Bio-inspired optimization techniques using Apache Hadoop and Oozie

master thesis in computer science

by

Christian Gapp

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

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

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

Innsbruck, 18 November 2014

Certificate of authorship/originality

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

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

Christian Gapp, Innsbruck on the 18 November 2014

Abstract

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

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

Contents

Chapter 1.

Implementation

1.1. System architecture

Biohadoop is a framework that provides the capabilities to perform asynchronous parallel computations on Apache Hadoop. It leverages the algorithm implementors from the burden of writing a Hadoop Yarn program (see chapter **??**) and provides services that can be used to offload compute intensive work to local threads and remote machines using a simple interface. In addition, Biohadoop provides facilities to load or save snapshots of the current state to disk and to build a high level parallelization between different instances of the same algorithm, called island model (see chapter!!!ISLAND MODEL!!!!). Figure 1.1 gives a high level overview of the system architecture, that shows how the Biohadoop architecture maps to Yarn:

1. The Yarn client. In Biohadoop it is implemented through the class `at.ac.uibk.dps.biohadoop.hadoop.BiohadoopClient`. It is the main entrance point to run Biohadoop in a Hadoop environment and responsible to start the application master under the control of Hadoop.

2. The Yarn application master. In Biohadoop it is implemented through the class `at.ac.uibk.dps.biohadoop.hadoop.BiohadoopApplicationMaster`. It is the first class that is started in a managed Hadoop container. In a local (development) environment, it is also the main entrance point to run Biohadoop (more on how to run Biohadoop can be found in **??**). Beside of starting the algorithm, it is responsible to launch the task system.

3. The Yarn containers are launched by the application master. Each container runs exactly one worker, that can be used for computation.

In addition, Figure 1.1 shows the architecture for offloading data from the main algorithm to local or remote workers. This is done by submitting chunks of data and a description of how to compute the results for this data chunks from

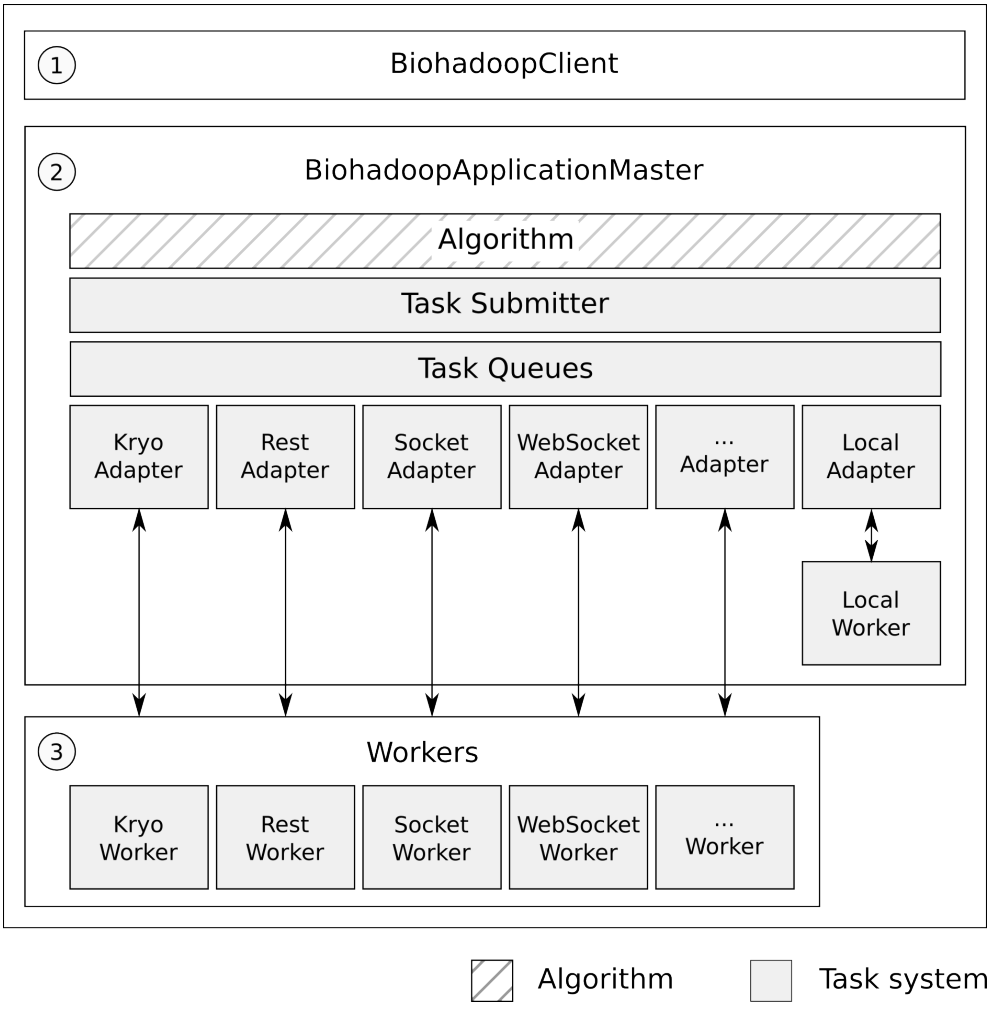

Figure 1.1.: Biohadoop system architecture

the algorithm to the task system. In this architecture, the algorithm (section 1.2) defines the problem that should be solved, e.g. an optimization problem. The data chunks, from here on called tasks, are smaller parts of this problem (e.g. compute the square root for a number), that can be submitted in an asynchronous fashion to the task system.

The task system consists of several parallel pipelines, whereas each pipeline is composed of task submitters, a task queue, adapters and workers. The tasks are actively requested by the workers, that communicate to their corresponding adapters. The adapters get the tasks out of the queues and deliver them to the workers. After the results are computed by the workers, those results are propagated back to the algorithm and the workers are free to request new tasks.

The whole task system runs in an asynchronous fashion, submitted tasks are not blocking the algorithm. Section 1.3 describes the task system in detail.

The task system relies on different communication mechanisms, to distribute the work among the waiting workers. Those communication mechanisms can be of any kind, whereas Biohadoop supports some types out of the box (local, HTTP, Kryo, Socket, Web Socket, as seen in figure 1.1). The reason behind the decision to enable different kinds of communication mechanisms was, that all of them have different properties, and that for some workloads one may fit better then another. It is for example possible to distribute work to so called local workers. They are just threads and run inside the same JVM (Java Virtual Machine) as the algorithms, thus providing very little overhead to a solution that is implemented straight into the algorithm. By exposing a simple interface to the task system, changing later the type of worker (and with it the communication mechanism) to e.g. a remote type is just a matter of configuration. Another example would be when, during development, the socket communication is chosen. This is a performant way of communication, but may lead to firewall problems on some systems, as the right ports must be open. Changing this to use e.g. Web Sockets later on is, again, just a matter of configuration. More information about the communication mechanisms can be found in section 1.4.

When an algorithm runs for a long time, it can be very annoying to lose results just because there was some kind of failure at any stage of computation. One solution to mitigate such problems is to save the current state of work to disk. Biohadoop supports its users in the task of saving and loading data with a set of classes and methods. One usage example is to reload the last state of an algorithm at startup, to continue the work at this point. To get more information about the persistence, refer to section 1.5.1.

Biohadoop is capable of running several algorithms at the same time. They all run in the same JVM as Biohadoop does, and may be of the same type. As Hadoop on its own doesn't guarantee when a program runs, the mentioned capability of launching several algorithms at the same time in the same JVM is a useful enhancement when it comes to high level parallelization using the island model. If the algorithms were started by Hadoop one by one, it could be possible, that they were run in sequential order, instead of running at the same time.

But Biohadoop doesn't restrict the usage of the island model to algorithms that run in the same JVM. By taking advantage of ZooKeeper [1], running

numbers	sum
$[0]$	0
$[0, 1]$	1
$[0, 2, 3]$	5

Table 1.1.: Example values for `Sum` algorithm

algorithms may find each other also across the boundaries of different Biohadoop instances (Biohadoop can be started several times). This can lead to higher scalability, but entails the aforementioned problem, that Hadoop doesn't guarantee when a program runs, so a trade off has to be made. More information about how to use the island model in Biohadoop can be found in section 1.5.2

To improve the understanding of the following sections, a simple example algorithm is introduced, called `Sum`. The algorithm takes an array of integers as input and provides the sum of them as result. For the sake of simplicity, we suppose that the input value of the `Sum` algorithm is never null. A simple implementation in Java can be found in listing 1.1, the output for different input values is given in table 1.1.

Listing 1.1: Algorithm for integer summation

```
1 public int sum(int[] numbers) {
2   int sum = 0;
3   for (int number : numbers) {
4     sum += number;
5   }
6   return sum;
7 }
```

This example algorithm is good enough to demonstrate the usage of Biohadoop, other (more sophisticated) examples can be found at [2]. Chapter **??** provides a step by step tutorial on how to implement this algorithm on Biohadoop.

