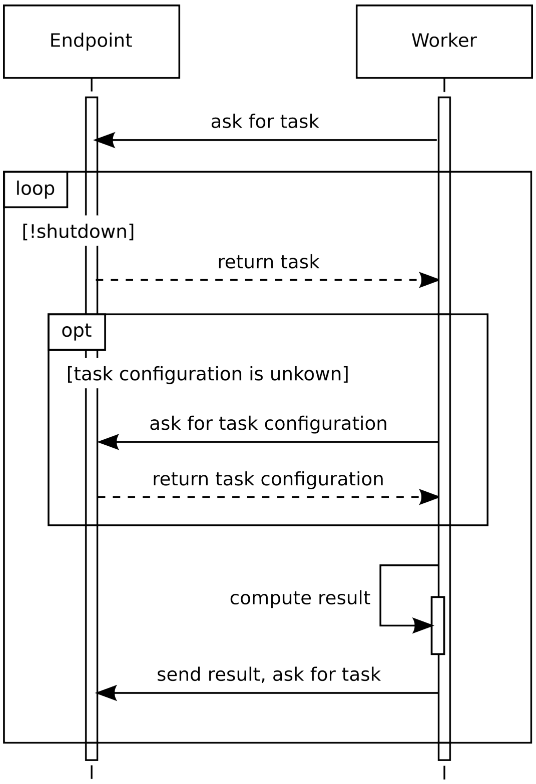
Endpoint
Worker
ask for task
loop
[!shutdown]
return task
opt
[task configuration is unkown]
ask for task configuration
return task configuration
compute result
send result, ask for task

Figure 4.8.: Communication flow between endpoints and workers

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.

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

4.4.2. The Island Model

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 register to ZooKeeper, 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, 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 simultaneously. Scheduling several Biohadoop instances in parallel doesn't guarantee that they also run in parallel, as YARN decides when an application is launched. The island model is useless if the algorithms don't run at the same time.

4.5. 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 appendix A.1). Biohadoop stops immediately with an exception if the path is empty, not existing 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 of keys and values as strings. 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 defining the paths where needed libraries can be found. This isn't important for a locally running instance of Biohadoop (e.g. during development), as the necessary classpaths must be set when starting Biohadoop. It is, however, important when Biohadoop runs in the Hadoop environment, since these 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.

Although possible, it is not always convenient to write a configuration by hand. Biohadoop provides two builder classes to simplify the generation of 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.

4.6. Biooozie

Biooozie implements a custom action for Apache Oozie (see section 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

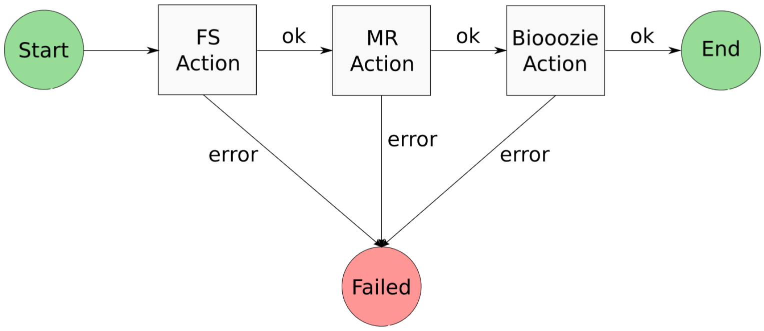

Figure 4.9.: Example Oozie workflow, including Biooozie action

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 basis for a GA computation, performed by Biohadoop (see figure 4.9).

The action is configured in an Oozie workflow as an XML element named `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 sequentially.

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.

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). That means the speedup increases when computational resources are added. To verify this assumption, two bio-inspired optimization algorithms are parallelized using Biohadoop. Their parallel parts are executed on Biohadoop workers. The execution times of the algorithms are measured on a Hadoop cluster. Then, the speedups are calculated based on those execution times. The speedups demonstrate if the algorithm execution times benefit from the parallelization.

The rest of this chapter is structured as follows: section 5.1 provides information about the cluster used for the benchmarks. Section 5.2 describes the benchmarked test problems. The benchmark results are presented in section 5.3 together with a check for their correctness. Section 5.4 gives a discussion of the most notable factors that influence the execution time. The speedup calculations can be found in section 5.5. Finally, section 5.6 compares the execution times of Biohadoop parallelellized algorithms to their standalone implementations.

5.1. Cluster Hardware

All benchmarks are performed on a Hadoop cluster with 6 identical computers. Each machine has the following specifications:

- Intel Core2 Duo CPU E8200 @ 2.66 GHz (2×2.66 GHz, no hyperthreading)
- 6 MB shared L2 cache, 32 KB L1 data cache, 32 KB L1 instruction cache
- 4 GB (2×2 GB) DDR2 RAM @ 667 MHz
- 64 Bit Ubuntu Linux 14.04.1 LTS with kernel 3.13.0-37

