SPIS TREŚCI

SPIS RYSUNKÓW

SPIS LISTINGÓW

SPIS TABEL

1. BACKGROUND

1.1. DOMAIN OVERVIEW

Starting from the definition, an evolutionary algorithm (EA) is a subset of evolutionary computation, a generic population-based metaheuristic optimization algorithm. An EA uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the quality of the solutions. Evolution of the population then takes place after the repeated application of the above operators.

Evolutionary algorithms often perform well approximating solutions to all types of problems because they ideally do not make any assumption about the underlying fitness landscape. Techniques from evolutionary algorithms applied to the modeling of biological evolution are generally limited to explorations of microevolutionary processes and planning models based upon cellular processes. In most real applications of EAs, computational complexity is a prohibiting factor. In fact, this computational complexity is due to fitness function evaluation. Fitness approximation is one of the solutions to overcome this difficulty. However, seemingly simple EA can solve often complex problems[citation needed]; therefore, there may be no direct link between algorithm complexity and problem complexity.

Overall the optimization methods can be divided generally into two groups: the gradient methods, that operate on a single potential solution and look for some improvements in its neighborhood, and global optimization techniques – represented here by so called evolutionary methods – that maintain large sets (populations) of potential solutions and apply some recombination and selection operators on them. During the last decades, evolutionary methods have received a considerable attraction and have experienced a rapid development. Main paradigms are: genetic algorithm (binary or real coded), augmented simulated annealing (binary or real coded), evolution strategy and differential evolution. Still, each of these methods has many possible improvements. In this theses the author is concentrating on subfamily of evolution methods named genetic algorithms. It is one of the most popular representative of evolutionary methods and serves as a great example, because it allows to observe and study the properties of the whole family.

1.1.1. How evolutionary algorithms are used nowadays?

Typically, optimization methods arising in engineering design problems are computationally demanding because they require evaluation of a quite complicated objective

function many times for different potential solutions. Moreover, the objective function is often multi-modal, non-smooth or even discontinuous, which means that traditional, gradient-based optimization algorithms fail and global optimization techniques, which generally need even a larger number of function calls, must be employed. Fortunately, the rapid development of computational technologies and hardware components allows us to treat these problems within a reasonable time.

Genetic algorithms have been applied in science, engineering, business and social sciences. Number of scientists has already solved many engineering problems using genetic algorithms. GA concepts can be applied to the engineering problem such as optimization of gas pipeline systems. Another important current area is structure optimization. The main objective in this problem is to minimize the weight of the structure subjected to maximum and minimum stress constrains on each member. GA is also used in medical imaging system. The GA is used to perform image registration as a part of larger digital subtraction angiographies. It can be found that GAs can be used over a wide range of applications [6]. Genetic algorithms can also be applied to production planning, air traffic problems, automobile, signal processing, communication networks, environmental engineering and so on. There is a lot of examples discussed in [1] from music, art in general, architecture and engineering design.

1.1.2. Disadvantages

As every other technique, genetic algorithm has its flaws. In this theses the author tries to attract attention to the two most important disadvantage of genetic algorithms, due to subjective opinion. For each of them is proposed an alternative solution, which will later take place in functional requirements of the thesis work.

1. **Implementation complexity**
   There are many problems of different types, which may me optimized by genetic algorithms. The downside is that for each of those problems there is a need of specific implementation, which is at least time consuming. Even though the low level details, which depend on the problem GA is applied to, will change, the high level abstraction of genetic algorithm workflow remains the same. This makes it possible to implement generic representation of genetic algorithm, which may be reusable among variety of application and be independent from the low level details.

2. **Computation cost**
   Genetic algorithms tend to be very time consuming. This is strictly related to its computation model, as the function, used in GA, are usually very demanding. There are a lot of approaches to decrease the time it takes to produce desirable result by applying cached (or partly cached) fitness function or use its approximation, avoiding

the large-size data and using many low-level optimization in different parts of the algorithm. All of these improvements are beneficial but take time to implement.
An additional method for performance optimization, available right out of the box is workflow parallelization. This should allow users to use full capabilities of hardware in their use and also be transparent for users, saving them a lot of efforts and time.

1.2. SOLUTION PROPOSAL

A solution for described flaws would be a tool, which allows to integrate a high-performance implementation of the genetic algorithm into user application and use it for the arbitrary optimization problem with minimum changes required to the existing codebase. Additionally, this should be a convenient toolbox for theoretical and practical problems which might be resolved using genetic algorithms, providing a ready-made implementation of the most popular algorithms with extra features, which save users from implementing one themselves. For a solution to be considered as fulfilled, it is necessary to meet both functional and non-functional requirements predefined in advance.

1.2.1. Non-functional requirements

1. **Java Virtual Machine**
   This platform was chosen, because of its popularity and open nature. Java virtual machine is supported by many languages, what allows a user to pick a specific one, which is the best for a given circumstances. JVM is also vastly used and run on over 3 billion devices[1]. It takes the responsibility of platform organizing hardware environment, allowing the application designers to be platform agnostic. It is very convenient in terms of program correctness - meaning that if the program runs on one machine, it is highly probable to run as expected on the other machine satisfying minimum requirements. Sun's famous slogan *"Write once, run everywhere"* is only valid due to Java virtual machine and its ability to run standard bytecode.

2. **Open implementation**
   The implementation of the tool has to be open to its users, allowing to view and expand source code to the specific needs.

3. **Modular architecture**
   The architecture solution used to build the tool has contain modularity among its fundamentals. This means, that allowing different features, its components have to be grouped according to their functions, so the users, who are interested in a small part of provided functionality wouldn't be forced to use all capabilities of the tool.

[1] According to official information provided by Oracle Corporation

1.2.2. Functional requirements

1. **Limiting the evolution in terms of time and/or number of iterations**
   This is basic requirement for every genetic algorithm implementation. Evolution process takes time and in order to benefit from it, user must have access to the evolved version of population. Number of iterations corresponds to the number of evolution cycles performed on the population, when every genetic operator is applied to it. On the other hand, the actual time of the evolution may be changed even in terms of the same number of iterations, as it strongly depends on performance of genetic operators and fitness function, so in some cases time may be the only valid criterion, which decides that evolution process should be stopped.

2. **Control of the best individual through evolution**
   During the evolution process, due to the chosen selection, crossover or mutation strategies, even the fittest individual may not end in the next population. As for business needs it is important to get the best solution ever discovered by algorithm, it is useful to store reference to that candidate separately, as well as its fitness value to avoid unnecessary re-computation.

3. **Fitness values caching**
   In genetic algorithms, the computation of the fitness function comes with the largest computational load for the algorithm. Each population generation is composed of individuals who are formed from previous generation via cloning, crossover, or mutation. For each new individual it necessary to assign a fitness function value, which determines how good a candidate is. It is also common for the individual to migrate to the next population without a change or, less common, a crossover of two different individuals may produce a genotype that already has been discovered earlier. If this occurs frequently, it is unprofitable to compute the fitness value all over again. As a solution one may consider caching the results of fitness computation for later re-usage. Although it noticeably increases the speed of algorithm, the cache tends to grow very fast, so the choice of data structure for this purposes is extremely important.

4. **Parallel evolution processing**
   Genetic algorithms are generally known as a time-consuming technique, as it is often used with complex, high-dimensional problems. Evolution cycle may take up to hours and days of continuous computing, depending on the parameters of the algorithm and hardware in use, which comes with a great cost and very low flexibility, as with every mistake or change was done to algorithm implementation, evolution needs to be re-executed from scratch. On the other hand, even small upgrades in speed may result as a major cost cut.