The following sections in this chapter continue to describe the implementation of Biohadoop in more detail, starting with the notion of Algorithm in section 1.2. The next section 1.3 talks about the task system, followed by the description of the communication mechanisms in section 1.4. The enhancements section 1.5 talks about how Biohadoop supports persistence and the island model. How to configure Biohadoop is explained in section 1.6. If one just wants to use Biohadoop, chapter **??** may be more helpful.

1.2. Algorithm

An algorithm in terms of Biohadoop is the implementation of an abstract problem that should be solved. For example, a genetic algorithm may be implemented to solve an optimization problem. Biohadoop supports the algorithm implementor with an easy way to parallelize the algorithm by providing an asynchronous task system (see 1.3). In addition, mechanisms for persistence and high level parallelization using an island model are offered (see 1.5). All those capabilities can be used by the algorithm, but Biohadoop does not force their usage. The only thing an algorithm has to do to be run by Biohadoop, is to implement the `at.ac.uibk.dps.biohadoop.algorithm.Algorithm` interface, shown in listing 1.2. This interface defines one method, which is invoked by Biohadoop after the system initialization has completed. There may run several algorithms of any kind at the same time in one Biohadoop instance, this is just a matter of configuration. After all algorithms have terminated, Biohadoop stops all components and shuts down.

Listing 1.2: Algorithm interface

```
1 public interface Algorithm {
2   public void compute(SolverId solverId, Map<String, String>
        properties) throws AlgorithmException;
3 }
```

- `void compute(SolverId solverId, Map<String, String> configuration)`: This method is invoked by Biohadoop after the system initialization has completed. It is the place, where the algorithm should be implemented. The parameter `solverId` defines the global id of this instance of the algorithm, as there may run several instances of the same or different algorithms at the same time. The `solverId` can be used by other components to uniquely identify the algorithm. For example the persistence extension, which saves an algorithms state to a file, uses the `solverId` as part of the filename.

  The `properties` map contains the configuration for this algorithm (see 1.6 for more information about how to configure Biohadoop). It is a map with key - value pairs of strings.

  The return value of the `compute` method is void, as there was no general return data identified, that could be useful. This may change in future versions.

  If there is an error during the algorithm execution, it may throw a `at.ac.uibk.dps.biohadoop.algorithm.AlgorithmException` at any

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

When using the example algorithm from listing 1.1, without any further usage of the task system, persistence or high level parallelization, it would look like in listing 1.3. Note that there is no difference to the original implementation, except that now we have implemented the `compute(SolverId, Map<String, String>)` method.

Listing 1.3: `Sum` algorithm for integer summation on Biohadoop

```
1  public class Example implements Algorithm {
2    public void compute(SolverId solverId, Map<String, String>
         properties) throws AlgorithmException {
3      int[] numbers = new int[]{0,2,3};
4      int sum = sum(numbers);
5    }
6
7    private int sum(int[] numbers) {
8      int sum = 0;
9      for (int number : numbers) {
10       sum += number;
11     }
12     return sum;
13   }
14 }
```

1.3. Task system

If an algorithm implementor decides to parallelize some parts of the algorithm, it can use Biohadoops task system. The task system works in an asynchronous

manner and doesn't block its users while processing the tasks. However, it provides also mechanisms to block and wait for a result to be computed, if this is preferred.

1.3.1. Task pipeline

The task system is divided into several task pipelines, that run in parallel. Each pipeline has one or several task submitters, exactly one task queue, one or several adapters and one or several workers. Figure 1.2 depicts the architecture of the task system with one task pipeline, consisting of one submitter, one queue, one adapter and one worker.

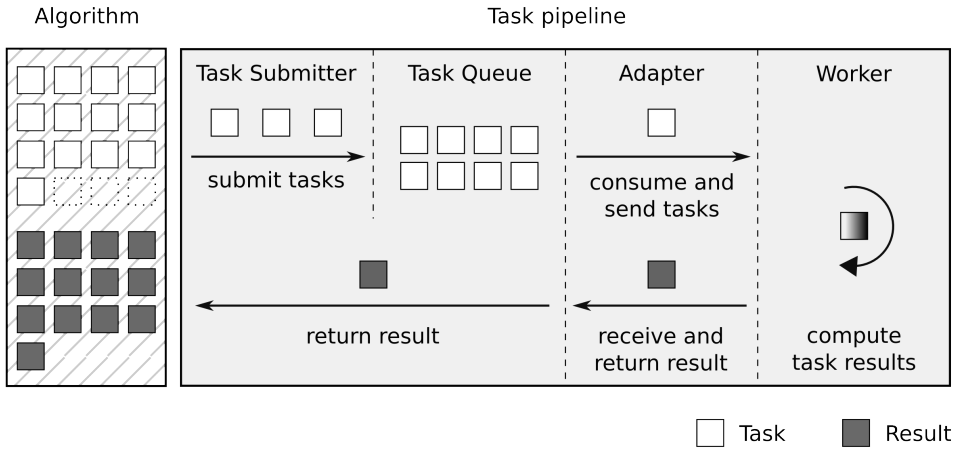

Figure 1.2.: Task system with one task pipeline. The pipeline in this figure consists of one task submitter, one task queue, one adapter and one worker

Central to each pipeline is the task queue. The queue is filled with tasks by the task submitters, that in turn get the tasks from the algorithm. Task submitters are a convenience abstraction that hide the complexity of task submission from the algorithm. An algorithm may choose to directly use a queue, although it is not recommended. Several submitters for one pipeline are supported.

On the other side of the queue, we have the adapters and workers. The duty of the adapters is, to take tasks out of the queue and to pass them to the waiting workers, which do the (possibly heavy) computation of the tasks, and deliver the results back to the adapters. The adapters then propagate those results back to the algorithm. There may run several adapters and workers in each pipeline to enable parallel computation of the tasks. Each adapter can handle several workers.

From the several parallel task pipelines, an algorithm can deliberately choose which one to use, it may use more then one. Figure 1.3 for example depicts an algorithm, that uses four different task pipelines to handle its tasks, each pipeline consists of the before mentioned components. To enable the usage of several task pipelines in parallel and to distinguish them, unique names are used.

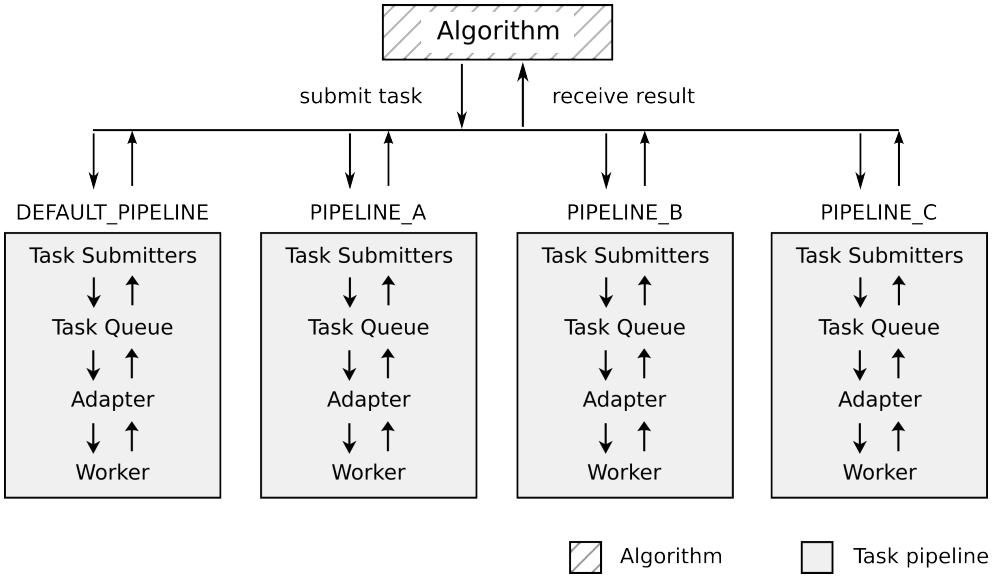

Figure 1.3.: Several different named task pipelines may exist at the same time, consisting of submitters, queues, adapters and workers

Every pipeline needs to be configured. To simplify this step, Biohadoop provides a pre-configured default pipeline, the `DEFAULT_PIPELINE`. Using this default pipeline is advised, although it is possible to choose different pipelines for the task computation. Those other pipelines are from now on summarized under the term "dedicated pipelines". For example, the pipelines `PIPELINE_A`, `PIPELINE_B` and `PIPELINE_C` in figure 1.3 are dedicated pipelines, whereas the pipeline `DEFAULT_PIPELINE` is the pre-configured default pipeline. All tasks submitted to the same pipeline share the same queue, adapters and workers.

