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, 19 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 19 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 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.

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). This chapter is devoted to the study of Biohadoops speedup characteristics and possible influencing factors.

Two bio-inspired optimization algorithms are used as benchmarks. Both use Biohadoop and its task system to solve a test problem. The algorithms are executed on a Hadoop cluster to study the impact on their execution times by varying problem sizes and number of workers.

The rest of this chapter is structured as follows: section 1.1 provides information about the cluster used for the benchmarks. Section 1.2 describes the benchmarked test problems. Section 1.3 provides guidance for the interpretation of the benchmark results. Finally, section 1.4 presents the results, section 1.5 gives a discussion about factors that influence the benchmarks.

1.1. Cluster Hardware

All experiments were 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

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

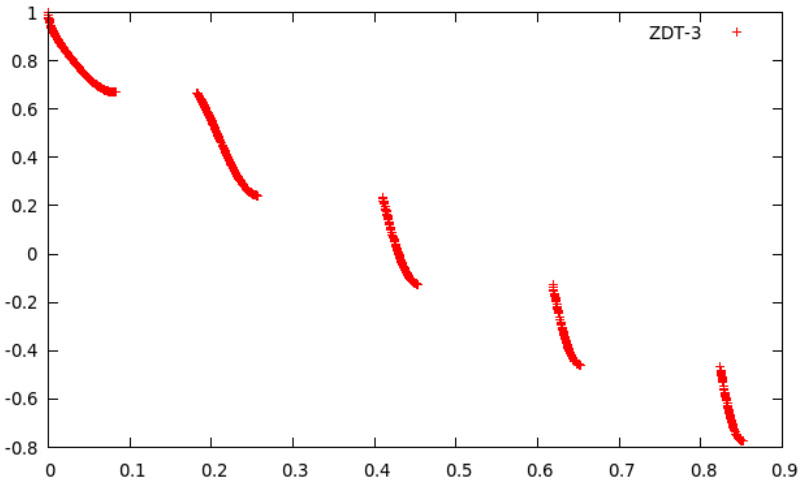

Figure 1.1.: Optimal Pareto Front for ZDT-3

1.2. Test Problems

Both implemented algorithms are part of the GA family. They differ in the number of objectives that they can handle. While NSGA-II is used to solve the MOP in section 1.2.1, a simple GA is used to solve the SOP in section 1.2.2.

1.2.1. ZDT-3

The first optimization algorithm is NSGA-II, used to find optimal solutions for the Zitzler–Deb–Thiele's function nr. 3 [1]. ZDT-3 is part of the well known ZDT family of MOP. It was chosen because of its discontinuous optimal Pareto Front (see figure 1.1) that not every optimization algorithm can handle, e.g., gradient based algorithms. The expected outcome is to find an approximation to the optimal Pareto Front.

The ZDT-3 benchmarks are executed with genome sizes of 10, 100, 1000 and 10000. The genome size corresponds to the number of values of an individual and the dimension of the solution space. Each individual is represented by its genome. ZDT-3 can handle any genome size. Changing this number influences two properties of the ZDT-3 benchmark. First, increasing the number of genomes also increases the computation effort for the workers that generate new offsprings and compute their fitness. This is due to the fact that workers generate new individuals 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.

The implementation uses Biohadoop workers to create and evaluate the offsprings. Simulated Binary Crossover (SBX) and Parameter based mutation [2] 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.

1.2.2. 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 is a parameter set 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 [3]. The computation is performed on smaller blocks (tiles) of the matrices:

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

If the blocks are small enough they fit into the L1 CPU cache which results in a speedup. For example, the average of ten consecutive test multiplications of two matrices of size 1024×1024 took 11.384 seconds for the simple matrix multiplication. It took 2.669 seconds for the tiled multiplication with tile sizes $i = 32$, $j = 32$ and $k = 32$. Both tests were performed on a single computer of the cluster. This numbers show that it is appropriate to use the tiled approach for the matrix multiplication. But the speed of TMM depends heavily on the tile sizes, the same tiled multiplication as above with tile sizes of $i = 1$, $j = 1$ and $k = 1$ took 25.683 seconds to finish, an increase of about 10 times compared to a good tile size. Because of the number of possible tile size combinations

and the time it takes to execute a matrix multiplication (e.g. matrix size=1024, 2 seconds for a matrix multiplication: $1024^3 \times 2 = 2 \times 10^9$ seconds), it is not feasible to do an exhaustive search for the optimal tile sizes.