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

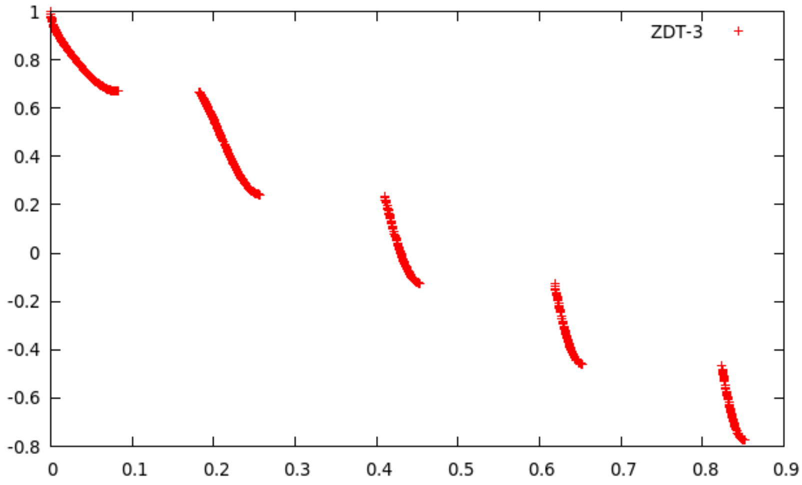

Figure 5.1.: Optimal Pareto Front for ZDT-3

5.2. Test Problems

NSGA-II and a simple GA are the implemented bio-inspired optimization algorithms used for the benchmarks. They can handle different numbers objectives. While NSGA-II is used to solve the MOP in section 5.2.1, a simple GA is used to solve the SOP in section 5.2.2.

5.2.1. ZDT-3

The first optimization algorithm is NSGA-II, used to find optimal solutions for the Zitzler–Deb–Thiele's function nr. 3 [11]. ZDT-3 is part of the well known ZDT family of MOP. It was chosen because of its discontinuous optimal Pareto Front (see figure 5.1) that can pose problems to simpler optimization strategies, e.g., gradient based algorithms. The expected outcome is to find an approximation to the optimal Pareto Front.

The implementation uses Biohadoop workers to create and evaluate the offsprings. Simulated Binary Crossover (SBX) and Parameter based mutation [12] 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.

The ZDT-3 benchmarks are executed with genome sizes of 10, 100, 1000 and 10000. The genome size 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 is due to the fact that new individuals are generated using 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 sent between master and workers is, therefore, related to the gnome size of each individual.

5.2.2. Tiled Matrix Multiplication

The second benchmark implements a GA to solve the SOP for finding optimal tile sizes for the tiled matrix multiplication (TMM). The objective is to minimize the execution time for a matrix multiplication. The expected outcome are tile sizes that minimizes the TMM execution time.

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 [13]. 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. Tests with a matrix size of 1024×1024 were performed on a single computer of the test cluster to demonstrate the advantages of TMM. Table 5.1 shows that suitable tile sizes can reduce the execution times for a tiled matrix multiplication. The results show also that bad tile sizes increase the execution time drastically. Therefore, one needs to find good tile sizes to profit from TMM.

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

Table 5.1.: Execution times for 1024×1024 matrix multiplications

Setting	execution time [s]
Standard algorithm	11.384
TMM, tile size $i = j = k = 1$	25.683
TMM, tile size $i = j = k = 32$	2.669

two matrices using a given tile size. The selection of the fittest individuals for the next population is performed on the master.

The TMM benchmarks are executed with matrix sizes of 128×128 and 256×256. The matrix size influences the number of computations that need to be performed for a full matrix multiplication and, therefore, also influences the execution time. In contrast to ZDT-3, the matrix size has no impact on the amount of data transferred between the master and the workers. The matrices are part of the "initial data" (see chapter 4.2.3) and, hence, 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.

5.2.3. Settings

All benchmarks are executed on the cluster described in section 5.1. They 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.

Parallel benchmarks are executed with different problem sizes and different numbers of workers. Standalone sequential benchmarks, used for comparison with the parallel results in section 5.6, are executed with different problem sizes only. It makes no sense to vary the worker size for a sequential benchmark.

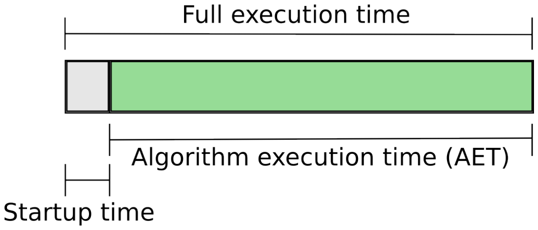

Figure 5.2.: Biohadoop execution times are composed of start up times and algorithm execution times

5.3. Biohadoop Benchmarks

The benchmark results presented in this section show the algorithm execution times (AETs) for the test problems that were parallelized using Biohadoop. It is expected that the AET of a test problem reduces when the number of parallel workers increases.

Before the presentation of the benchmark results, a definition of AET is given for clarification. Then, the benchmark results are presented. Afterwards, the output of the benchmarks, i.e., the optimization results, are checked for correctness.

5.3.1. Definition of Algorithm Execution Time

The full execution time of an algorithm running on top of Biohadoop is composed of the time Biohadoop needs to start up as a Hadoop application and the time spent in the algorithms `run` method. The time spent in the algorithms `run` method is from here on defined as AET. Figure 5.2 demonstrates this concept.

The distinction between start up time and AET is made because the main part of the start up time is spent between the application submission to Hadoop and the beginning of its execution. It is not possible to predict when an application is executed by Hadoop as it depends on different factors like the available cluster resources. To minimize the impact of this uncertainty, the following benchmark measurements are based on AET, without the application start up time.

In contrast, the definition of AET for a program that doesn't use Biohadoop is given as the execution time of the Java program. This definition applies to the standalone sequential implementations in section 5.6, which have no Hadoop startup time.