The main advantage of several parallel pipelines is better scalability. If one pipeline gets congested, a second pipeline, which works completely in parallel to the first one, can help to diminish this problem - given there are enough computational resources to effectively run the second pipeline. Another usage scenario for additional pipelines is, when there are tasks with different priorities, and one pipeline should only contain high priority tasks. By filling this pipeline only with high priority tasks, the tasks don't have to wait for other low priority tasks.

The disadvantage of several pipelines is, that the workers of each pipeline receive only tasks from their pipeline. If no tasks are available (e.g. because there is not much work to do), then the workers have to wait, which can result in poor resource usage. Work stealing [3] between pipelines could reduce this problem, but is not implemented at the moment.

Another disadvantage of dedicated pipelines is the slightly higher configuration effort, which really boils down to configure additional adapters and workers (more about this topic can be found in chapter 1.6, where the Biohadoop configuration is explained).

In most cases, there is no need to run several pipelines and it is advised to start with the default pipeline. If there are good reasons, adding additional pipelines and dividing the tasks between them is straightforward.

1.3.2. Task Submitter

The task submitter serves as the entry point to the task system and hides the complexity of the underlying components from the algorithm implementor. The generic interface `at.ac.uibk.dps.biohadoop.tasksystem.submitter.TaskSubmitter<T,S>`, shown in listing 1.4, defines the methods that a task submitter has to implement. T denotes the type of the task data, S denotes the type of the return value.

Listing 1.4: TaskSubmitter interface

```
1 public interface TaskSubmitter<T, S> {
2   public TaskFuture<S> add(T data) throws TaskException;
3   public List<TaskFuture<S>> addAll(List<T> datas) throws
        TaskException;
4   public List<TaskFuture<S>> addAll(T[] datas) throws
        TaskException;
5 }
```

- `TaskFuture<S> TaskSubmitter.add(T)`: This method adds a single task to a queue and returns a `TaskFuture`.
- `List<TaskFuture<S>> TaskSubmitter.addAll(List<T>)`: This method adds a list of tasks to a queue and returns a list of `TaskFuture` objects, each one representing the result of exactly one submitted task. The tasks are then send one by one to the waiting workers.
- `List<TaskFuture<S>> TaskSubmitter.addAll(T[])`: This method adds an array of tasks to a queue and returns a list of `TaskFuture` objects, each

one representing the result of exactly one submitted task. The tasks are then send one by one to the waiting workers.

The methods of the `TaskSubmitter` interface return objects that implement `at.ac.uibk.dps.biohadoop.tasksystem.queue.TaskFuture<T>`, shown in listing 1.5, where T is the type of the return value. A `TaskFuture` object represents the result of an asynchronous computation performed by a worker, much like the `java.util.concurrent.Future` object, that is part of the Java standard.

Listing 1.5: TaskFuture interface

```
1 public interface TaskFuture<T> {
2   public T get() throws TaskException;
3   public boolean isDone();
4 }
```

- `T get()`: Is used to wait until the result of a remote computation is available, which leads to a blocking behavior. After the result is available, `get()` returns this result.
- `boolean isDone()`: Is used to check the task status, by which a non-blocking behavior can be realized. When `isDone()` returns true, the result can be obtained by invoking `get()` on the `TaskFuture` object.

The following code snippet illustrates a simple example, where a task is submitted to a queue, after which the program blocks until the result arrives by invoking `get()`. We assume here, that `taskSubmitter` is an initialized object that implements the `TaskSubmitter` interface. Further we assume, that the task data type is `String` and the return type is `int`:

```
1     int result = taskSubmitter.add("Hello World!").get();
```

The default implementation of the `TaskSubmitter` interface, that is provided by Biohadoop, is `at.ac.uibk.dps.biohadoop.tasksystem.queue.SimpleTaskSubmitter`. It provides three different constructors that can be used to configure the details of task submission e.g. to which task pipeline a task is submitted. See listing 1.6 for more details about the constructors.

Listing 1.6: SimpleTaskSubmitter

```
1 public SimpleTaskSubmitter(Class<? extends AsyncComputable<R
    , T, S>> asyncComputableClass);
2
```

```
3 public SimpleTaskSubmitter(Class<? extends AsyncComputable<R
      , T, S>> asyncComputableClass, R initialData);
4
5 public SimpleTaskSubmitter(Class<? extends AsyncComputable<R
      , T, S>> asyncComputableClass, String pipelineName, R
      initialData);
```

As one can see, all constructors expect as parameter at least a class that extends the `at.ac.uibk.dps.biohadoop.tasksystem.AsyncComputable` interface. This class implements the part of the algorithm, that should be run on the workers and that is used to compute the results for the submitted tasks (more information about this interface and its usage can be found at section 1.3.6).

The second constructor expects an object containing the initial data, `initialData`, that should be available in the `AsyncComputable` class and therefore on the workers. This is suitable when several tasks need the same data for their computation, that doesn't change over time. The `initialData` is send to the workers when they first encounter a task that should be computed with a given `AsyncComputable` class. After that, this `initialData` is cached at the workers.

For an example usage of the `initialData`, lets suppose we write an algorithm, that computes, for a list of locations, the distances to the nearest city and returns the nearest city. As input we have a list of cities with their coordinates (called `CITY_LIST`), and a list of locations with their coordinates (called `LOCATION_LIST`). This problem can be parallelized by breaking the list of locations down to single locations, for which the results can be computed in parallel. To get the nearest city for a given location, the algorithm computes the distances to all cities, and returns the city with the smallest distance. This algorithm could be implemented and parallelized with Biohadoop and its task system, where the task of computing the smallest distance for a given location is done by workers. Now, the `CITY_LIST` remains the same for all locations, so it would be a bad idea to send the `CITY_LIST` data to the workers for every task. This is the reason why the `initialData` exists: we set `CITY_LIST` as initial data for the task submission and it gets transferred to a worker the first time it encounters this type of task (namely to compute the distance). After this first time, the `CITY_LIST` can be reused by the worker, without the need to transmit it over and over. This usually results in much less overhead when the next task of this type has to be computed by the worker.

The last constructor expects an additional `pipelineName` argument, that defines to which task pipeline the tasks, submitted using this task submitter, should be send. Using this constructor, an algorithm implementor is able to

choose a dedicated pipeline for the task computation. Constructors that take no `pipelineName` argument, submit their tasks to the default pipeline.

1.3.3. Task Queue

Tasks, submitted by the algorithms to be processed by the task system, are stored in task queues. The queue, where a task is stored, is part of the same pipeline as the task submitter, that put the task into the queue. There may exist several task queues, each queue is assigned to one task pipeline, which uses the queue exclusively during their whole lifetime. All tasks submitted to a given task queue have the same priority. Stored tasks are consumed by adapters and, from there on, handed to the workers for computation.

Each task queue works as a first-in first-out (FIFO) queue. The head of the queue is that element, that has been on the queue the longest time. The tail of the queue is that element, that has been on the queue the shortest time. New elements are inserted at the tail of the queue. When reading from the queue, the head element is taken out of the queue and returned as result.

Every task queue is backed by `java.util.concurrent.LinkedBlockingQueue`, which supports at maximum $2^{31} - 1$ number of elements, in this case tasks. The `LinkedBlockingQueue` supports concurrent addition and retrieval of elements. This facilitates the usage of many task submitters and adapters per queue. The `LinkedBlockingQueue` may block when adding an element and the queue is already full, or when an element should be taken out of an empty queue. In this two corner cases, also the task queue blocks. This may lead to the following problems:

- The task queue blocks when an algorithm wants to submit new tasks to a full queue. This kind of problem may arise if there is a huge number (more than $2^{31} - 1$) of tasks that are added to a task pipeline way faster than they can by computed. As a result, the algorithm gets also blocked and has to wait for free space in the task queue. This is a seldom case, but if it happens, there is the possibility to distribute the work between several task pipelines (and therefor between several queues), or to do the task submission in an own thread (could be implemented e.g. in a `TaskSubmitter`).

- The task queue blocks, when an adapter wants to take a task out to send it to a waiting worker, but the queue is empty. This case happens often and results in blocking behavior of the adapters and workers. Adapters should therefor be prepared for this case and nevertheless at least accept new worker requests (Biohadoops provided adapters work in this manner).

A way to diminish the blocking behavior would be to implement some kind of work stealing between queues, that is, an empty queue may fetch tasks from other queues that contain elements. This is not implemented at the moment. As a result, workers block if there is no work to do for their pipeline.

As mentioned in the chapter before, it is preferable to submit tasks to a task queue by using the provided `at.ac.uibk.dps.biohadoop.tasksystem.queue.SimpleTaskSubmitter`, although it is not necessary. It is still possible to submit tasks directly to a task queue. To do this, first a reference to a task queue must be retrieved. This can be done by using the `at.ac.uibk.dps.biohadoop.tasksystem.queue.TaskQueueService`, which is a singleton and provides methods as seen in listing 1.7. Please keep in mind that the rest of this section is for advanced users only and may be skipped.