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

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 **??**) 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.

1.3. Benchmark Details

The benchmarks are executed on the cluster described in section 1.1. The assumption is that the execution time of an algorithm depends both on the problem size and the number of workers. To evaluate this assumption, the algorithms presented in section 1.2 are executed with different problem sizes and different numbers of workers. For NSGA-II the problem size is defined as the genome size, for TMM it is the matrix size. The number of workers range for both test problems from 1 to 15.

All performed benchmarks have the following settings in common:

- The number of iterations is set to 250.
- The population size is set to 100.
- The distribution index n_c for the SBX crossover is set to 20.
- The distribution index n_m for the mutation is set to 20.

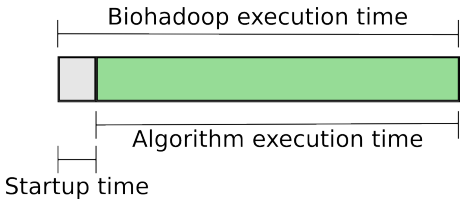

Figure 1.2.: Biohadoop execution times are composed of start up times and algorithm execution times

- The mutation probability for each offspring value is set to $1/n$, i.e., on average one offspring value is mutated.

Each benchmark is repeated five times to improve the reliability of the results, making it 300 benchmark runs for ZDT-3 (4 genome sizes $\times$ 15 worker setting $\times$ 5 repetitions) and 150 benchmark runs for TMM (2 tile sizes $\times$ 15 worker settings $\times$ 5 repetitions).

The execution time of a benchmark is composed of the time Biohadoop needs to start up and the algorithm execution time (see figure 1.2). The start up begins with Biohadoops submission to Hadoop and ends when the algorithms `run` method is invoked. The algorithm execution time starts with the invocation of the algorithms `run` method and ends when this method returns.

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

1.4. Results

To measure and compare the speedups of the benchmarks, lower and upper maximum speedup bounds are established. This is done because it is difficult to compute exact maximum theoretical speedup results based on the algorithm execution times. Those times are influenced by many factors, e.g, free resources on the computers, YARN behavior, etc. An overview and discussion of the recognized effects is given in section 1.5.

The speedup bounds are computed using the formula $S = T/(T - t_p)$ from Amdahl's law [4], where S is the speedup. The serial algorithm execution time T is measured using one worker and doesn't include the startup time, as explained

Table 1.1.: Upper and lower speedup bounds

Test Problem	Lower bound	Upper bound
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
128×128 tiled mul	63.649	134.404
256×256 tiled mul	139.685	271.925

in section 1.3. The time spent in parallel code parts t_p is measured on the master and comprises the time between task submission and result reception for all tasks. Although five benchmarks are used to compute the bounds, it is most likely that additional benchmarks would produce different bounds.

Table 1.1 shows the lower and upper speedup bounds. It is immediately recognizable that the lower and upper bounds differ largely, e.g., in the case of NSGA-II with 10 genomes the difference is more than a factor of two. This is a strong hint that algorithm execution times on Hadoop depend on more factors than just the algorithm parameters. A discussion about this factors can be found in section 1.5.

1.4.1. Benchmark Speedups

The algorithm execution times are measured and used to calculate the speedups for the benchmarks using the formula $S = T_S/T_P$. Here, S is the speedup. The sequential time T_S is the time for the execution of the algorithm with one worker. T_P is the time for the execution with 2 to 15 parallel workers.

Figure 1.3 depicts the speedups for all test problems with respect to increasing worker sizes. The ZDT-3 benchmarks show poor results. This is not surprising as the maximum theoretical speedups of this problem are small (see table **??**) and the communication overhead is bigger compared to TMM. The only unexpected outcome was that the benchmarks scale very bad with a maximum speedup of 2.498 for 1000 genomes. The reason is that the ZDT-3 benchmarks are CPU bound by the master, as the investigations in section 1.5.2 suggest.