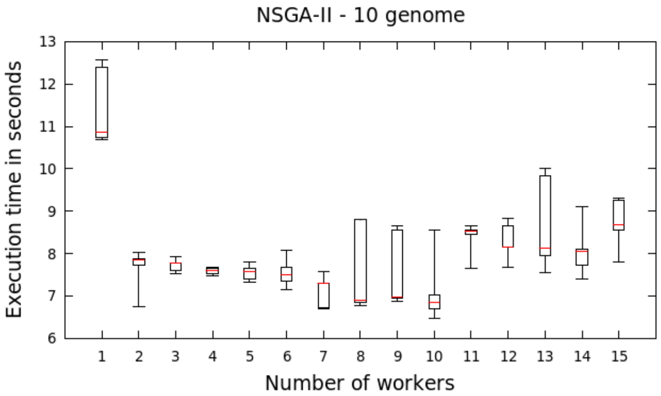

Figure 5.3.: ZDT-3 execution times for a genome size of 10

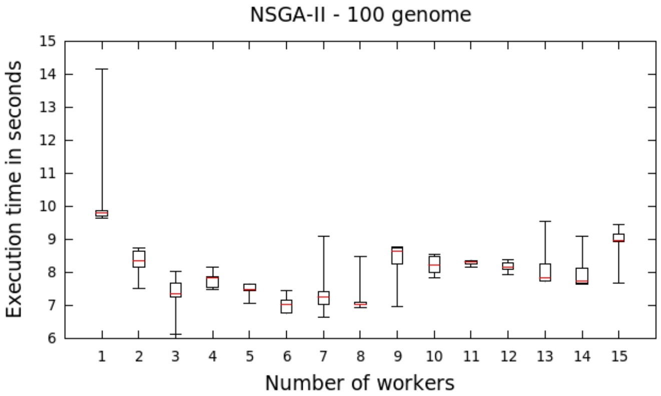

Figure 5.4.: ZDT-3 execution times for a genome size of 100

5.3.2. Benchmark results

The results presented in this section reflect the AET for the test problems of section 5.2. The test problems are parallelized using Biohadoop and its task system. The parallel parts are executed on Biohadoop workers.

The benchmarks are performed with worker sizes ranging from 1 to 15. Each benchmark is repeated five times to improve the reliability of the results. This gives a total of 300 benchmark runs for ZDT-3 (4 genome sizes × 15 worker setting × 5 repetitions) and 150 benchmark runs for TMM (2 tile sizes × 15 worker settings × 5 repetitions). The AET for the benchmarks are presented in fig. 5.3 to 5.6 for the ZDT-3 test problem. Fig. 5.7 and 5.8 show the results for the TMM benchmarks.

All benchmark results demonstrate reduced AETs when the number of workers is increased. The biggest performance gains can be found when stepping