Listing 1.7: TaskQueueService

```
1 public static TaskQueueService getInstance();
2 public TaskQueue getTaskQueue(String name);
3 public void stopAllTaskQueues();
```

- `getInstance()`: is used to get an instance of the `TaskQueueService` singleton.
- `TaskQueue<R, T, S> getTaskQueue(String name)`: this method provides a reference to a task queue with the given name. If no such task queue exists, a new one is created.
- `stopAllTaskQueues()`: this method stops all task queues and should not be invoked by an algorithm implementor. It is used internally by Biohadoop in the shutdown process. When this method is called, all workers get notified to shut down. In addition, the task queue is removed from the internal storage and if there are no other references to this queue, it is made available for garbage collection.

By invoking `getTaskQueue(String)` on the `TaskQueueService` singleton, a reference to a generic task queue is retrieved, which is of type `at.ac.uibk.dps.biohadoop.tasksystem.queue.TaskQueue`. The task queue offers the methods in listing 1.8. The `add...()` methods are useful to submit new tasks, the `stopQueue()` can be used to stop the queue and all of its depending workers. The other methods are usually used by adapters. When invoking the `add...()` methods, new objects of type

`at.ac.uibk.dps.biohadoop.tasksystem.queue.ClassNameWrappedTask` are created, that wrap the actual data, the information about which class to use to compute the result and a unique id. Those task objects can be retrieved by using the `getTask()` method.

The generic types of the `TaskQueue` are T, which denotes the type of the task data, S denotes the type of the return value and R denotes the type of the inital data.

Listing 1.8: TaskQueue

```
1 TaskFuture<S> add(T data, String asyncComputableClassName, R
      initialData) throws InterruptedException
2 List<TaskFuture<S>> addAll(List<T> datas, String
     asyncComputableClassName, R initialData) throws
     InterruptedException
3 List<TaskFuture<S>> addAll(T[] datas, String
     asyncComputableClassName, R initialData) throws
     InterruptedException
4 Task<T> getTask() throws InterruptedException
5 R getInitialData(TaskId taskId) throws TaskException
6 void storeResult(TaskId taskId, S data)
7 void reschedule(TaskId taskId) throws InterruptedException,
     TaskException
8 void stopQueue()
```

- `add(T data, String asyncComputableClassName, R initialData) throws InterruptedException`: adds `data` of type T to the task queue, where `asyncComputableClassName` is the name of the class that should be used by the workers to compute this task. `initialData` is the data that is send to a worker when it first encounters this `asyncComputableClassName`. The method returns a `TaskFuture` object that represents the result of the computation (see chapter 1.3.2 for more information about the `TaskFuture`). This method may throw an `InterruptedException`, if the underlying queue is interrupted while the task is inserted.

- `addAll(List<T> datas, String asyncComputableClassName, R initialData) throws InterruptedException`: adds a list of `datas` of type T to the task queue, where `asyncComputableClassName` is the name of the class that should be used by the workers to compute this tasks. `initialData` is the data that is send to a worker when it first encounters this `asyncComputableClassName`. The method returns a list of `TaskFuture` objects, each one representing the result of a computation.

This method may throw an `InterruptedException`, if the underlying queue is interrupted while the task is inserted.

- `addAll(T[] datas, String asyncComputableClassName, R initialData) throws InterruptedException`: adds an array of `datas` of type T to the task queue, where `asyncComputableClassName` is the name of the class that should be used by the workers to compute this tasks. `initialData` is the data that is send to a worker when it first encounters this `asyncComputableClassName`. The method returns a list of `TaskFuture` objects, each one representing the result of a computation. This method may throw an `InterruptedException`, if the underlying queue is interrupted while the task is inserted.
- `getTask() throws InterruptedException`: gets the next `Task` object out of the task queue and returns it as result. This method may throw an `InterruptedException`, if the underlying queue is interrupted while the task is retrieved.
- `getInitialData(TaskId taskId) throws TaskException`: this method gets the initial data for a task with the given `TaskId`. The `TaskId` is an internal id that is unique for every task. `Task` objects, that are returned by the `getTask()` method, encapsulate this `TaskId`.
- `storeResult(TaskId taskId, S data)`: is used to store the result of the computation for the task with the given `TaskId`.
- `reschedule(TaskId taskId) throws InterruptedException, TaskException`: if something goes wrong while computing the result for a task, this task can be rescheduled, so it can be computed again. This method is useful e.g. in error conditions. It may throw an `InterruptedException`, if the underlying queue is interrupted while the task is inserted. It may throw a `TaskException` if the given `TaskId` is unknown to this task queue.
- `stopQueue()`: stops this queue and notifies all workers to shut down.

As one can see, the methods offered by the `TaskQueue` are rather low level. Direct usage of the `TaskQueue` is advised only if Biohadoop should be extended for example with new task submitters or new adapters.

1.3.4. Adapter

An adapter takes tasks out of a queue and passes it to waiting workers through some kind of communication facility. The results returned by the workers are

than handed back to the algorithm. So the most important aspect of an adapter is to communicate with the corresponding workers. This communication can be of any kind, and its technical details are hidden from an algorithm implementor. Already implemented examples for such communication types are HTTP, Kryo, Socket, WebSocket, but also simple variable sharing when the communication is done between different threads in the same JVM. Section 1.4 talks in greater detail about the communication aspects.

As explained in the previous sections, there may be several task pipelines. Each pipeline consists of task submitters, a queue, the adapters and workers. Because of this rather tight coupling of queue, adapters and workers, adapters must be configured in a way, such that they know from which queue they should take the work. The default pipeline needs no configuration, for the configuration of a dedicated pipeline, take a look at section 1.6. There is no restriction on the number of running adapters.

The communication types provided by Biohadoop should be enough for the average needs. But if one wants to use a different communication type, an implementation of the adapter and worker is needed. For the adapter side of the communication, this can be done through implementing the interface `at.ac.uibk.dps.biohadoop.tasksystem.adapter.Adapter`, shown in listing 1.9. Be aware that the methods in this interface are called by Biohadoop, so there is no need to call them somewhere manually, they are part of the life cycle of an adapter. This life cycle methods are called by Biohadoop in a sequential manner throughout all configured adapters, if one call blocks, the whole Biohadoop system blocks. The communication between adapter and workers is not part of this interface and can be designed as needed.

Figure 1.4 depicts the typical life cycle of an adapter. If any `configure(String)` or `start()` method throws an exception (in contrast to the figure), Biohadoop terminates at this point.

Listing 1.9: Adapter interface

```
1 public interface Adapter {
2   public void configure(String pipelineName) throws
        AdapterException;
3   public void start() throws AdapterException;
4   public void stop();
5 }
```

- `void configure(String pipelineName) throws AdapterException`: this method is called by Biohadoop when it prepares all the configured adapters. It provides a `pipelineName`, that defines the pipeline to which

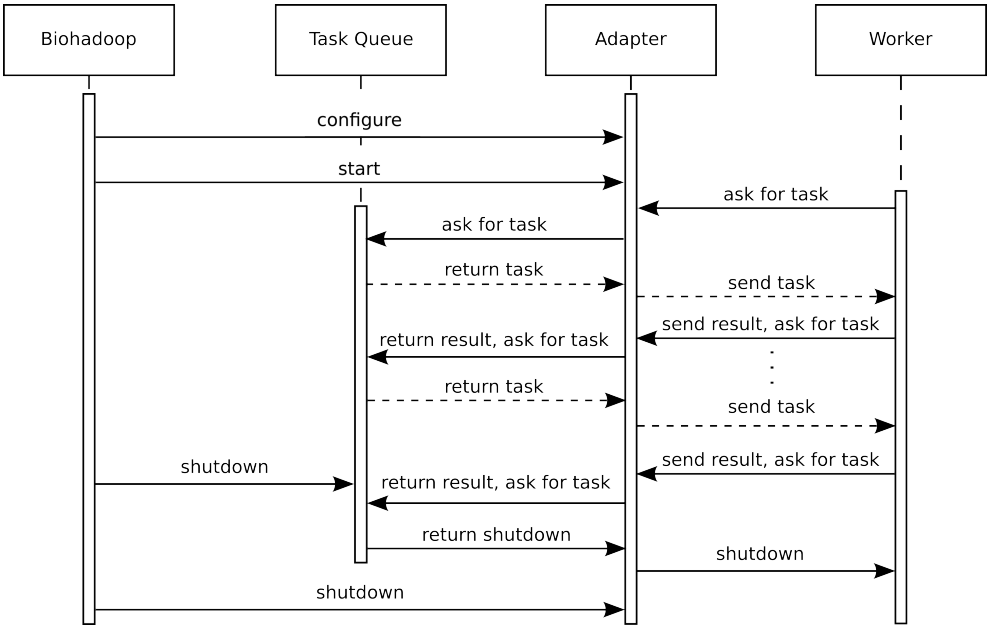

Figure 1.4.: Life cycle of an adapter

the adapter belongs. In addition, this is by convention the queue name, from which the adapter should consume the tasks. In this method, an adapter may initialize its behavior, but it should not start working (e.g. open a socket). It may throw a `AdapterException`, which leads Biohadoop to shutdown completely. Any other uncatched exception also leads to a complete shutdown. This is done to comply with the fail fast principle of application behavior [4], which leads in this case to a system that is either running fully functional or not running at all.