TMM demonstrate better results, the maximum speedup was 10.507 for a matrix size of 128×128. In this case, the speedup grows near linear or even slightly better than linear with the number of workers. That a speedup is better than linear is usually suspicious but can be explained by the fact that each benchmark was repeated five times and the average times of this five executions were taken to compute the speedups. Five executions seem to be too small

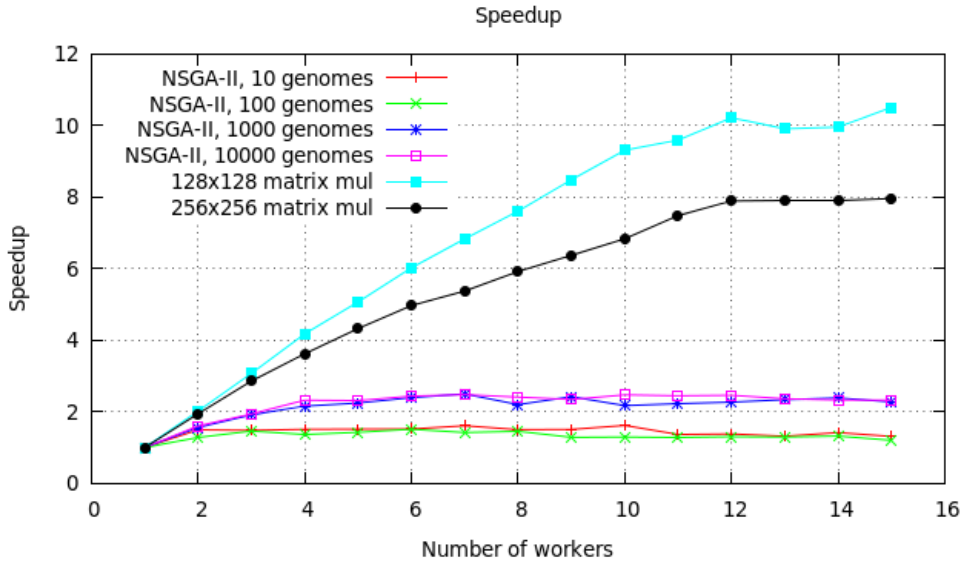

Figure 1.3.: Speedups for ZDT-3 and TMMs

for getting smooth results, especially when taking into account that the YARN container placement has big influences on the execution times.

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

The speedup for the 256×256 TMM benchmark is worse compared to the 128×128 TMM, although it also grows nearly linear until 12 workers.

1.4.2. Comparison with Implementation without Hadoop

1.5. Discussion

1.5.1. Influence of YARN Container Placement

The first thing to note when looking at the figures 1.4 to 1.9 is that the five benchmark times for a given setting (e.g. NSGA-II, 10 genomes) and one worker are very different. The explanation for this effect can be found in the YARN container placement. If a worker container is executed on the same machine as the master container, they communicate without using the physical network. This effect brings a huge performance gain, as can be seen for example in figure 1.5. 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 the execution times. This is especially true if the master is already at the limit of the machines resources and must share them with the workers. An example for this can be found in figure 1.4 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 %.

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

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

1.5.2. ZDT-3

The next thing to notice are the speedups for the ZDT-3 benchmarks. ZDT-3 is not well suited for parallelization as can be seen from the speedups in table 1.1, but the results are even worse than expected, with maximum speedups of 1.619 for 10 genomes, 1.513 for 100 genomes, 2.498 for 1000 and 2.479 for 10000 genomes. Figure 1.3 shows the speedup results of the ZDT-3 benchmarks together with the speedups for TMM.

The ZDT-3 benchmarks seem to suffer from the lack of one or more resources (bound by the resources), which prohibits further speedup increases. The investigations show that the ZDT-3 benchmarks are not bound by memory, i.e., memory issues don't slow the execution down. All benchmarks start with 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 (delivered with Java) for the ZDT-3 benchmark with 10000 genomes.