For this reason, it is highly important to use the maximum of the given resources, avoiding unreasonable workloads. It may be achieved through the computation parallelization between multiple virtual CPU cores within a single machine.

5. **Asynchronous evaluation of fitness value**
   The solution must allow to compute fitness values on different machine or even cluster of machines, as well as using different low-level programming language or some specific optimization, with no need to switch the entire application to another platform. This feature will make great impact in case of cloud computing solutions, as fitness evaluation may be entirely isolated from the rest of application.

6. **Generic representation of individual**
   There is a wide set of problems which might be resolved using genetic algorithms. Still, this optimization technique is rarely used in business applications and mainly considered as a theoretical solution. One of the reasons for such phenomenon is the difficulty which comes with a try to use genetic algorithm optimization without a previous plan to do so. The process of reshaping the existing codebase in such way, so typical genetic algorithm could be applying often demands big costs and high level of understanding GA workflow and its possibilities.
   Depending on the problem a user is trying to solve, a standard representation of each candidate solution would be an array of bits, integers or real numbers. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple crossover operations. For the purposes of genetic programming and evolutionary programming tree-like and graph-form representation are used respectively, which causes the necessity in different implementations for each problem.
   A solution to this problem is an implementation of the genetic algorithm, which is able to work on generic types, which represents a candidate solution for the applied problem. Due to this property, it is possible to reuse the implementation for different problems and create a generic tool, which is decoupled from the problem it is applied to.

7. **Evolution in distributed environment**
   As already mentioned, parallelization is a big part of speed optimization for genetic algorithms. While local parallelization is useful and convenient for medium- and small-sized problems, working with complex use cases often demands the resources, which is may not fit into one machine. Distributed environment is a next step of parallelization, when evolution process is split between different machines, each computing its own part of the workload. This approach enables horizontal scaling, when each added machine increases performance of the algorithm.

8. **On-demand evolution**

Genetic algorithm optimization is generally used for finding the best solution for a defined problem, which is the fittest candidate to the provided fitness function. It means, that the best solution is defined relative to all the solutions assigned earlier and cannot be proved as the optimal one, even when it is. But when applying to a nontrivial problem, finding the optimal solution is not always cost-effective, as it may take up to exponential time. For this reason, it is common to compute a 'good enough' solution in reasonable time. Depending on the business reasons, this time varies so sometimes it is possible to wait until the better solution is computed and sometimes it is not. For instance, if the application which uses genetic algorithm has a real-time interface and its user is interested in gaining a solution for the particular case in the range of seconds, it is justifiable to provide a solution which may not be optimal but is close to it.

On the other hand, when a user is ready to allow the application to take its time in order to provide a better result, it has to still be possible to resume the evolution. Again the application should not limit users in such important cases, as the end user needs may vary depending on different circumstances, which may not be predicted in advance. The evolution of a solution to a given problem should be available on-demand, without forcing unreasonable resource uses, and adjust to a specific case as much as possible.

2. EXISTING SOLUTIONS OVERVIEW

Earlier addressed problems are well known in world, so there already were tries to create similar solutions. While searching for alternative solutions non-functional requirements were taken into account:

- a solution had to be written or at least executed on JVM platform
- a solution had to present users its open code (not necessarily be open-source)

As a result of the search only two nearly comprehensive solutions were found:

1. Java Jenetics library
2. Apache.math.genetics package

After these candidates were precisely studied, here is published a high level overview for both of them.

2.1. APACHE.MATH.GENETICS

Apache.math.genetics is a package under Apache Software Foundation for Java programming language. It covers the basic components of genetic algorithm implementation such as genetic algorithm, fitness function, stop condition, genetic operator.

GeneticAlgorithm provides an execution framework for Genetic Algorithms (GA). Populations, consisting of Chromosomes are evolved by the GeneticAlgorithm until a StoppingCondition is reached. Evolution is determined by SelectionPolicy, MutationPolicy and Fitness. [8]

However the amount of topic it covers is relatively small to the scope, described earlier. In terms of predefined functional requirements this solutions meet only small part of them (2, 1) and thus can not be considered as accomplished solution.

2.2. JAVA JENETICS

Java Jenetics - is an advanced Genetic Algorithm, Evolutionary Algorithm and Genetic Programming library, respectively, written in modern day Java. [9]. This is more comprehensive toolbox, then one described earlier, containing a lot of different features and active community. The main advantage of this library is parallelization of fitness evaluation, by splitting a population into number of batches and proceeding them in parallel. It is

worth mentioning that Jenetics use Java Streams API as a result of evolution, which is considerably good solution, taking into account the background of evolutionary algorithms.

Still, being a reacher candidate it misses some crucial functional requirements. For example, the parallelization process cannot be expanded to cluster and is locked within one instance of JVM 7. Asynchronous fitness evaluation is possible only with blocking calls 5, which negates all the benefit of such solution. In addition, the problems which may be used with this implementation of GA are limited to represent a solution as an array of bits, real or integer numbers, failing one more crucial functional requirement 6.

Summarizing, even though Java Jenetics may be found usable for some subset of problems, it is not suitable as solution for the requirements of this theses.

2.3. CONCLUSIONS OF OVERVIEW

After performing comparison and overview of the existing solution, which could potentially met created requirements, it is clear, that there was not found a comprehensive solution, which could be considered as fulfilled from the previously stated points.

3. PRESENTATION OF SOLUTION

3.1. IMPLEMENTATION OVERVIEW

The solution is presented as an open-source library, that provides the major part of genetic algorithm implementation, which can be shared between different application. Base algorithm consist of a pipeline separated by three stages: selection, crossover and mutation, which evolves the population of individuals, increasing average level of adjustment and finding the best individual according to provided fitness function; preimplemented most popular strategies of genetic selection, crossover and mutation, with possibility of extending with the additional ones for specific reasons and a set of platforms which allow to run the pipeline in a different ways, best suited for existing hardware. The library is supplied with key features, basing on the concepts described earlier, wrapping all the components into sufficient framework.

3.1.1. Technology stack

Scala programming language

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. Its name stands for "scalable language." The language is so named because it was designed to grow with the demands of its users. It can be applied to a wide range of programming tasks, from writing small scripts to building large systems. Scala is easy to get into. It runs on the standard Java platform and interoperates seamlessly with all Java libraries. It's quite a good language for writing scripts that pull together Java components. But it can apply its strengths even more when used for building large systems and frameworks of reusable components. [3]

Technically, Scala is a blend of object-oriented and functional programming concepts in a statically typed language. The fusion of object-oriented and functional programming shows up in many different aspects of Scala; it is probably more pervasive than in any other widely used language. The two programming styles have complementary strengths when it comes to scalability. Scala's functional programming constructs make it easy to build interesting things quickly from simple parts. Its object-oriented constructs make it easy to structure larger systems and adapt them to new demands. The combination of both styles in ScaalMatei Zaharia, "Spark: Cluster Computing with Working Sets makes it possible to express new kinds of programming patterns and component abstractions. It also leads to a legible and concise programming style [3]

Due to its programming style and variety of features Scala serves a great tool for developing this library. A combination of paradigms is extremely useful in this case and was crucial during selection of programming language. Both of the programming styles were heavily used in order to gain the best possible result:

- Object-oriented style
  Scala is an object-oriented language in pure form: every value is an object and every operation is a method call. [3] It provides modularity and maintainability: all application is split into a self-sufficient containers, which store the relevant data and provide available operations, that may be performed on its data. These containers may be used as the assembling parts for another container, passed as function parameter and returned as function result. Such approach allows to keep track of the growing codebase, aligning its components into domain hierarchies and letting its parts to communicate with one another.
- Functional style
  Functional programming provides expressiveness, composability and conciseness. Building the library with mathematical-like functions makes it more understandable, replacing the mutable state from the application. As the result, it becomes clear, unequivocal and may be safely used in multi-thread environment. One of the most important assets, which comes from functional programming, is scalability. A well constructed application may easily be enlarged for user needs to handle a growing amount of work; it is more adaptive to the changing needs and the performance of such system improves proportionally to the additional hardware.