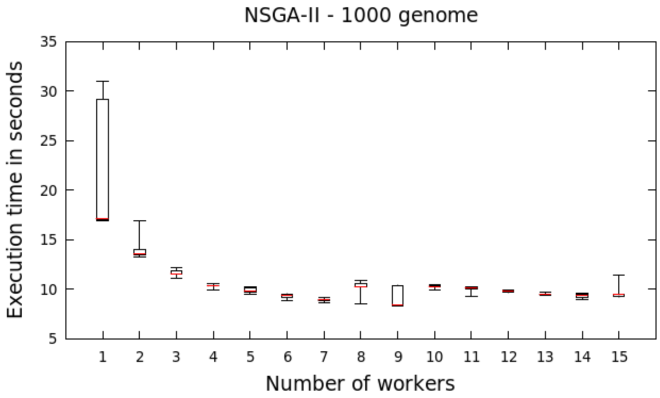

Figure 5.5.: ZDT-3 execution times for a genome size of 1000

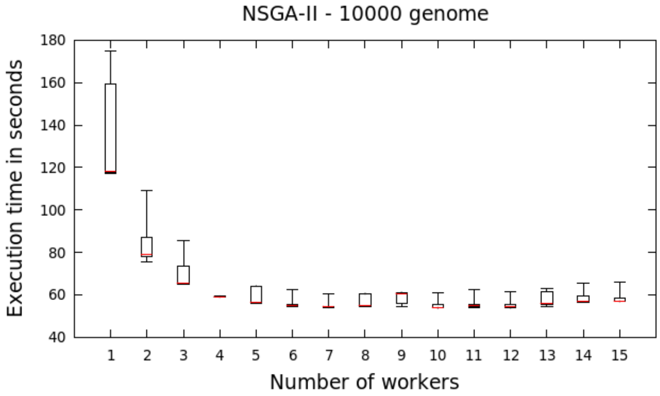

Figure 5.6.: ZDT-3 execution times for a genome size of 10000

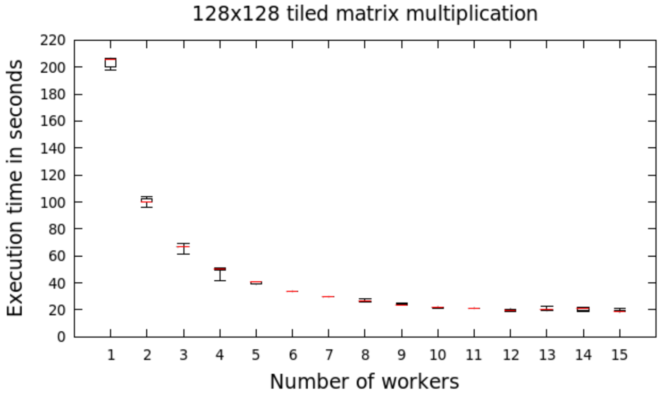

Figure 5.7.: TMM execution times for a matrix size of 128×128

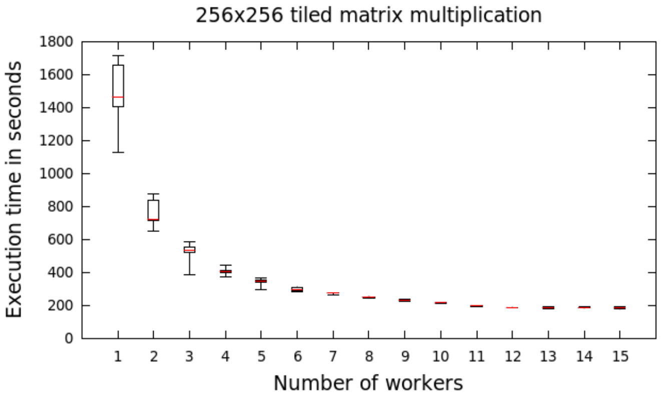

Figure 5.8.: TMM execution times for a matrix size of 256×256

Table 5.2.: Hypervolume results for ZDT-3 benchmarks

Test Problem	Hypervolume
NSGA-II, 10 genomes	0.515
NSGA-II, 100 genomes	0.458
NSGA-II, 1000 genomes	0.340
NSGA-II, 10000 genomes	0.330

from 1 to 2 workers. Further increases of parallel workers show varying success. ZDT-3 benchmarks with 10 and 100 genomes don't seem to profit from more than 2 workers. The performance for genome sizes of 1000 and 10000 increases in the best case until 8 - 9 workers, although, the performance gains are very small. The TMM benchmarks on the other hand show remarkable AET decreases when the number of workers is increased up to 12 workers.

At this point a more in-depth discussion of the benchmark results is put back on purpose, it can be found in section 5.4 and 5.5. The correctness of the optimization results must be verified first, otherwise, the test problems could execute arbitrary computations and the benchmarks would be meaningless.

5.3.3. Correctness of ZDT-3 results

In the case of ZDT-3 the task was to find an approximation to the optimal Pareto Front (OPF). The computed Pareto Fronts (CPF) are compared to the OPF using the Hypervolume (HV) indicator [14]. Table 5.2 gives the results for the indicator, where the HV is computed as the median over all benchmarks for a given genome size. Higher HV values indicate a better CPF.

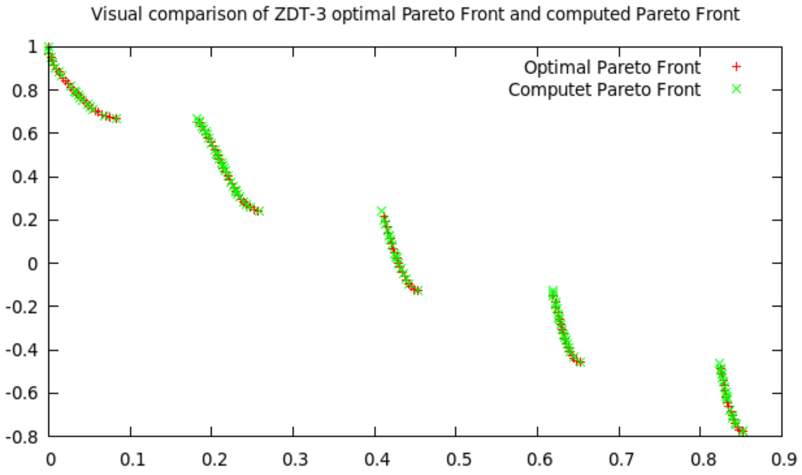

Figure 5.9.: Visual comparison of ZDT-3 optimal Pareto Front and computed Pareto Front

One can see that HV decreases with the number of genomes. It is well known that ZDT-3 optimization is harder the larger the genome is. 10 genomes produce the best results with HV = 0.515. Manual comparison of a CPF (randomly chosen from the results with 10 genomes) show high similarities with OPF, as can be seen in fig. 5.9. It is therefore concluded that the ZDT-3 optimization results are correct.

5.3.4. Correctness of TMM results

The goal of TMM optimization was to find tile sizes that reduce the computation time for a tiled matrix multiplication. Other than expected, the found tile sizes follow no recognizable pattern. The explanation is that the solution space is shallow which makes most solutions almost equally well suited. The exception are very small tile sizes with values less than 4.

The TMM optimization results were checked by executing a matrix multiplication with the optimized tile sizes and comparing the results to the execution time of the simple matrix multiplication (SMM). The results were twofold.

For 128×128 matrix sizes, SMMs were faster by 10 % to 15 %. The reason is, that the additional nested loops of TMM and the consequent increased overhead is higher than the performance gain. Another reason is that 128×128 matrices are small enough that sufficient parts of it fit into the L1 CPU caches to profit from the fast access rates.

In the case of 256×256 benchmarks, all TMM executions were faster compared to the SMM by 20 % to 30 %. Additional experiments with larger matrices show that the performance gap between SMM and TMM increases with the matrix size. For example, 512×512 matrix sizes provided about 50 % better performance for TMM. 1024×1024 matrix sizes have about 400 % better performance. Again, the conclusion is that TMM benchmarks produce correct outputs.