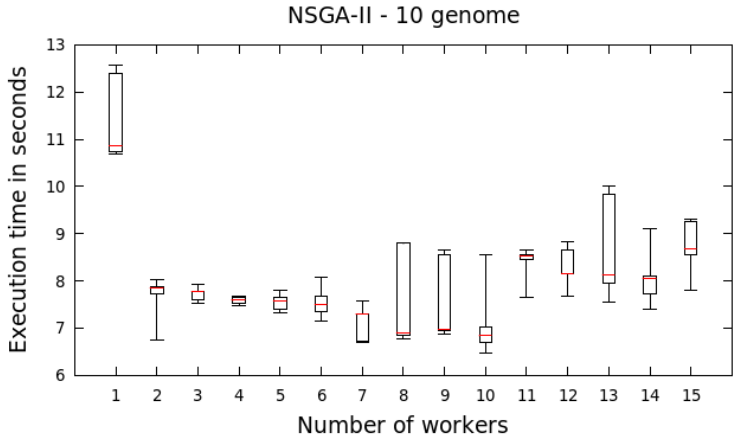

Figure 1.4.: ZDT-3 execution times for a genome size of 10

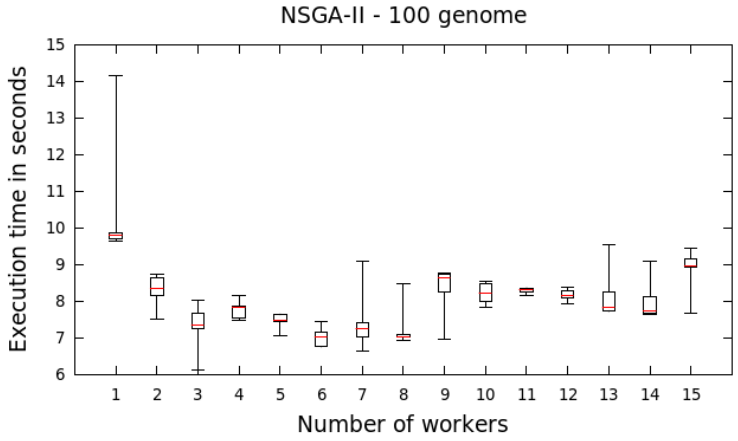

Figure 1.5.: ZDT-3 execution times for a genome size of 100

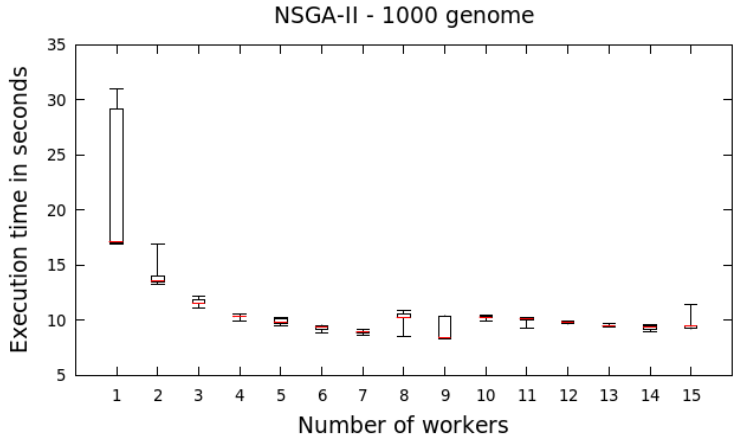

Figure 1.6.: ZDT-3 execution times for a genome size of 1000

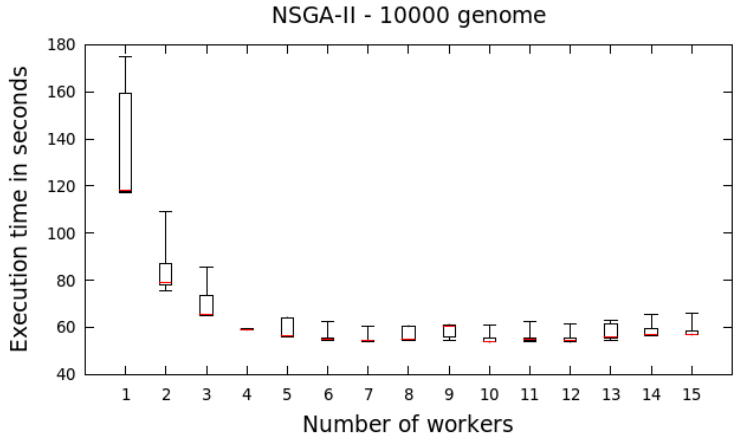

Figure 1.7.: ZDT-3 execution times for a genome size of 10000

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

Table 1.2.: Amount of network data sent from master to workers, theoretical transfer time and fastest algorithm execution