Parallel collections

Encouraging usage of immutable objects, from version 2.8 Scala provides new collection API, which contains parallel collections package as part of it. Parallel collections were included in the Scala standard library in an effort to facilitate parallel programming by sparing users from low-level parallelization details, meanwhile providing them with a familiar and simple high-level abstraction. The idea is simple– collections are a well-understood and frequently-used programming abstraction. And given their regularity, they're able to be efficiently parallelized, transparently. By allowing a user to "swap out" sequential collections for ones that are operated on in parallel, Scala's parallel collections take a large step forward in enabling parallelism to be easily brought into more code. [5]

How parallel collections are built? What may be acquired with Scala parallel collections?

Apache Spark

Apache Spark is a high-performance, general-purpose distributed computing system that has become the most active Apache open source project, with more than 1,000 active contributors. Spark enables users to process large quantities of data, beyond what can fit

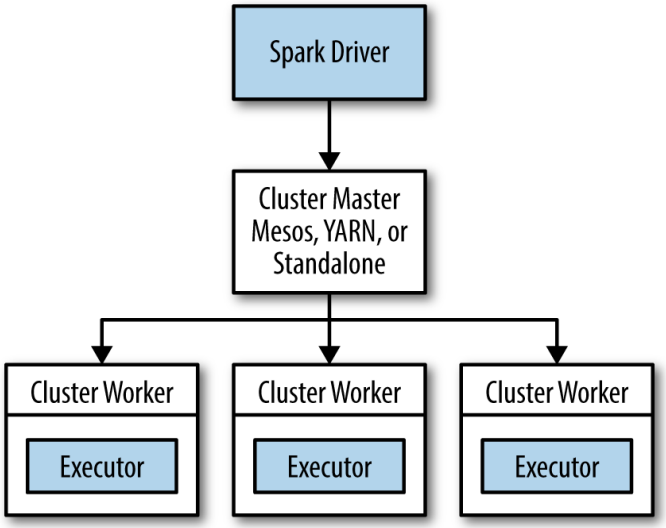
on a single machine, with a high-level, relatively easy-to-use API. Spark's design and interface are unique, and it is one of the fastest systems of its kind. Uniquely, Spark allows to write the logic of data transformations and machine learning algorithms in a way that is parallelizable, but relatively system agnostic. So it is often possible to write computations that are fast for distributed storage systems of varying kind and size.

On the generality side, Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming. By supporting these workloads in the same engine, Spark makes it easy and inexpensive to combine different processing types, which is often necessary in production data analysis pipelines.[2]

Spark can run over a variety of cluster managers to access the machines in a cluster. The easiest way is to run Spark by itself on a set of machines. For this purpose Spark comes with built-in Standalone mode. Spark's Standalone manager offers a simple way to run applications on a cluster. It consists of a master and multiple workers, each with a configured amount of memory and CPU cores. When application is submitted, one can choose how much memory its executors will use, as well as the total number of cores across all executors.

Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more. Spark Core is also home to the API that defines resilient distributed data-sets (RDDs), which are Spark's main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. Spark Core provides many APIs for building and manipulating these collections. RDDs offer two types of operations: transformations and actions. Transformations construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate. Actions are operations, which produce different result than the transformation, like collecting the elements into a local collection on master node or counting the number of elements in RDD. Transformations and actions are different because of the way Spark computes RDDs. Although new RDDs can be defined in any time, Spark computes them only in a lazy fashion — that is, the first time they are used in an action. The transformations, on the other hand, are scheduled to a DAG (directed acyclic graph) of computations. This approach has a lot of advantages, among them: effective fault tolerance and ability to make a lot of optimization decisions before actually running operations. This would not be possible if it executed computations as soon as it got it.

In distributed mode, Spark uses a master/slave architecture with one central coordinator and many distributed workers. The central coordinator is called the driver. The driver communicates with a potentially large number of distributed workers called executors. The driver runs in its own Java process and each executor is a separate Java process. A



Pic. 3.1. The components of a distributed Spark application [2].

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Pic. 3.2. Persistence levels from `org.apache.spark.storage.StorageLevel` and `pyspark.StorageLevel` [2].

driver and its executors are together termed a Spark application. Then a Spark application is launched on a set of machines using early described cluster manager.

This architectures allows to distribute the data between different executor nodes and perform operations on it locally until the shuffling is needed. Only then the data is shared between different nodes and the network communication is handled. In a distributed program, communication is very expensive, so laying out data to minimize network traffic can greatly improve performance. Much like how a single-node program needs to choose the right data structure for a collection of records, Spark programs can choose to control their RDDs' partitioning to reduce communication.

In order to be able to transforms large sets of data without redundant re-computation Spark provides data persistence. When the Spark is asked to persist an RDD, the nodes that compute the RDD store their partitions. Spark has many levels of persistence to choose

from based on what the goals are. The list of storage levels with performance comparison may is shown on 3.2. Combining different persistence modes, an executor can effectively share the data between its disk and RAM.

Akka Streams

Akka Streams is a library to process and transfer a sequence of elements using bounded buffer space. This latter property is what is usually referred as boundedness and it is the defining feature of Akka Streams. Translated to everyday terms it is possible to express a chain of processing entities, each executing independently (and possibly concurrently) from the others while only buffering a limited number of elements at any given time.

The purpose is to offer an intuitive and safe way to formulate stream processing setups such that they may be executed efficiently later and with bounded resource usage. In order to achieve this streams need to be able to limit the buffering that they employ, they need to be able to slow down producers if the consumers cannot keep up. This feature is called back-pressure and is at the core of the Reactive Streams initiative of which Akka is a founding member.

Unlike heavier "streaming data processing" frameworks, Akka Streams are not "deployed" nor automatically distributed. Akka stream refs are, as the name implies, references to existing parts of a stream, and can be used to create a distributed processing framework or introduce such capabilities in specific parts of your application.

Stream refs are well suited for any system in which you need to send messages between nodes and need to do so in a flow-controlled fashion. Typical examples include sending work requests to worker nodes, as fast as possible, but not faster than the worker node can process them, or sending data elements which the downstream may be slow at processing. [7]

Using reactive streams the first limitation of the pipeline is downstream, which implicitly asks upstreams for the data. With this in mind the upstream can be unbounded by the definition and emit data as long as there is someone, who is interested in it. If there is no one asking for new commits the stream remains inactive without resources consumption.

This is a great candidate for representation an evolution flow in genetic algorithms as primarily it is unlimited, but may paused, slowed or stopped in any given moment, basing on some external data (like time or number of iterations) or state of the stream itself (the data it is committing).