5.4. Performance Influencing Factors

Looking at the details of the benchmark results (presented in fig. 5.3 to 5.8) two strange behaviours can be observed that need explanation. First, the AET for a benchmark with given setting (e.g. ZDT-3, 100 genomes, one worker) varies by a large amount. This is due to the influence of YARN, discussed in section 5.4.1. Second, ZDT-3 benchmarks for genome sizes of 10 and 100 experience AET reductions only up to 2 parallel workers, although the cluster provides more resources. The reason for this has been found to be the CPU saturation on Biohadoops master, discussed in section 5.4.2. Other hard- and software influences on the benchmark results are summarized in section 5.4.3. Their impact was not quantified.

It is difficult and a lot of work to find the reasons for performance degradations and strange phenomenas. This is especially true for a distributed environment like Hadoop. Further research could therefore focus on the development of distributed tracing and profiling tools, perhaps adapted to Hadoop. This would greatly simplify performance and bottleneck evaluations. Since Hadoop gains a lot of attraction also in business, it would be a good chance to bridge the gap between science and business.

5.4.1. Influence of YARN

An interesting benchmark result is that the AETs for a given benchmark and setting vary by a large degree, e.g., ZDT-3 with 100 genomes and one worker. YARNs container placement and startup time was identified as the source of this variations.

The container placement of YARN has a big impact on AET. It defines on which machine a YARN container is allocated. If a worker container is executed on the same machine as the master container, they communicate without using the physical network. This effect brings a huge performance gain, as can be seen for example in figure 5.4. In the single worker benchmarks, 4 out of 5 benchmarks executed with both the master and worker container running on the same machine. The result was a 50 % better performance (9.761 s average) compared to the fifth benchmark (14.164 s) where the master and worker were executed on different machines. Potential research projects could focus on a Hadoop scheduler that tries to put a YARN ApplicationMaster and its containers on the same machine to reliably produce similar results.

The number of worker containers running on the same machine as the master can also have a negative effect on AET. 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 5.3 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 87.977 s and 88.014 s. 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 68.423 s on average, a difference of more than 20 %.

Beside container placement, YARN influences AET through the container startup delay. Biohadoop can not use all of its configured workers until they are started by YARN (this is true for all YARN applications that use additional containers). An example of this phenomena can be found in the ZDT-3 benchmarks with settings of 10 genomes and a single worker. In all five benchmarks for this setting, the master and worker container executed on different computers. Therefore, the results are comparable regarding the container placements. Nevertheless, the execution times vary from 10.690 s to 12.560 s, a difference of about 15 % due to delayed container allocations.

5.4.2. CPU utilization

The results for ZDT-3 benchmarks with genome sizes of 10 and 100 show no significant AET reductions for more than two workers (see fig. 5.3 and 5.4). The reason for this odd behaviour is that the CPU running Biohadoops master is fully utilized when two ore more parallel workers are used.

This was established by measuring the CPU usage of the master during the execution of the benchmarks using Javas `OperatingSystemMXBean`[1]. The measurement was performed at an interval of 1 s. Figure 5.10 shows the example CPU loads for ZDT-3 with 10 and 100 genomes executing with 3 workers.

It is arguable that the AET of ZDT-3 with 10 and 100 genomes is to short to get correct informations about the CPU usage. Therefore, the ZDT-3 benchmarks for 3 workers with 10 and 100 genomes were performed for 1000 iterations (original benchmarks iterate only 250 times). The results can be seen in fig. 5.11.

[1] `http://docs.oracle.com/javase/7/docs/jre/api/management/extension/com/sun/management/OperatingSystemMXBean.html` last access: 27.01.2015

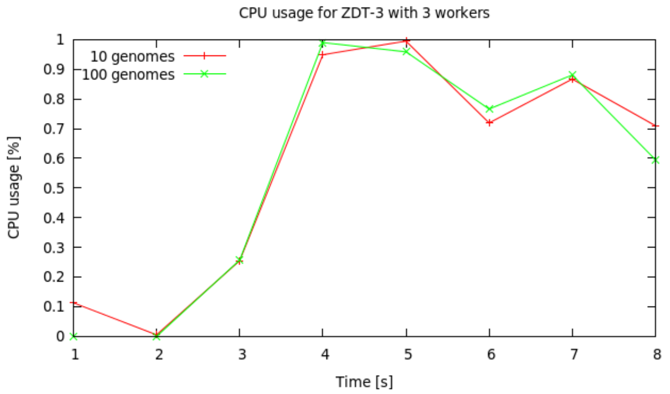

Figure 5.10.: CPU usage for ZDT-3, 10 and 100 genomes, 3 workers - 250 iterations

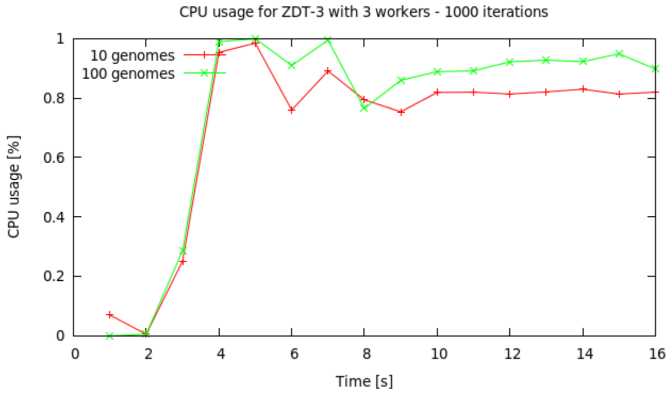

Figure 5.11.: CPU usage for ZDT-3, 10 and 100 genomes, 3 workers - 1000 iterations

They confirm the CPU saturation for ZDT-3 with 10 and 100 genomes and three and more workers.