- `void start() throws AdapterException`: this method is called by Biohadoop to start all of the configured adapters. It is the right place to start up the adapters part of the communication, e.g. to open a listening socket. The method may throw a AdapterException which leads to a complete shutdown. Again, every uncatched exception also leads to a complete shutdown.

- `void stop()`: This method is called by Biohadoop after all algorithms are finished, no matter if they threw an exception or not. In this method, the communication facility (e.g. socket) should be closed.

1.3.5. Worker

The workers implement the asynchronous (and possibly parallelized) part of the algorithm and actively request new tasks from the adapters. If the adapters have currently no work to offer, for example because the running algorithms have not submitted any tasks to the task system, then the workers wait until new work is available. Of course, the workers need some resources during the waiting times too (in terms of CPU, RAM, storage and so on), so when running Biohadoop, one should consider how much workers are really needed and at which point more of them are counterproductive.

There are two different kinds of workers: the ones that run under the control of the Apache Hadoop system, (from now on called embedded workers) and the ones that run from outside of this system (from now on called external workers), Biohadoop supports both forms. Embedded workers must be configured in Biohadoop, but then the whole startup and shutdown is done automatically. In contrast, external workers don't have to be configured by Biohadoop, but their whole life cycle must be controlled in some other way, which is not part of Biohadoop. This can pose some problems, as external workers need to know e.g. when and where Biohadoop is running and how to connect to its running adapters. If there are no special needs, embedded workers are just fine. There are although some good reasons to use external workers:

- external worker don't necessarily depend on the Hadoop ecosystem, they may run wherever they want, as long as they are able to communicate to at least one adapter. It is possible to develop external workers, that run in completely different environments for example mobile phones.
- there is no restriction on the program language for an external worker, as long as it knows how to talk to at least one adapter. For example it is possible to implement a worker in JavaScript [5] or Python [6]. In contrast to this, embedded workers have to be written in Java.
- there is no limit in the number of external workers that may run. With suited algorithms to solve, this can lead to enormous scale. The algorithm should be suited in the sense, that the tasks are not to short running, as this would lead to high network traffic, effectively making this the bottleneck. The number of embedded workers is limited by the Hadoop environment, on which Biohadoop runs.

Crucial to external workers is the fact, that they have to know how to communicate to the adapters. As it is not always easy to simulate e.g. the Java serialization mechanism in other languages, there are different kinds of

adapters, that run different types of communication facilities. The most straight forward type of communication is using HTTP, a protocol for which there is an implementation in almost every language of the world. This way it is possible to implement external workers in a broad range of programming languages.

!!!!Workers don't get restarted if they fail!!!!

Embedded workers must implement the interface `at.ac.uibk.dps.biohadoop.tasksystem.worker.WorkerEndpoint`, that is shown in listing 1.10.

Listing 1.10: Worker interface

```
1 public interface WorkerEndpoint {
2   public String buildLaunchArguments(WorkerConfiguration
        workerConfiguration) throws WorkerLaunchException;
3   public void configure(String[] args) throws
        WorkerException;
4   public void start() throws WorkerException,
        ConnectionRefusedException;
5 }
```

- `String buildLaunchArguments(WorkerConfiguration workerConfiguration) throws WorkerLaunchException`: this method is called by Biohadoop before the startup of the worker. The return value is a String, that contains needed configuration options for the worker to start. This string is provided to the worker at its startup. The reason for this method to exist is, that it is called inside the Biohadoop process, which opens the possibility to make use of Biohadoop internal configuration options and variables. After an embedded worker is started by Biohadoop (using the Hadoop Yarn capabilities) it runs in a separated process on perhaps a different machine, and has no easy way to access the internal Biohadoop state. The method may throw a `WorkerLaunchException`, after which the whole Biohadoop system shuts down, in order to adhere to the fail fast principle. Every other uncatched exception also leads to the complete shutdown of Biohadoop.

- `void configure(String[] args) throws WorkerException`: this method is called, after the embedded worker is instantiated i.e. when it is already running as a separate process on perhaps a different machine. The method takes an array of Strings as input parameters, which is exactly the array of Strings used to instantiate the worker process, or in other words, these are the parameters provided to the static `main(String`

`args[])` method of the worker at startup. The arguments contain, beside the arguments of the `buildLaunchArguments(WorkerConfiguration)`, additional values, that may be useful, e.g. the path to Biohadoops config file. The method may throw a WorkerException, after which this worker is terminated and an error code of 1 is returned to Biohadoop.

- `void start() throws WorkerException, ConnectionRefusedException`: this method starts the main loop of the worker, where it begins to request tasks from the corresponding adapters. Usually, the adapter notifies the worker when to shutdown, which leads the worker to exit this method and to terminate its process with an exit code of 0. The method may throw a WorkerException after which this worker is terminated and an error code of 1 is returned to Biohadoop. It may throw a ConnectionRefusedException if it is unable to connect to its adapter. In this case, the worker shuts down, and an error code of 2 is returned to Biohadoop. Every other uncatched exception also shuts the worker down and returns an error code of 1 to Biohadoop.

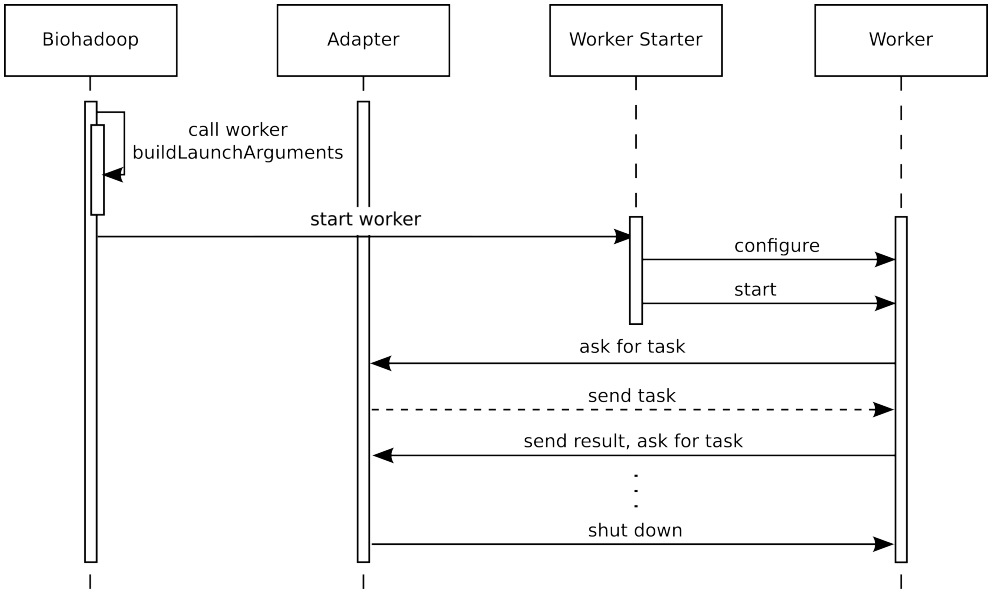

Figure 1.5.: Life cycle of a worker

The typical life cycle of a worker is depicted in figure 1.5. It begins at the Biohadoop instance, where the `buildLaunchArguments(WorkerConfiguration)` method is called to get the necessary parameters. After that, Biohadoop launches for every worker instance a Yarn container and

starts the `at.ac.uibk.dps.biohadoop.tasksystem.worker.WorkerStarter` inside it. The `WorkerStarter` is responsible to call the `configure(String[] args)` and `start()` method of the worker. If an exception occurs, it is the responsibility of the `WorkerStarter` to terminate the work and return a corresponding exit value to Biohadoop. Please note, that figure 1.5 lacks information about the steps that are necessary to instrument the adapters. Information about those steps can be found in section 1.3.4.

Inside the workers `start()` method, there is a main loop, where the worker asks for new tasks and submits the results of the computation to the adapters, which return them to the algorithm. The workers then wait for new tasks or for a shut down information from the adapters, which leads to its termination.

By making the workers actively ask for new work, the adapters don't have to keep track of the workers. This way, the number of workers has no real limit. As long as a worker is able to connect to a corresponding adapter (where they have to agree on the protocol, communication flow and exchanged data, see section 1.4.2 for more information), it may ask for work and return its results.

!!!provide and describe implementation for AsyncComputable for Sum algorithm!!!