3.1.2. Development tools

For convenient and easier application development process, basing on personal experience and conducted research a list of software development tool were chosen to be used during the implementation of thesis application.

- Intellij IDEA

Popular integrated development environment (IDE) provided by JetBrains inc. originally designed for Java projects. With its popularity growth and open for user created plugins architecture, there was added support for other mainstream languages like Scala, Python etc. This is professional development tool, so it also provides built-in support for integration with other tools, like a number of version-control systems (VCS), building tools, automatic JavaDoc generation tools, static code analysis tools and many others. Using this IDE provides the whole ecosystem of development tools and practices.
For the purpose of thesis application development Intellij IDEA Ultimate version was used, provided for free by student support license.

- Git
  Version-control system for tracking changes in source code and file system itself used by the large group of software developers. This is free and open-source tool, providing a wide range of features, like cheap local branching, convenient staging areas and multiple workflows. This tool helped to keep different versions of application in comfortable and convenient way.
  For the whole stage of developing there were two main branches: `master` and `dev`. Master branch was used for stable releases of application keeping the quality of the product on high level. Developing branch was used for improving the system by implementing new functionalities and was periodically merged to master branch. For the purpose of implementing each feature or task, new branch with corresponding name was created and merged to development branch after its finish.

- ScalaStyle
  Scala static code analysis tool for code examination and detecting potential problems. With predefined list of rules the code is analyzed and every violation is reported. There is Intellij IDEA plugin for integration ScalaStyle into IDE. This tool covers issues ranging from code style to best practices and helps to keep the codebase in sustainable state. Here a few examples of ScalaStyle rules:
  · Checks that null is not used
  · Checks that return is not used
  · Checks that while is not used
  · Checks that functions do not define mutable variables
  · Check the number of lines in a file
  · Check the average number of lines for a class
  · Check the function nesting level
  A full list of rules contains over 60 positions with support for custom rules creation.

- GitHub
  A platform for software developers, for hosting, sharing and developing user projects using Git. After last releases it also provides reach support for project management,

with such features as issues tracking, managing pull request, milestones, project boards, repository wiki page, etc. With this set of tools gather in one place, GitHub opens as a decent project management platform.
For more advantageous experience GitHub Student Developer Pack was used, allowing to use private repositories and number of additional applications for project developing for free.

- DigitalOcean
  A cloud application platform for application deployment and management. This tool will be used for hosting Droplets (virtual machines), which will form a computational cluster for application testing in distributed environment. DigitalOcean provides easy and simple one-click application deployments on its multiple Droplets available worldwide. It provides a range of different hardware solutions, including lightweight, basic and optimized machines, which may be chosen depending on resources needs.
  This platform does not provide free computation nodes, with starting price of $5/month for the smallest droplet. However, a coupon for $50 in platform credit may be accessed as part of GitHub Student Developer Pack.
- Spark Web UI
  Spark Web UI is the web interface of a Spark application to monitor and inspect Spark job executions in a web browser. Its UI visualization saves a lot of time, comparing to extracting information by reading logs.
- Databricks
  Databricks is a unified analytics platform, from original creators of Apache Spark. It allows to construct, execute and test Spark lifecycles in web-page design. It is a comprehensive tool for learning and working with Apache Spark, as it does not require any installation of heavy Spark packages on the user side.
- Resource monitoring tools Depending on the operation system one of resource monitoring tools was used. The list of tested operating systems contains Windows 10, Ubuntu Desktop 18.04, Ubuntu Server 18.04.

3.1.3. Core components

This is a list of core components of demonstrated solution. It contains also a description of what is the purpose of the component and how it is implemented. All of the functional requirements are fulfilled in terms of these components.

- Individual
  An individual is a core component of genetic algorithm, often also referred to as a *genotype, chromosome* or *a candidate*, which represents a candidate solution to a given problem. The number of individuals are evolved during the algorithm workflow, mixed and mutated in order to find the fittest one according to provided fitness function. This

concept is purely abstract in the library, encoded as a generic type which may have an arbitrary representation defined by the users depending on their needs.
In order to handle an individual through the process of evolution there should be defined a set of operators, where each encapsulates logic of some part of evolution procedure. Each operation is implemented as a **type class**. A type class is a type system construct, that supports ad hoc polymorphism. This type of polymorphism is available in Scala, using the idea of *implicit parameters*. Generally a type class is notated as type `T[A]`, where generic `T[]` is a type class and `A` is a parameter type. On the higher level of analysis, an instance of such type class may be considered as mathematical proof, for every value `a: A` to be capable of functionality declared in type class definition. The advantages of this approach over other kinds of polymorphism (both parametric and subtyping) comes with possibility of usage of multiple instances of the same type class, where the behavior of value `a` may be different, without a change to `A` implementation.

- Fitness
  Fitness is a type class which represents a function of type `I => Double`, where `I` is type of individual. This function summarizes how close a given design solution is to achieving the set aims. A fitness function must be devised for each problem to be solved, thus should be provided by the user and may not be preimplemented. Fitness function value is main criterion of future selection and is very important attribute of individual representation.
  ToDo - how much does it cost

- Join
  Join is a type class, which represents a cross function of type `(I, I) => (I, I)`, where `I` is type of individual. This function defines how two instances of type `I` may be combined together to produce a new pair of `I`. It describes a crossover operation, due to which all the individuals from a population combined into pairs will be mixed during evolution process. An instance of the type class may be created from a single function of type `(I, I) => (I, I)`, as well as a function of type `(I, I) => I`. The latter case is suitable when identical function is used to compute both of result elements or in case of commutative operation. These cases were separated in order to enable memory allocation optimization by reusing the same product object to construct result pair.

- Modification
  Modification is a type class, which represents a mutation function of type `I => I`, where `I` is type of individual. This operator is used to maintain generic diversity from one generation of a population to another. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state, during which the solution may change entirely from the former shape. Because of

its loose representation (even identity function may be considered as modification) it is important to provide a set of laws, which will distinguish a modification, which actually brings a diversity into the population. A *lawful* modification function is one, which holds following properties:

1. A modified instance does not equal to original one (*e.i modification function does not equal to identity function*) 3.1

```
modify(i) != i
```

Listing 3.1: The first laws of Modification instance

2. After a certain number of modification the same input produces different outputs (*e.i. modification function is randomized or depends on outer variables*) 3.2

```
def modify5(i: I) = modify(modify(modify(modify(modify(i)))))
modify5(i) != modify5(i)
```

Listing 3.2: The second laws of Modification instance

· Scheme
Scheme is a type class, which is responsible for a creation of new individual. This is additional function, which may be replaced by predefined initial population during the start of evolution process, but still offers more functionality and may be used when individuals are created randomly, from limitless iterator, smaller collection, which should be cycled or even single instance of individual. This concept is implemented as a covariant class, so the scheme object of parent class may be replaced by the scheme object of child class. It allows to easily use this class with hierarchies, avoiding redundant code.

Together this satisfies predefined functional requirement 6

- Population
Population is a collection of individuals, which holds them during single evolution step. Individuals are processed together and are viewed only as a part of population from the perspective of algorithm. During the evolution step the representation of population or holding type may be changed (e.g. after selection stage further processed population contains pair of individuals, which are meant to be mixed later).
From implementation perspective, population is a type alias to Scala vector collection. The choice of data structure was motivated by fast average performance and effectively constant random access time in particular.