Since the master executes the sequential parts of the algorithms and in addition has to communicate with all workers, it is clear that a saturation of the master CPU prevents further performance gains. The problem worsens when additional YARN containers run on the computer that executes the master.

5.4.3. Other influences

YARN and the CPU had the biggest effects on the benchmark results. In the following, other possible influencing factors are summarized. Their impact on the benchmark results is not quantified, as their correct evaluation is not trivial but very work intensive. As already mentioned, further research projects could develop tools to simplify this process.

Network : The network influences the benchmark results, as the master and its workers exchange messages through it. This influence is apparent in cases where YARN places its containers on the same computer, which can result in greatly reduced AETs. Nevertheless, the network influence is consistent: throughput and latency don't change over the time on the used test cluster.

RAM : All benchmarks start with 256 MB 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[2] (delivered with Java) for the ZDT-3 benchmark with 10000 genomes and the TMM with a matrix size of 256×256. The memory usage for the master container is at about 100 MB to 150 MB. The memory usage for a worker is even lower and lies in the range of 5 MB to 30 MB. The conclusion is, that enough RAM was available during the benchmarks. To few RAM would have had significant impact on the performance.

CPU caches : The CPU caches show their biggest influence during the TMM benchmarks. The workers execute matrix multiplications using provided tile sizes. If the tiles are small enough they fit into the L1 CPU cache which is the fastest memory available to a processor. Increased performance is the result. If the tiles don't fit into L1 cache, they need to be retrieved from L2 cache or even worse from main memory, which introduces latencies. Since TMM executes matrix multiplications to evaluate the fitness of an individual, the tile sizes affect the fitness evaluation of a single individual, and the whole runtime of the benchmark.

Java Java Just in Time Compiler (JIT) : JIT[3] compiles Java byte code into machine executable code that can be executed faster. During the benchmarks, JIT was set to optimize code parts that are executed more than 10000 times (this is the standard setting for the used Java Server runtime[4]). Small benchmarks, like ZDT-3 with 10 genomes, hit this threshold later in their execution. Therefore, the share of executed un-optimized code is larger compared to large benchmarks, e.g., ZDT-3 with 10000 genomes. This problem worsens with increasing numbers of workers.

[2] `http://visualvm.java.net/` last access: 26.01.2015

[3] `http://en.wikipedia.org/wiki/Just-in-time_compilation` last access: 26.01.2015

[4] `http://en.wikipedia.org/wiki/HotSpot` last access: 26.01.2015

Operating system (OS) : The OS provides the basis for application execution. It provides also the interfaces for hardware usage, e.g. network communication. Therefore, it has large influence on executed applications. Using the right drivers and OS settings, e.g., for buffers, influence the performance of running applications significantly.

Building blocks : Biohadoop uses Netty and Kryo for the communication between the master and its workers. Those libraries can be configured in many different ways. Again, the right configuration can provide performance increases or degradations.

5.5. Speedup

Speedup is a metric that tells how many times a parallel version of an algorithm (or system) runs faster compared to the sequential version. In the following, the maximum achievable speedups and the experimental speedup, i.e., the speedup achieved in the benchmarks, are presented in section 5.5.1 and 5.5.2, respectively.

5.5.1. Maximum achievable Speedup

The max. achievable speedup (S_{max}) for a given problem is the best-case result achievable through parallelization. It is computed using Amdahl's law [15]:

$$S_{max} = T/(T - t_p) \tag{5.1}$$

In this formula, T is the sequential execution time of an algorithm. During this sequential execution it is also measured how much time is spent in code parts that could be executed in parallel. This amount of time is defined as t_p.

The values for T and t_p are taken from the benchmark executions with a single worker. Since the measured AET for a given benchmark and one worker differs largely (e.g. ZDT-3 with 100 genomes), lower and upper max. achievable speedup boundaries are computed. The lower bound is computed using the worst sequential-to-parallel execution time ratio of a sequential benchmark run. The upper bound reflects the best sequential-to-parallel ratio. This proceeding gives more details over the achievable speedups.

Table 5.3 presents the lower and upper bounds for the max. achievable speedups based on the benchmark results. The calculations show that ZDT-3 is generally not well suited for parallelization. In the best case with 1000 workers its performance can be increased by a factor of about 20 (upper bound). In the worst case with 10 genomes the performance can be increased by a factor of 5 (lower bound). The reason for the small achievable speedups is the bad

Table 5.3.: Lower and upper bounds for max. achievable speedups

Test Problem	Lower Achievable Speedup Boundary	Upper Achievable Speedup Boundary
NSGA-II, 10 genomes	4.816	11.046
NSGA-II, 100 genomes	5.228	9.398
NSGA-II, 1000 genomes	9.438	20.275
NSGA-II, 10000 genomes	8.861	17.081
TMM, 128×128	63.649	134.404
TMM, 256×256	139.685	271.925

sequential-to-parallel execution time ratio. ZDT-3 as fitness function is barely compute intense. As the fitness function is usually (but not in this case) the most compute intense part of an optimization, this leads to the mentioned bad sequential-to-parallel execution time ratio.