1.3.6. AsyncComputable

In the previous sections, the task system was introduced, but it wasn't mentioned, how the results for submitted tasks are computed. This is done, by defining classes that implement the generic interface `at.ac.uibk.dps.biohadoop.tasksystem.AsyncComputable<R,T,S>`. Each class that implements this interface declares, that it can be executed by workers. When an algorithm submits a new task, it has to define which `AsyncComputable` should be used for computing the result for the task. This information is then send, along with the task itself, to a worker, where the result is computed and then returned.

In listing 1.11, the method for the `AsyncComputable` interface is shown, where R is the type of the `initalData`, T is the type of input data and S is the type of the result.

Listing 1.11: Classes that implement the AsyncComputable interface can be executed by workers

```
1 public interface AsyncComputable<R, T, S> {
2   public S compute(T data, R initialData) throws
       ComputeException;
3 }
```

- `S compute(T data, R initialData) throws ComputeException`: The code inside this method is executed by the worker. It gets `data` of type T and `initialData` of type S as arguments. The `data` represents the data for this task, `initialData` is some static data that doesn't change over time and that can be therefor cached by the workers, once they received it the first time. When the computation has finished, the result of type S is returned to an adapter and a new task is requested by the worker. The `compute(T, R)` method may throw a `ComputeException` at any time, indicating that something went wrong during the computation. This exception is logged by the workers, after that they terminate.

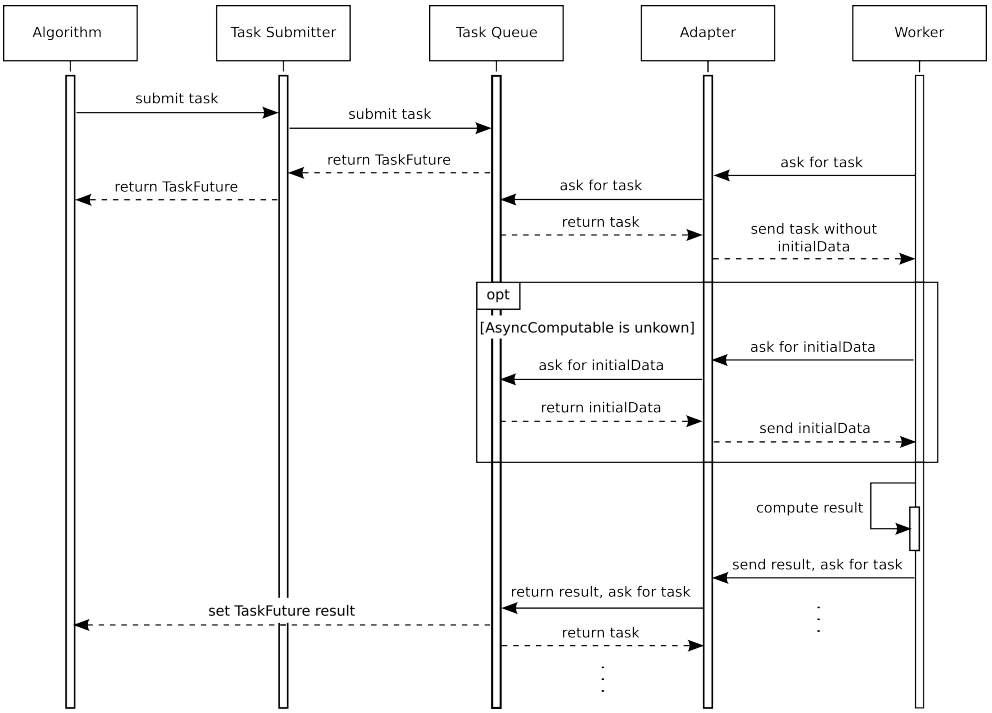

Figure 1.6.: Lifecycle of a task, computed by a worker using an AsyncComputable

Figure 1.6 depicts the general sequence of action for an asynchronous computation: a task is submitted to the task queue by an algorithm. As seen in section 1.3.2, this should be done using a `TaskSubmitter`. On submission, an `AsyncComputable` class must be declared, which is used by workers to compute the result for the task. At some point in time, an adapter takes the task out of the queue, together with the description of its related `AsyncComputable`. This step is triggered by an incoming worker request. The task, together with the

description of its related `AsyncComputable`, is packed into a message and send to the requesting worker. On the worker side, the message is received and analyzed. Then, the result of the task is computed using an instance of the related `AsyncComputable`. The result is returned to the adapter and a new task is requested.

There is one more step included, if the worker encounters an `AsyncComputable` that it hasn't seen before. In this case, the worker requests the `initialData` for this `AsyncComputable`, and only after it receives this data, it can proceed in computing the result of the task. The `initialData` should be cached by the worker, to minimize the communication overhead. All worker implementations provided by Biohadoop work this way.

1.4. Communication

In Biohadoop, algorithms can take usage of the task system to distribute their computation onto several waiting workers. As seen in the previous chapters, the task system consists of the submitters, a queue, adapters and workers. The submitters, queue and adapters all work in the same JVM and therefor in the same process. The communication between these parts is not difficult, it is just a matter of reading and writing shared variables between the different threads, possibly protected by concurrency protocols (for example, the queue is based on Java's `LinkedBlockingQueue`, which is a thread safe queue that supports many writers and readers).

The communication between the adapters and workers is more complicated, as the adapters and workers may run in different processes, or even on different machines, so they can not rely on sharing variables between some threads. A more sophisticated method of communication must be used.

There are many different ways of communication between different (remote) processes. Biohadoop provides five different communication protocols, that can be used to transfer the data between adapters and workers. The reason to not stick to just one implementation was, that each one has its advantages and disadvantages. As the whole communication process is hidden from the algorithm by the task system, changing the communication facility is just a matter of configuration, which makes it easy to try out different protocols and decide later on about which one to use.

If there is the need for different, yet not implemented protocols, Biohadoop provides a way to implement its own protocols by implementing the appropriate parts of `Adapter` (section 1.3.4) and `Worker` (section 1.3.5). The adapters and

workers have to agree about the protocol, the serialization of the data and the communication flow.

An introduction to the provided protocols is given in section 1.4.1. In section 1.4.2, the communication flow for the provided protocols is shown. It defines the steps involved for the communication between the adapters and the workers. Those steps are the same for all provided protocols. If one wants to implement its own communication mechanisms, it doesn't have to stick to the presented control flow and is free to choose its own one.

1.4.1. Protocols

In this section, the five different communication protocols of Biohadoop are given, each one has its advantages and disadvantages.

HTTP

Biohadoop provides an implementation of adapters and workers, that can communicate to each other by using HTTP. The adapter makes the tasks available as resources under a given URL. The URL is composed by the server name and port where the resource is available and the path `rs`, which is followed by the name of the task pipeline as subpath. For the default pipeline this would be for example `http://example.org/rs/DEFAULT_PIPELINE`. The workers take the tasks from this URL, compute the results and return those results to the same URL by using the `POST` verb.

The main advantage of the HTTP protocol is, that it is in broad usage and that there exists an implementation for it in almost every language. This makes it very easy to write external workers in the desired language. The disadvantage of using HTTP is its speed, as there is a lot of overhead involved when sending HTTP requests over the wire (for example the HTTP headers).

For serialization purposes, the JSON format is used. This is a very common format and suitable for machine communication, but also understandable for the human reader. Most programming language support JSON.

Kryo and KryoNet

Kryo [7] is a library for high speed serialization of Java objects. It is usually faster than the build-in serialization features of Java. The developers of Kryo provide also the library KryoNet [8] for communication between different remote entities. This library is used by Biohadoop as one type of its provided protocols. The advantage consists of its speed, at least for the serialization.

But as there where some problems when using KryoNet, the achieved communication speed is somewhat slower than with most of the other protocols. The problems are related to the asynchronous nature of KryoNet, where all requests are handled by exactly one event loop. If one request blocks (e.g. because it is waiting for new tasks), all other requests block too. To circumvent this, the requests are dispatched to special threads, but this slows the whole system noticeable down.

Another disadvantage of Kryo and KryoNet is that they are custom serializers and protocol, that are not in broad usage. This restricts the worker implementations to be written in Java, which may not be a problem at all, especially if only embedded workers are used. But if external workers should be used, they have to be written in Java.

Each class, that should be submitted between adapters and workers using Biohadoops Kryo protocol, must be registered with Kryo. The registration can be done by implementing the `at.ac.uibk.dps.biohadoop.utils.KryoRegistrator` interface in a registration object. This object then has to be made known to Biohadoop, by specifying its full class name in the `globalProperties` section of the configuration file (see section 1.6 for more information about this topic). The name of the property must be `KRYO_REGISTRATOR`. As an example, lets assume, that the full name of the configurator class is `com.biohadoop.KryoConfigurator`, then the property that needs to be set in the `globalProperties` section would be:

```
1 "KRYO_REGISTRATOR" : "com.biohadoop.KryoConfigurator"
```

When this parameter is set in the configuration file, Biohadoop knows that the classes, that are listed inside the configurator class, should be registered with Kryo.