- Evolution environment

  Evolution environment is a component, which taking the set of options constructs endless evolution flow of given population and makes it ready to be run. It is a parent type to all implementations of different approaches in processing computations, which may be sequential or parallel, local or distributed. This concept is very important from the usage perspective as it divides the implementation of genetic algorithm from its usage, so the end user may even not be aware of the way how the evolution is performed. It is also the end point of all the dependent components, as there is no direct dependency to any of concrete implementation of evolution environment. This abstraction may be considered as high level representation of genetic algorithm itself. The list of concrete derivatives of this interface contains:

  – Local environment

    This type of environment presumes local evolution of evolution flow. It is the richest environments in terms of possible configurations. The logic, behind the strategy of applying genetic operators and evaluating fitness values is decoupled from one another into instances of `EvolutionCompanion` and `FitnessEvaluator` classes respectively. For each of the strategies the are sequential and parallel versions, with additional fitness caching option for repetitive individuals within population. Any fitness evaluation strategy may combined with any possible technique of applying genetic operators, creating a number of possibilities ready for use. *This satisfies predefined functional requirement 4*

  – Asynchronous environment

    Asynchronous environment delegates the function of fitness evaluation to another system, by sending non-blocking asynchronous calls for every element of the population. It may be extremely useful for the problems with heavy fitness functions. This approach allows to compute fitness values on different machine or even cluster of machines, as well as using different low-level programming language or some specific optimization, with no need to switch the entire application to another platform. This type of environment also fits perfectly to cloud computing solutions, as fitness evaluation is entirely isolated from the rest of application. However, applying genetic operators in same way needs to send a lot more data over the network, which negates the benefits one can gain from this approach. For this reason, genetic operators are applied locally with any of the strategies described earlier.

    This satisfies predefined functional requirement 5

  – Distributed environment

    Distributed environment is very different from the previous options. Here the evolution is performed on the population in terms of Spark RDD, distributed over the cluster. Each of the executors works on its part of the data, which increases overall

performance. This technique enables processing of very large population, since it is split between many executor nodes. Once the evolution step is done executors shuffle the data between each other in order to perform selection of individuals for the next population and continue working on the local values. It minimizes the network communication, which may be a bottle-neck in such applications.

This satisfies predefined functional requirement 7

- Evolution flow
  Evolution flow represents the endless evolution process of given population. It is implemented as the reactive stream with one output, which may be plugged in into further part of pipeline or executed standalone. During running, evolution flow emits the instance of population once it is computed and starts working on the next one. Since it is a reactive stream, the process of evolution is continued during there is a demand from the downstream. For example, if the sink of pipeline needs to take only 50 first populations, no more than 50 will be computed. This technique allows to leave the variety of possible stop conditions to the end user, including number of elements pushed from the flow, total time, idle time since the last push or any predicate based on the last produced population(s).

  This satisfies predefined functional requirements 1, 8, 2

- Fitness caching
  Fitness caching is implemented using decorator design pattern over the provided fitness function. It supports every fitness function provided by user, as it simply holds the record over the ranked individuals and returns stored result if it was encountered earlier. The data structure used for holding cached values is concurrent hash trie. It guarantees consistency in multi-threaded environment, enabling safe usage with parallel workflows. From the performance side, concurrent hash tries allow to lookup element in logarithmic time, with constant snapshot time [4] (operation which is performed when two or more threads try to modify the same element).

  This satisfies predefined functional requirement 3

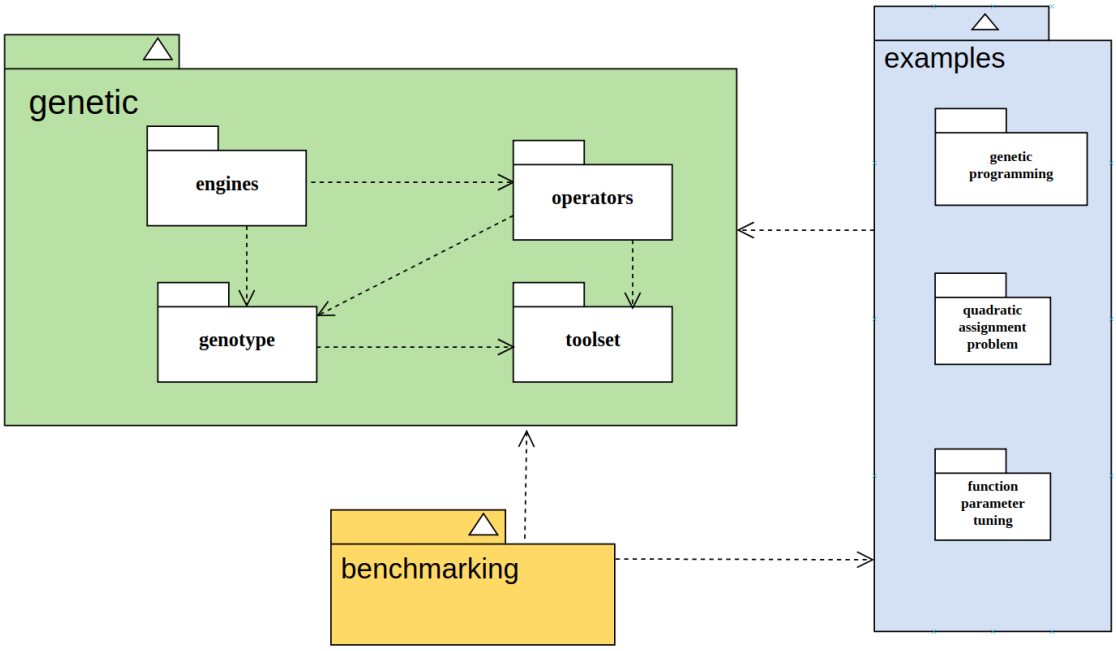
3.1.4. Project structure

The whole project is divided into three main packages:

- *genetic*
- *examples*
- *benchmarking*

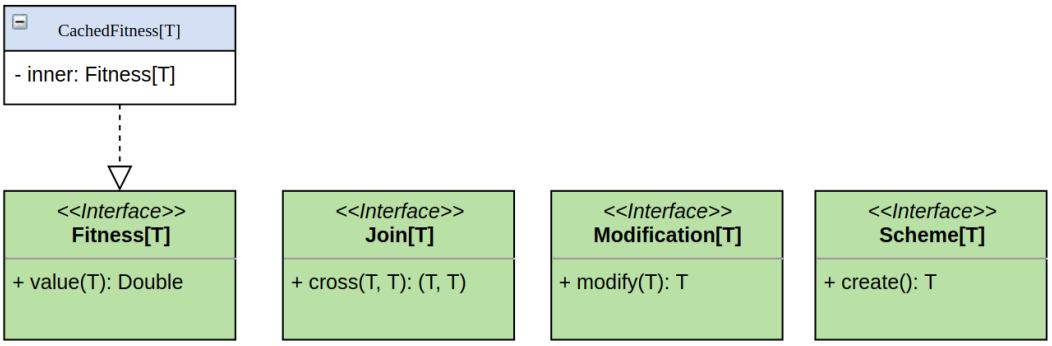
examples and *benchmarking* are additional packages, which demonstrate how the library may be used and what results one may achieve by using it, while the main value have the

classes from package *genetic*. The source code dependencies between the packages may be found on package diagram 3.3



Pic. 3.3. Packages and their source dependencies

genotype package contains classes, which correspond to the information used to describe behavior of an individual. This is achieved using 4 type classes: *Fitness, Join, Modification* and *Scheme*. This sub-package also contains *CachedFitness* class, which is a wrapper around provided fitness function, with additional caching behavior 3.4.