TMM on the other side shows better results. It demonstrates that the test problem is well suited for parallelization. This is not surprising, as the parallel part in form of the fitness evaluation consists of a matrix multiplication which is known to be compute intensive. Therefore, the sequential-to-parallel execution time ratio is advantageous and opens the possibility to achieve good experimental speedups.

5.5.2. Experimental Speedup

To compute the experimental speedup S for a given benchmark, its sequential (T_S) and parallel (T_P) execution times are needed. T_S is the execution time for a benchmark with a single worker. T_P is the execution time for a benchmark with P workers. S is then calculated using the following formula:

$$S = T_S/T_P \tag{5.2}$$

Since the execution times for a given benchmark and setting vary often in a broad range, the lower and upper speedup boundaries for a given benchmark are computed. This is the same proceeding used for the computation of the max. achievable speedups. The boundaries are computed by using the min. and max. execution times for T_S and T_P, respectively. For example, min. T_S and max. T_P of a given benchmark and setting are used to compute the lower speedup boundary. Max. T_S and min. T_P are used to compute the upper speedup boundary.

Fig. 5.12 shows the lower speedup bounds for all benchmarks, fig. 5.13 the upper speedup bounds. The figures demonstrate that the speedups for ZDT-

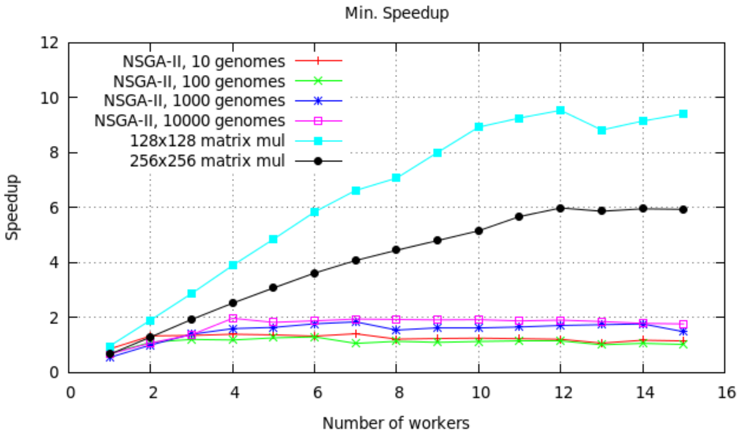

Figure 5.12.: Lower speedup boundaries for ZDT-3 and TMM benchmarks

3 benchmarks are small. Even when looking at the best case of upper bound speedups it is clear that ZDT-3 takes little profit of the parallelization. The lower bounds of the experimental speedups show even poorer results. This comes at no surprise, since the max. achievable speedups computed in section 5.5.1 are already small and it is very unlikely to reach them in practice.

In contrast, the TMM results show good speedup behaviours. The benchmarks with a matrix size of 128×128 demonstrate almost linear performance increases, which tend to be super-linear for the upper bound speedups. Super-linear speedups are usually suspicious. In this case they can be explained by the way the boundaries of the experimental speedups are computed. Nevertheless, also the lower bounds for TMM benchmarks with a matrix size of 128×128 show almost linear increases. All speedups of all TMM benchmarks reach their maximum when 10 to 12 workers are used. This observation matches with the fact, that the whole cluster provides 12 cores.

The reason for the worse speedup of 256×256 TMM compared to the 128×128 version lies in the bigger solution space of 256×256 TMM. It is more likely to find bad performing tile sizes for bigger matrices. The L1 cache sizes are limited and to big tile sizes affect the performance negatively.

The conclusion of the speedup results is that the parallelized search for values that minimize the ZDT-3 function is not well suited for Biohadoop — at least for the used implementation. More advanced implementations may influence the sequential-to-parallel ratio and achieve better results. A different, more compute intense fitness function may also entail better speedup results. TMM on the other side provides good speedups and decrease the AET by a large amount. The parallelized search for optimal tile sizes using Biohadoop is favorable.

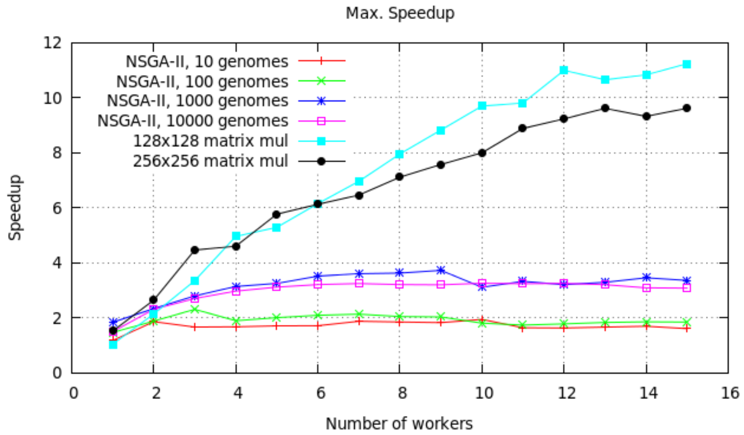

Figure 5.13.: Upper speedup boundaries for ZDT-3 and TMM benchmarks

5.6. Comparison to Standalone Implementations

Standalone, sequential benchmarks (SB) of the test problems in section 5.2 are executed to determine if the algorithm parallelization using Biohadoop is advantageous. SB are written in Java and executed on a single cluster machine. They use the same algorithm implementations and settings as the parallel versions, but run without Biohadoop and its task system. This entails that they perform no network communication at all.