Socket

When using the socket protocol, the communication and serialization is performed by the standard Java libraries. The communication is done using sockets, the serialization by the standard Java serializer. Each class, that should be submitted between adapters and workers using the socket protocol, must implement at least the `java.io.Serializable` marker interface. This is necessary for the standard Java serialization mechanism to succeed. As this serialization mechanisms can be somehow slow, there is the additional possibility, that a class implements the `java.io.Externalizable` interface, where the serialization has to be done manually.

The advantage of the socket protocol is its very good performance, and that it is just a plain socket. Most languages provide mechanisms to connect to a socket, so this should be a rather easy task.

The disadvantage of Biohadoops socket protocol is its dependence on the Java serialization, which restricts the workers to be written in Java or in a language that provides implementations for the Java serialization. This disadvantage may be mitigated by the possibility to implement a custom serialization mechanism by implementing the `Externalizable` interface.

WebSocket

WebSockets can be used for bidirectional communication. They were first used as an additional type of communication between a web application and an application server. They rely on the HTTP protocol for the handshake, during which the communication partners agree to upgrade to the WebSocket protocol. After the upgrade is done, the communication between the endpoints (in this case between the adapters and workers) can be performed using binary or text streams. Unlike in the plain HTTP protocol, the HTTP headers are not transmitted with every message, which reduces their size.

Like in HTTP, the adapter makes the tasks available as resources under a given URL. The URL is composed by the server name and port where the resource is available, the path `ws`, which is followed by the name of the task pipeline as subpath. For the default pipeline this would be for example `http://example.org/ws/DEFAULT_PIPELINE`

Unlike the HTTP protocol, for WebSockets there is no need to make a distinct request for every exchanged message. The worker just connects once to the URL and can then exchange all messages through this established connection.

As already mentioned, the HTTP headers are omitted when WebSockets are used. This makes the data transfer faster than with HTTP, which is one of the advantages of using the WebSocket protocol in Biohadoop. Another advantage is, that WebSocket implementations are now available in many programming languages, making them available to a broad range of users, that may implement workers in their desired language. The disadvantage is that, if no WebSocket implementation exists in a language, it is harder to write a client, than e.g. for HTTP. Also, as WebSockets are a relatively new type of communication protocol, some available libraries may still contain bugs.

JSON is used as the format for the data serialization, most languages provide parsers for it.

Local

The local communication can only be used with local workers. Those are workers, that run inside the same JVM (and therefor process) as Biohadoop and the algorithms. This makes it very easy to submit tasks and return results between adapters and workers. The data transfer is simply done by accessing shared variables. No serialization is needed, which greatly speeds up the communication between the adapters and workers. The access to the shared variables must be synchronized, as there may be several adapters and workers. This is done through standard Java mechanisms.

The main advantage of the local protocol is its speed. As a process may use as much computational resources as available and because current hardware may provide a lot of shared CPU cores, the local protocol and workers may even scale to a sufficient degree for many tasks. But when greater scalability is needed, or when there are not enough computational resources available on the machine that runs Biohadoop, remote workers are needed, that rely on other communication protocols than the local one. Therefor the biggest disadvantage of local workers is, that they can run only on the local machine, where also Biohadoop runs.

1.4.2. Communication flow

Each protocol presented in the prior section has to use some kind of communication flow between the corresponding adapters and workers. This communication flow, depicted in figure 1.7 is the same for all protocols that are provided by Biohadoop. If one wants to write a worker for one of the provided protocols, it must adhere to the communication flow presented here. Custom protocols don't need to reuse this, they are free to implement their own communication pattern.

As one can see in figure 1.7, the communication between adapters and workers is initialized by the worker. Also, after each iteration, the worker actively asks for new tasks.

After the worker gets a task from the adapter, it has to control, if it already has the needed `initialData`. This is done by looking at the class name of the `AsyncComputable`, which is send along with the task. If the class is unknown, the worker asks the adapter for the `initialData` for the task. The adapter responds with the `initialData`, that is then cached by the worker, in case that the result for another task with the same `AsyncComputable` should be computed. Then, the worker computes the result for the task using the tasks data, the `AsyncComputable` and the `initialData`. The result is then retransmitted to the adapter and, at the same time, a new task is requested. Then the loop starts from the beginning.

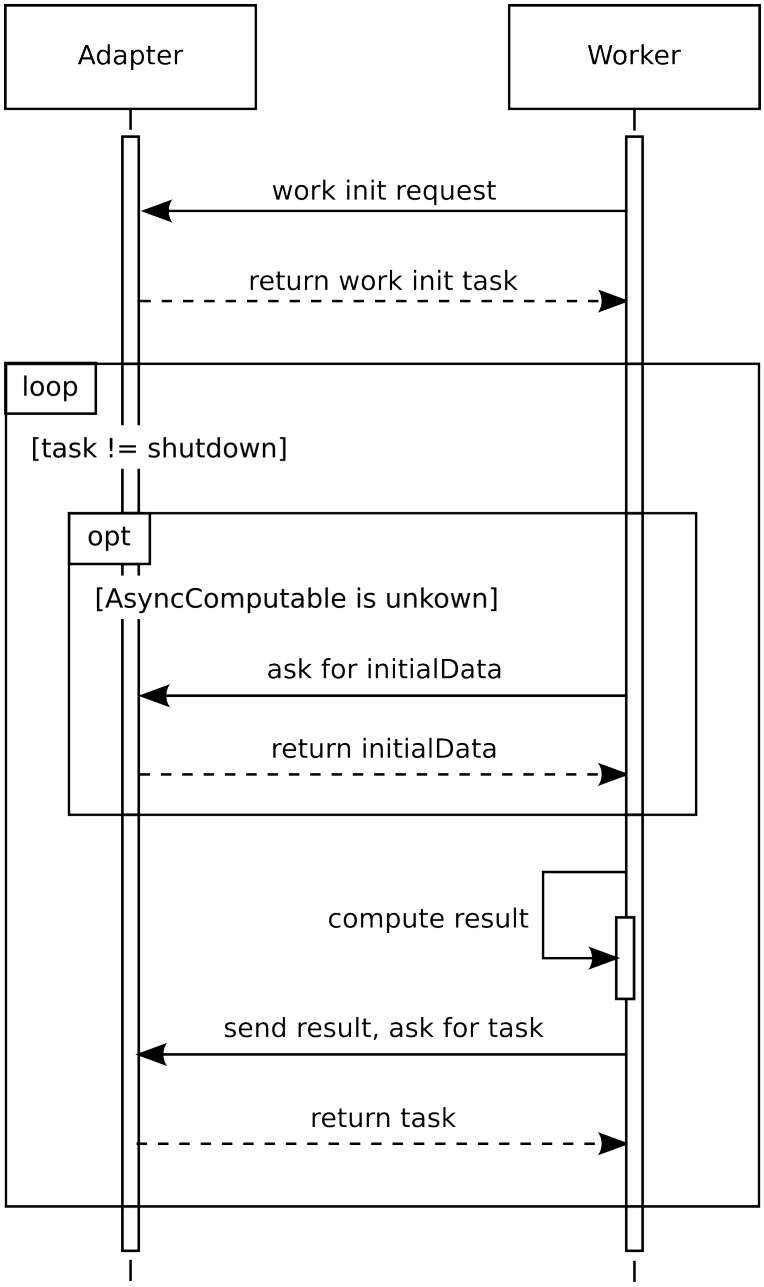

Figure 1.7.: Communication flow

If the answer of the adapter consists of a shutdown message, the worker disconnects from the adapter and does the shutdown.

The implementation of the presented communication flow has one big problem: a worker doesn't recognize when the same `AsyncComputable` is used for

different kinds of tasks with different `initialData` values. For example, lets assume that we have two different tasks, `TASK_A` and `TASK_B`. Each task uses the same `AsyncComputable` for its computation, but a different `initialData` value. Lets further assume, that a worker first encounters `TASK_A`. In this moment, the worker will look up internally, if it has already the `initialData` for the `AsyncComputable`, that comes along with the task. In this case, this is not true, as the worker encounters this `AsyncComputable` the first time. The worker asks the adapter for the `initialData` and after its retrieval it stores this data internally. Then it computes the result for the task and returns it to the adapter. Now lets assume, that the worker gets next the `TASK_B`. As the `AsyncComputable` is the same as for `TASK_A`, and the worker looks up the `initialData` based on the `AsyncComputable`, the worker believes that it knows already the right `initialData`, which is wrong.

This problem can be solved by transmitting a hash value of the `AsyncComputable` and `initialData`, in addition to the normal task data, and letting the worker decide, based on this hash, if it needs to fetch the `initialData`. This is not a big code change, and is on top of the TODO list for Biohadoop.