Pic. 3.4. Type classes used to describe an individual

.

engines package contains the heavy part of genetic algorithm implementation. Here one may find EvolutionEnvironment class and its concrete implementations. Every child class

(parallel implementation) is placed in its specific subpage, so users have to import only the part, which they are interested in or everything, if this is desired. This approach may be seen in project structure 3.5, where the interfaces, which are communicating between each other are placed at the top of the package and different variants of their implementation are placed into specific sub-package, which later is important for user needs.

This satisfies predefined non-functional requirement 3

The way in which parts of *engines* package are related to each other may be observed in *genetic* class diagram3.6

3.2. USAGE

The library artifact may be added into the project as dependency using maven tool or sbt. After this, if library requirements are met, imported classes are ready to be used.

3.2.1. Requirements

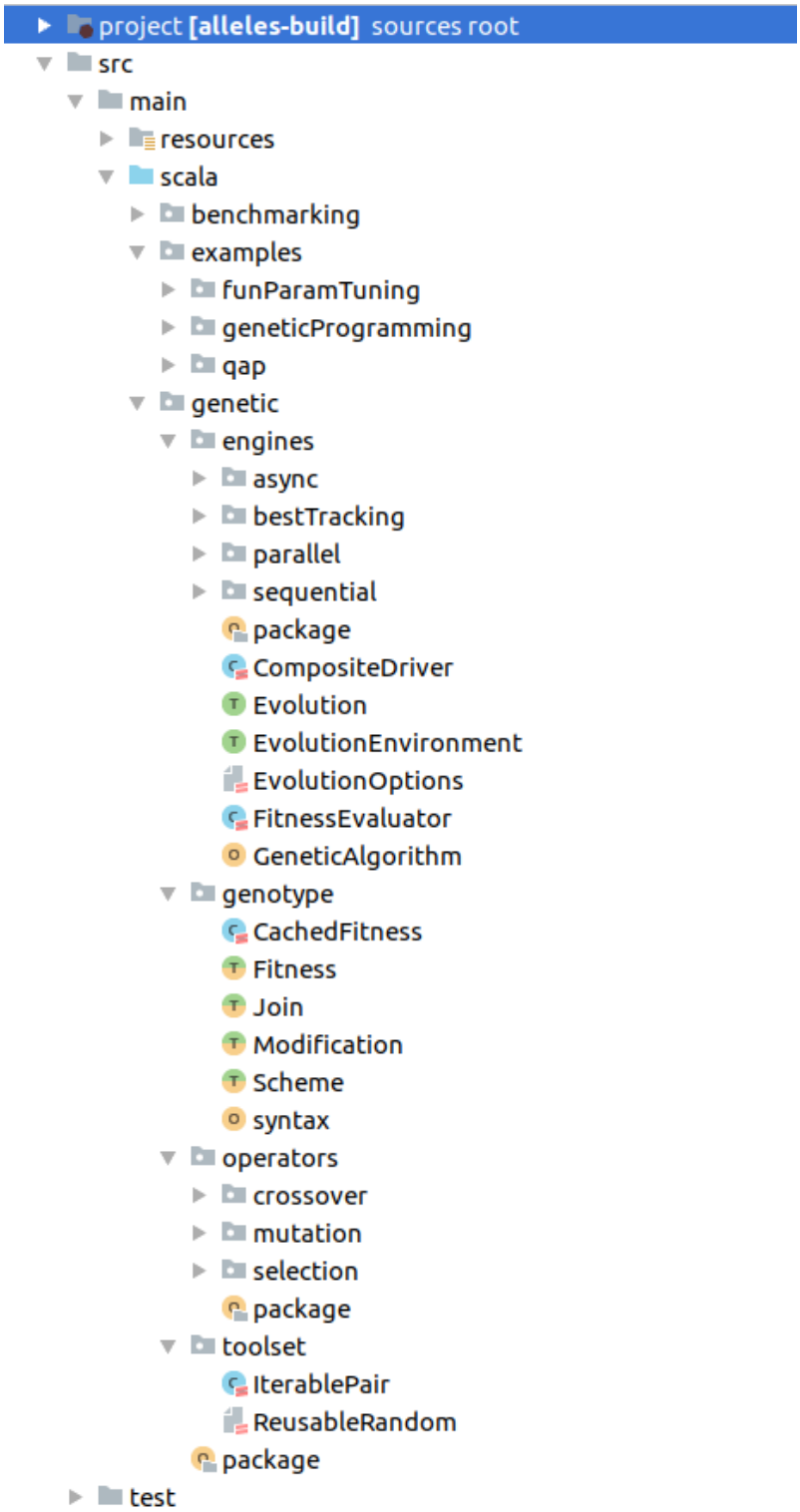
- Scala 2.12.6
- sbt 1.2.1
- Apache Spark 2.3

3.2.2. API

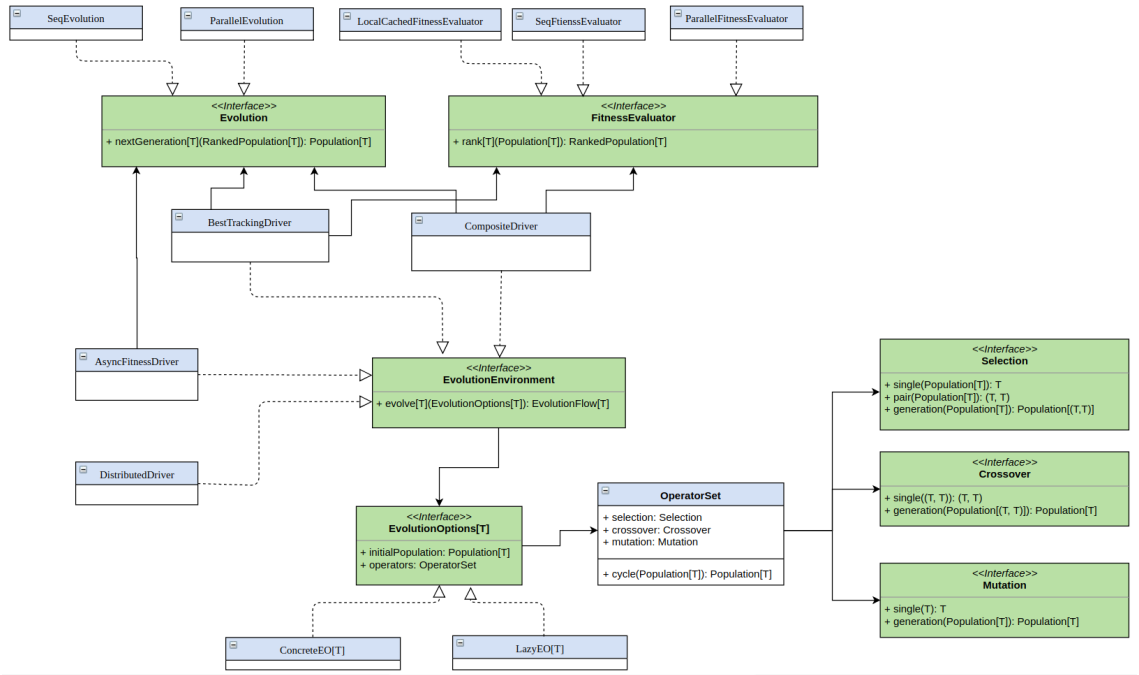
Here is described the application programming interface of the library. Due to its specific type, the frameworks do not usually have use case diagram defined, but instead present its value through API. It is brief description of the classes and functions available to a user, in which they communicate and actually use a programming library.[1]