Table 5.4 shows the fastest standalone sequential execution times (T_S), together with the fastest execution times achieved with parallel implementations (T_P). It provides also information about the speedup gained through parallelization. This speedup is computed the same way as in section 5.5.2. The speedup is abbreviated with letter G (gain) for better distinction to the previous results:

$$G = T_S/T_P \tag{5.3}$$

If G is smaller than 1, the parallelization has a negative effect and should be avoided. In this cases, the parallel execution takes longer than the sequential execution. Values for G that are bigger than 1 are favorable, since they provide reduced execution times.

The ZDT-3 benchmarks for genome sizes up to 1000 show that the parallelization has a negative impact, the parallel execution times are bigger than the serial execution times. In this cases, it is better to stick with the serial implementation. ZDT-3 with a genome size of 10000 shows that the execution of parallel versions provide advantages in terms of execution time, although the performance gains are not very big.

Table 5.4.: Execution times for *SB*

Test Problem	Sequential [s]	Parallel [s]	Performance gain
NSGA-II, 10 genomes	2.852	6.461	0.441
NSGA-II, 100 genomes	2.956	6.128	0.482
NSGA-II, 1000 genomes	7.673	8.330	0.921
NSGA-II, 10000 genomes	71.390	53.706	1.329
TMM, 128×128	132.066	18.386	7.183
TMM, 256×256	1500.705	178.158	8.423

The situation is completely different for the TMM benchmarks. The parallel implementations provide large benefits compared to the sequential versions.

Chapter 6.

Conclusions

This thesis presented Biohadoop, a new framework to build bio-inspired optimization techniques executable on Apache Hadoop. Two different GAs were implemented on top of Biohadoop. Their performance was evaluated on a Hadoop cluster with 6 dual-core computers that were connected to a 1 Gb Ethernet network.

The results show that algorithms implemented with Biohadoop have the potential to scale efficiently if the parallel part of the problem dominates the whole execution time. An example of such a problem is the tiled matrix multiplication (TMM). It provided speedups of up to 10 when compared to the execution with less resources (one worker). ZDT-3 as the other benchmark exposed a behavior that lead to poor speedups of about 2. The problem with this benchmark was that the parallel running parts were very short in terms of computational time. The sequential time and communication overhead dominated the whole runtime.

The execution time comparison with standalone, sequential implementations demonstrated further advantages of Biohadoop. TMM provided in this case speedups of about 8. In contrast, the performance of ZDT-3 was worse to the point that the parallel execution took longer than the sequential version.

In conclusion, the evaluation results show that Biohadoop is a useful framework for the implementation of bio-inspired optimization techniques. Given that an algorithm is well suited for parallelization, Biohadoop offers an easy way for its implementation. The evaluation demonstrated also that such parallelized algorithms are able to perform very well on Hadoop.

Appendix A.

A.1. How to 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` environment 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 section 4.5 on more information about how to configure Biohadoop). Those paths have to point to valid locations of an accessible HDFS file system.

Biohadoop was developed using Maven.[1] Therefore, it is rather easy to get all the needed libraries, since they are declared as dependencies. The source code for Biohadoop can be found on GitHub: `https://github.com/gappc/biohadoop/`. By invoking the following command on the projects root folder, all dependencies are accumulated and put to the sub folder `target/dependency`:

[1] `http://maven.apache.org/` last access: 11.09.2014

```
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 `includePaths` configuration option mentioned above.

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, therefore, in appendix A.2 a simple method can be found, to use a pre-build Hadoop environment. The only dependency which this environment has is Docker.[2] Docker is a simple and lightweight runtime for virtual containers available on all major operating systems.

A.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 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, endpoints and workers. After all of the algorithms have terminated, either because they have finished their computation or because of 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

[2] `https://www.docker.com/` last access: 11.09.2014

Biohadoop. 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.

A.1.2. Hadoop Environment

To start Biohadoop in a Hadoop environment (for example in the one provided in appendix A.2), all needed libraries and configuration files must be present in the HDFS file system. In addition, Biohadoops jar file, i.e., the compiled library, 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. Furthermore, 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 correctly.

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.

A.2. Pre-build Hadoop Environment using Docker

This section provides a method to run a pre-configured Hadoop environment using Docker. Docker uses its so called Dockerfiles for its configuration. The solution presented here installs Apache Hadoop 2.5.0, Apache Oozie 4.0.1 and Apache ZooKeeper 3.4.6 as a cluster environment.

A.2.1. Build the Hadoop Environment

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

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

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

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

A.2.2. Run Hadoop

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

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

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

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

```
jps
```

On the master node, it should output:

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

On the slave nodes it should output:

- DataNode
- NodeManager

A.2.3. Stopping Hadoop

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

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

A.2.4. SSH access

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

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

[4] 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.

[5] 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.

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

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] E. Zitzler and L. Thiele, “Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach,” *Evolutionary Computation, IEEE Transactions on*, vol. 3, no. 4, pp. 257–271, 1999.

[15] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, ACM, 1967.