The memory usage for the master container is at about 100 MB to 150 MB. The activity of Java's garbage collector, a good indicator for memory problems, ranges from 3 % to 5.5 % of the CPU time, with an average of 3.8 %. The memory usage for a worker is even lower and lies in the range of 5 MB to 30 MB. This numbers show no significant memory problems.

The next step is to investigate the network performance. Calculations give a first hint to understand if the bad speedups can be explained with the saturation of the 1 Gb network (a small "b" denotes bits, a big "B" denotes bytes, e.g., 1 Gb = 1 gigabit, 1 GB = 1 gigabyte). In each benchmark, 250 iterations on 100 individuals are performed, resulting in 25000 tasks. The tasks are send from the master to the workers and the workers return the results. The task data send from the master to the worker contains two individuals. Each individual consists of its genome, where each value in the genome is of type `double` (8 bytes). For a genome size of 10, this makes 2 (parents) × 10 (genomes) × 8 (bytes) × 25000 (tasks) = 4000000 bytes (4 MB) or 32 Mb of data that needs to be transferred from the master to the workers during the benchmark. The size of the results sent from the workers to the master is about the half, as it consists of an individual (the offspring) and its fitness (the fitness is composed of two `double` values). This fact allows to use the outgoing data amount as upper bound for the network usage: if the outgoing data rate doesn't exceed the network bandwidth. This will be true also for the incoming data. Table 1.2 shows the results for all genome sizes together with the best algorithm execution times. One can see from the table that the benchmark data can be transferred on the 1 Gb Ethernet network during the according fastest algorithm execution time.

Additional experiments were performed to improve the confidence in the calculations and to establish the true achievable data rate for the network, given different message sizes. The experiments measure the peak network bandwidth using a small Java program and iftop.[1] The Java program uses the same com-

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

munication techniques as Biohadoop (Netty + Kryo) and performs repeated request/response cycles between a master and several workers. The exchanged messages consist of 20, 200, 2000 or 20000 `double` values, corresponding to two parent individuals in the according ZDT-3 benchmarks. The resulting peak bandwidth was 134 Mb/s for 20, 489 Mb/s for 200, 901 Mb/s for 2000 and 552 Mb/s for 20000 `double` values. The CPU on the master was the limiting factor for 20, 200 and 20000 `double` values. For 2000 `double` values, the network was saturated at 901 Mb/s and, therefore, the limiting factor. No studies were performed to explain why the experiments delivered the best results with 2000 values as this lies out of the scope of this thesis.

One phenomena regarding the network bandwidth needs further investigation. The above measurements show a peak data rate of 552 Mb/s for the case of 20000 `double` values. If this data rate is taken as a basis for the ZDT-3 benchmark with 10000 genomes, one can calculate that more than 55 s are needed to exchange 32 Gb of data over the network between the master and its workers (32000 Mb / 552 Mb/s = 57.97 s). The explanation can be found once more in the YARN container placement. The 552 Mb/s peak bandwidth is the data rate that is send through the Ethernet port to the cluster, but worker containers that run on the same machine as the master don't use this port for communication. Instead, they communicate through the local interface. iftop showed an additional combined data transfer rate of 400 Mb/s (send and receive data rates are added) on the local interface when workers were running on the same machine as the master. This gives an aggregated peak data rate of 700 Mb/s to 800 Mb/s for the outgoing traffic, which is fast enough to transmit 32 Gb of data in less than 55 s. The execution times were higher in cases where no workers executed on the same machine as the master.