- `trait EvolutionEnvironment`
  - `def evolve[T: Fitness : Join : Modification](options: EvolutionOptions[T]): EvolutionFlow[Population[T]]`
    Creates an evolution flow out of evolution options of type T, with available implicit objects of class Fitness[T], Join[T], Modification[T]
- `type EvolutionFlow[T] = akka.util.Source[T, NotUsed]`
  A type alias to akka-streams Source. Contains all the methods of akka.util.Source
- `trait EvolutionOptions[T]`
  A trait which holds the parameter for constructing evolution flow
  - `def initialPopulation: Population[T]`
  - `def operators: OperatorSet[T]`

[1] Description of the rest of the classes was omitted due to its large size



Pic. 3.5. Expanded project structure



Pic. 3.6. Classes diagram from *genetic* package

- `trait Evolution`
  Corresponds to the one step long evolution. Derives rated population to the new one using `class OperatorSet`
  - `def nextGeneration[T: Join : Modification](ratedPop: Population[Rated[T]], operators: OperatorSet): Population[T]`
- `class FitnessEvaluator`
  A class which basing on the instance of Functor[Population[T]] computes the fitness values for the population.
  - `def rate(population: Population[T]): Population[Rated[T]]`
- `trait Fitness[T]`
  - `def value(t: T): Double`
    Computes the fitness value for an individual of type `T`
  - `def cached: CachedFitness[G]`
    Returns a cached version of this fitness function
- `trait Join[T]`
  - `def cross(a: T, b: T): IterablePair[T]`
    Mixes a pair of individuals into new pair with shared genome
- `trait Modification[T]`
  - `def modify(t: T): T`
    Returns modified version of parameter `t`

- `trait Scheme[+T]`
  - `def create: T`
    Creates a new instance of type `T`
- `trait Selection`
  - `def single[T](population: Population[Rated[T]]): (T, T)`
    Selects a single pair from the Population of rated individuals
  - `def generation[T](population: Population[Rated[T]]): Population[(T, T)]`
    Selects pairs of individuals into new population of the same size
- `trait Crossover`
  - `def single[T: Join](parents: (T, T)): IterablePair[T]`
    Mixes one pair of individuals with provided `Join[T]` instance
  - `def generation[G: Join](population: Population[(G, G)]): Population[G]`
    Mixes every pair of given population into the new population with provided `Join[T]` instance
- `trait Mutation`
  - `def single[T: Modification](individual: T): T`
    Mutates a single individual with the `Modification[T]` instance
  - `def generation[T: Modification](population: Population[T]): Population[T]`
    Mutates all population with the `Modification[T]` instance

3.3. MEETING THE PREDEFINED REQUIREMENTS

With current implementation all predefined functional and non-functional requirements were successfully met.

4. EXPERIMENTS

Here the purpose of experiments is to observe speedups gained in a result of local parallelization.

4.1. TESTING ENVIRONMENT

4.1.1. Settings

The experiments will be performed on one of the implemented examples of library usage. This is the solution for Quadratic Assignment Problem. It is one of the fundamental combinatorial optimization problems in the branch of optimization or operations research in mathematics, from the category of the facilities location problems. For this purposes could have been used any problem, as in order to make conclusions, the results will be compared with each other.

Here is a list of the parameters mutual to every test:

- Optimization problem
  The experiments will be performed on one of the implemented examples of library usage - Quadratic Assignment Problem. It is one of the fundamental combinatorial optimization problems in the branch of optimization or operations research in mathematics, from the category of the facilities location problems. For this purposes could have been used any problem, as in order to make conclusions, the results will be compared with each other.
- Genetic operators
  For the evolution flow the set operators stays unchanged. It contains of tournament selection strategy with the size of 20, one point crossover and simple chromosome swap mutation. This parameters have been left immutable for create minimum parameter noise during experiments.

Other algorithm parameters like number of iterations, size of the population and complexity of fitness function will be regularized in order to examine the correlation.

4.1.2. Hardware

The experiments were performed using two machines with different configuration, motivated by the different results, which may be observed depending on the used machine.

Tabela 4.1. Hardware characteristics

Label	Platform	Total RAM	vCPU cores
Windows machine	Windows 10	16 GB	8
Ubuntu/Linux machine	Ubuntu 18.04	8 GB	4

4.2. PARALLELIZATION AND ITS BENEFITS

In this section will be studied the benefits, which may achieved be applying parallelization to any of used algorithms.

4.3. SEQUENTIAL BEHAVIOR

In order to be able to make some conclusions having the data from experiment runs, it is important to know how the system behaves in sequential runs. This configuration may also be considered as one with parallelism level 1.

Run observations:

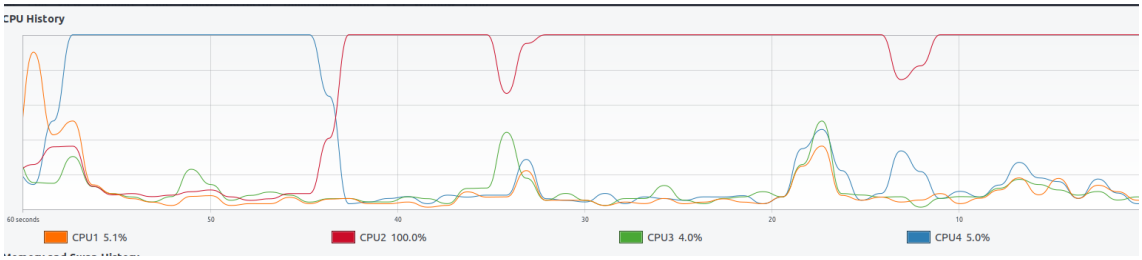
Tabela 4.2. Run observation #1

Machine	Platform	Total RAM	vCPU cores
Windows machine	Windows 10	16 GB	8
Ubuntu/Linux machine	Ubuntu 18.04	8 GB	4

Run parameters:

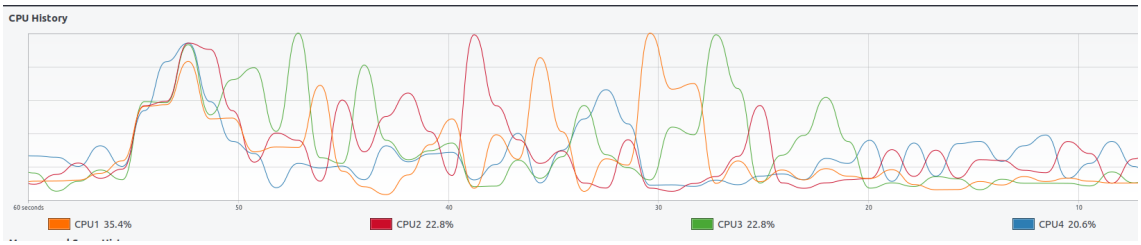
- Number of iterations: 10
- Size of population: 5000
- Fitness function duration: 5 mills

After the run with a sequential implementation of GA here is a plot of processor resources usage over time 4.1. It is clear that only one vCPU core is working on its full speed at a time, but there is no vCPU core, which runs all the evolution, as after some time it is changed to another core. In the meanwhile, there bumps on other cores every some time. This is garbage collector sessions, as by default JVM runs GC on the free cores in order to minimize actual working time.



Pic. 4.1. CPU load during test run #1

To test whether conjecture about GC runs is actually true for the next run the size of a population was increased to 50000 and fitness function time reduced to 0.5 ms. This should increase number of objects allocated during a given time segment, resulting in more frequent garbage collector sessions. Here is the observations over CPU status during the run 4.2. This plot confirms previous assumption.



Pic. 4.2. CPU load during test run #2

The important conclusion from these runs is that even during sequential computations JVM still uses other available cores to optimize working time. For this reason, when using more than one CPU core for the purpose of algorithm, JVM may find no free cores and will schedule garbage collection together with other computations, which will naturally result in a downturn of performance. **This is one of the reasons, why the speedup from parallelization is not ideally linear to the number of added cores**

4.4. MEASURING PARALLELIZATION IMPACT

In this section will be measured performance increase in different thread pool configuration. Performance of each configuration will be measured and compared to the performance of sequential run without thread pool limitation. By applying one, depending on the type of thread pool, application is restricted to use no more or exact number of threads to its purposes. This also includes garbage collection, so statement that sequential run is equal to parallel with limited pool to one thread is not true, as in the former case garbage collection may be scheduled on the other thread by JVM, where in the letter - one thread is responsible for all actions. This is why parallelization with small number of threads gives very small speedup.

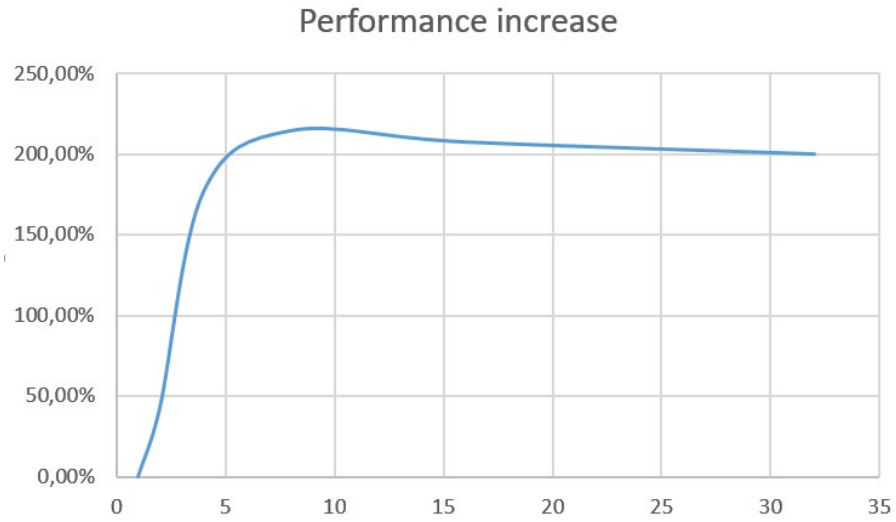
The measurements can be found in tables 4.4 and 4.5 in comparison to sequential run values placed in 4.3.

Tabela 4.3. Performance of sequential configuration of genetic algorithm

Machine	Taken time	Population size
Ubuntu machine	1779 ms	10000
Windows machine	1725 ms	10000

Tabela 4.4. Performance speedup depending on available threads comparing with sequential run on Ubuntu machine

Number of threads in the pool	**Taken time**	**Performance increase**	Available vCPU cores
2	1247 ms	42.6%	4
4	640 ms	177%	4
8	564 ms	215%	4
16	576 ms	208%	4
32	592 ms	200.5%	4

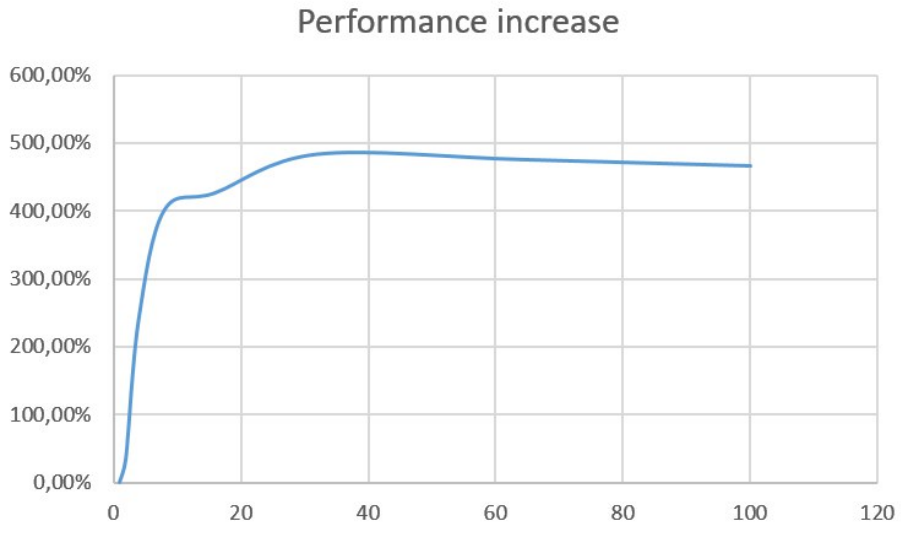


Pic. 4.3. Visualization of performance increase per available threads on Ubuntu machine

Using graph visualizations 4.3, 4.4 it is shown, that slope of performance increase is independent from machine or operating system. From observations it also appears, that **optimal point of performance gain is approximate to $2 * n$ threads available in case of fixed thread pool, where n is number of vCPU cores available on machine.**

Tabela 4.5. Performance speedup depending on available threads comparing with sequential run on Windows machine

Number of threads in the pool	**Taken time**	**Performance increase**	Available vCPU cores
2	1276 ms	35.2%	8
4	509 ms	238%	8
8	345 ms	400%	8
16	328 ms	426%	8
32	296 ms	482.7%	8
64	300 ms	475%	8
100	305 ms	465.6%	8



Pic. 4.4. Visualization of performance increase per available threads on Windows machine

BIBLIOGRAFIA

[1] Bentley, P., Corne, D., Engineering Information, I., Boden, M., *Creative Evolutionary Systems*, Evolutionary Computation Series (Elsevier Science, 2002).

[2] Karau, H., Konwinski, A., Wendell, P., Zaharia, M., *Learning Spark: Lightning-Fast Big Data Analytics*, 1st wyd. (O'Reilly Media, Inc., 2015).

[3] Odersky, M., Spoon, L., Venners, B., *Programming in Scala: Updated for Scala 2.12*, 3rd wyd. (Artima Incorporation, USA, 2016).

[4] Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M., *Concurrent tries with efficient non-blocking snapshots*, SIGPLAN Not. 2012, tom 47, 8, str. 151–160.

[5] Prokopec, A., Miller, H., *Parallel collections overview*, `https://docs.scala-lang.org/overviews/parallel-collections/overview.html`.

[6] Sivanandam, S.N., Deepa, S.N., *Introduction to Genetic Algorithms*, 1st wyd. (Springer Publishing Company, Incorporated, 2007).

[7] team, A., *Streamrefs - reactive streams over the network*, `https://doc.akka.io/docs/akka/2.5/stream/stream-refs.html`.

[8] team, A.F., *Streamrefs - reactive streams over the network*, `http://commons.apache.org/proper/commons-math/userguide/genetics.html`.

[9] Wilhelmstötter, F., *Jenetics. Library user's manual*.