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

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

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

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

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

1.5.3. Tiled Matrix Multiplication

The optimization goal of this benchmark was to find optimal tile sizes such that a matrix multiplication performs as fast as possible. The theoretical speedups for TMM promise better results (see table **??**) as matrix multiplications are compute intense and clearly dominate the algorithm execution time. Figure 1.8 and 1.9 show the execution times. One can see that the execution times decrease with the number of workers. This scales until 12 workers, after which the execution times remain constant or even increase slightly. The reason for this is that the cluster offers 12 CPU cores in total. When all cores are fully utilized, which happens with 12 workers, additional workers have to share CPU resources. This negatively impacts the execution times. So, TMM is CPU bound by the workers.

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

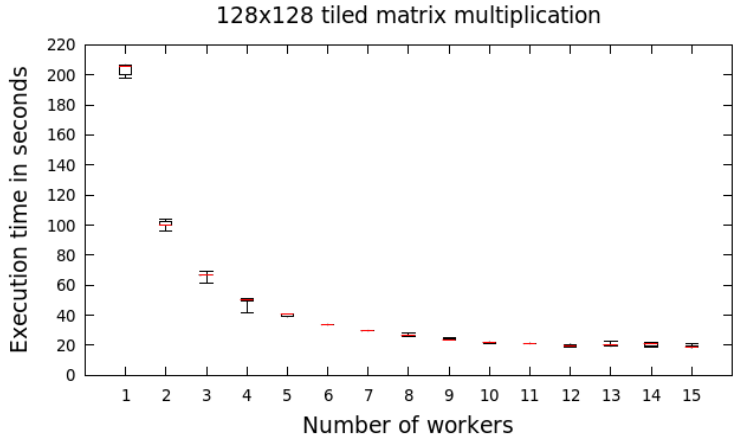

Figure 1.8.: TMM execution times for a matrix size of 128×128

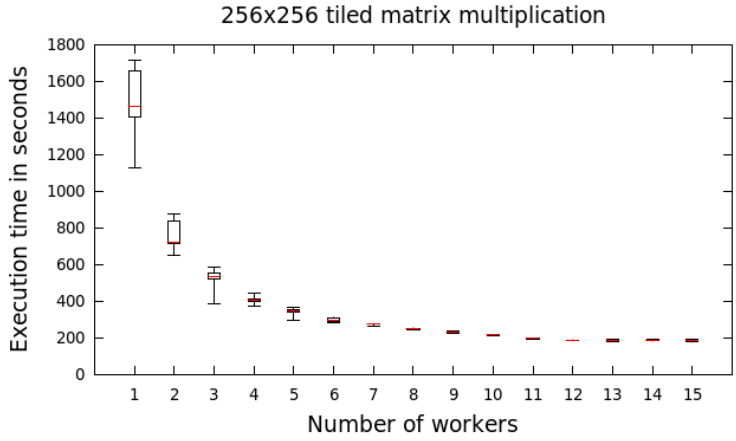

Figure 1.9.: TMM execution times for a matrix size of 256×256

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

Chapter 2.

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 Biohadoop has the potential to scale efficiently if the parallel part of the problem dominates the whole execution time. An example for such a problem is TMM, that scaled up to a factor of 10 for 12 cores. ZDT-3 as the other benchmark exposed a behavior that lead to poor scale factors of about 2 for 12 cores. 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.

Biohadoop demonstrated during the ZDT-3 benchmark that it is capable of saturating a 1 Gb Ethernet network. The achieved data rates peaked at about 900 Mb/s which is close to the theoretical limit of the underlying network. However, the achieved data rate depends on the size of exchanged message.

The evaluation results show that Biohadoop is able to both scale up with the number of available computers and to saturate the underlying network. This properties make it suitable for the use by bio-inspired optimization techniques. Further research would be needed to compare the performance of Biohadoop to other Hadoop frameworks like Spark or Storm.

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 **??** 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:

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

As one can see, now the command `yarn` is used to launch Biohadoop. This command takes care of setting all the needed Hadoop environment variables, after which it starts the provided main class. The `yarn` command is part of Hadoop since version 2, and should be available if Hadoop is configured 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.

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

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

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

A.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:

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

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

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

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

[2] 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.

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

[4] 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.