1.5. Enhancements

Beside the task system, that is presented in greater detail in section 1.3, Biohadoop provides two enhancements for algorithm authors. One is about persistence, where it is possible to load and store some data to disk (see section 1.5.1). The other is about high level parallelism between parallel running algorithms, called the "iceland model", that is sometimes used by optimization algorithms to enhance the solution (see section 1.5.2).

1.5.1. Persistence

There are a lot of reasons for an algorithm to store its data to a disk. The most important may be, that one wants to store the result of a computation. The next reason is, to store the computational work that is done until a certain point, in case something happens. If this data can be reloaded afterwards, the computation can be continued from that point on. Another reason may be, that the intermediate results are needed for some other computation or visualization.

So we see, that some kind of persistence is useful. It should include both the saving and loading of data. Biohadoop provides this kind of service by offering a simple API. Of course, an algorithm author may opt to use its own mechanism of data storage and retrieval, but the presented API has the advantage, that it is able to run both on local file systems and on Hadoops HDFS.

The file saving is performed by using the class `at.ac.uibk.dps.biohadoop.persistence.FileSaver` and its provided static method, as presented in listing 1.12.

Listing 1.12: Save method exposed by `FileSaver`

```
public static void save(SolverId solverId, Map<String,
    String> properties, SolverData<?> solverData) throws
    FileSaveException;
```

- `static void save(SolverId solverId, Map<String, String> properties, SolverData<?> solverData) throws FileSaveException`: This method saves the data, provided by the `solverData` argument, to a file. The path for the file consists of the path that is defined in the configuration file under the algorithms private property with name `FILE_SAVE_PATH`, enhanced with the `solverId`. The filename consists again of the `solverId`, the iteration count that is submitted through the `SolverData` and the timestamp, at which the saving is done. The elements of the filename are separated by the character _.

  So, for example, lets assume that the path defined in the configuration file is `/biohadoop/save`, the unique `solverId` is a UUID and has the value `08b1bc4c-e6d9-44ec-ba5d-24801f6c2339`, the iteration count is 100 and the timestamp, at which the saving is done, has the value `1411588421915`. Then the data, submitted with the `solverData` argument, is saved in the file:

  ```
  /biohadoop/save/08b1bc4c-e6d9-44ec-ba5d-24801f6c2339/08
      b1bc4c-e6d9-44ec-ba5d-24801f6c2339_100_1411588421915
  ```

  The saved data is stored in the file using the JSON format, which is a nice format for data serialization, that is available in many different programs, programming languages and environments.

  The method `save(SolverId, Map<String, String>, SolverData<?>)` throws a `FileSaveException` in the following cases:

  - the `FILE_SAVE_PATH` property is not set
  - the save path can not be created
  - a file with the given name already exists
  - there was an error during the save operation

  It is the responsibility of the algorithm author, to act appropriately to such an exception.

As mentioned, the saving is configured in the configuration file. There are two properties, by which the saving can be adjusted to the personal needs (for more information about the configuration of Biohadoop, refer to section 1.6):

- FILE_SAVE_PATH: This is the initial path, where the files should be stored when they are saved. See the description of the `save(SolverId, Map<String, String>, SolverData<?>)` method above for more information about how the final file name is defined.
- FILE_SAVE_AFTER_ITERATION: Through this property, an algorithm author can decide after how many iterations a file should be saved. This removes from the algorithm author the burden of checking the iteration by hand. The `save` method than compares this value with the value that is provided in the `solverId` argument and decides, if it should persist the data.

To load a file, for example to resume an old computation at the beginning of an algorithm, the class `at.ac.uibk.dps.biohadoop.persistence.FileLoader` can be used. This class provides a static method, presented in listing 1.13.

Listing 1.13: Load method exposed by `FileLoader`

```
1 public static SolverData<?> load(SolverId solverId, Map<
    String, String> properties) throws FileLoadException;
```

- `static SolverData<?> load(SolverId solverId, Map<String, String> properties) throws FileLoadException`: This method loads data from a file that is defined in the private property section of the algorithm under the name `FILE_LOAD_PATH`. This value can be defined in two ways. The first one defines the full path for a file, including the file name, that contains loadable data in a JSON format. In this case, exactly this file is loaded. In the second case, the value defines a directory path where loadable files can be found. In this case, the latest file in the directory is loaded, which is the file that was modified most recently. This can be suitable, when always the newest file of a directory should be loaded.

  The method throws a `FileLoadException` in the following cases:

  - the `FILE_LOAD_PATH` property is not set
  - the file or path can not be found
  - the `FILE_LOAD_PATH` points to an empty directory
  - there was an error during the save operation

It is the responsibility of the algorithm author, to act appropriately to such an exception.

Like in the case of file saving, the loading is configured in the configuration file. There are two properties, by which the loading can be adjusted to the personal needs:

- FILE_LOAD_PATH: This is the initial path, where the files should be stored when they are saved. See the description of the `save(SolverId, Map<String, String>, SolverData<?>)` method above for more information about how the final file name is defined.
- FILE_LOAD_ON_STARTUP: This boolean property defines, if the configured file should be loaded at all (the name is a bit confusing). If this property is not set, a call to `load` returns null as result. This can be convenient, if there is no useful data to load.

1.5.2. The island model

The island model is a high level parallelization model, that is sometimes used in optimization problems. In the island model, there are several optimization algorithms running in parallel, trying to compute the result to the same problem. Those optimizers are called the islands. Each of these optimizers is independent of the others, and each one may have a different solution at a given point of time. By exchanging their data after some intervals, islands may get interesting solutions from other islands, that can be integrated in their own computation to enhance their solution.

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

API

Biohadoop provides an API that can be used to build an island model. This API is exposed through the class `at.ac.uibk.dps.biohadoop.islandmodel.IslandModel` as seen in listing 1.14.

Listing 1.14: Methods exposed by the `IslandModel` class. The class can be used to exchange data between the islands and therefor to construct an island model

```
static void initialize(SolverId solverId) throws
    IslandModelException
static Object merge(SolverId solverId, Map<String, String>
    properties, SolverData<?> solverData) throws
    IslandModelException
```

- `static void initialize(SolverId solverId) throws IslandModelException`: This method must be called by an algorithm to register it at the ZooKeeper service (see below for more information about ZooKeeper). It takes the `solutionId` as its only argument, which is the unique identifier for the algorithm.

  The method throws a `IslandModelException` if the host or port of ZooKeeper is not specified in the global properties of the configuration file. The host is specified through the `ZOOKEEPER_HOSTNAME` property, the port through the `ZOOKEEPER_PORT` property.

- `static Object merge(SolverId solverId, Map<String, String> properties, SolverData<?> solverData) throws IslandModelException`: This method should be called, when the data merging should be done. The `solverId` identifies the algorithm, the `solverData` contains the current solution.

  The properties object is the map configured in the private property part of the algorithm. It uses strings for the keys and the values and must contain the following properties:

  – `ISLAND_DATA_REMOTE_RESULT_GETTER`: This property specifies a class that implements the interface `at.ac.uibk.dps.biohadoop.islandmodel.RemoteResultGetter` with the following method:

  ```
  Object getBestRemoteResult(List<NodeData> nodesData
      ) throws IslandModelException
  ```

  It has the job to get an appropriate remote result from a remote island. To perform this job, a list of `NodeData` elements is provided, that represents all currently known islands. From this list, a suitable solution should be chosen and returned.

  – `ISLAND_DATA_MERGER`: This property specifies a class that implements the merging of the local and a remote solution. This class must implement the generic `at.ac.uibk.dps.biohadoop.islandmodel.DataMerger` interface with the following method, where T is the generic type:

```
T merge(T o1, T o2)
```

The job of this method is to merge the data of the solutions given as `o1` and `o2` into a new solution.

The `merge(SolverId, Map<String, String>, SolverData<?>)` method throws an `IslandModelException` at the same conditions as the `initialize(SolverId)` method mentioned above. In addition, it throws the `IslandModelException` if the `RemoteResultGetter` or the `DataMerger` classes are not defined or can not be instantiated. Also, the `RemoteResultGetter` or the `DataMerger` may throw this exception, if there is something wrong during the data merging. It is the responsibility of the algorithm author, to act appropriately to such an exception.

To make its own solution available to other islands, the data must be explicitly set by the algorithm author. This is done by using the following method of the `at.ac.uibk.dps.biohadoop.datastore.DataClient` object:

```
static <T> void setData(SolverId solverId, Option<T> option,
    T data)
```

It takes as arguments the `solverId`, the second argument in this case must be `DataOptions.SOLVER_DATA`, and the third argument should be the data, wrapped in a `at.ac.uibk.dps.biohadoop.solver.SolverData` object.

ZooKeeper

As mentioned above, Biohadoops island model API takes usage of ZooKeeper [1], which is a Server that provides distributed configuration and synchronization services, and a naming registry. Therefor, a running ZooKeeper instance must be accessible by Biohadoop.

When using the island model, the algorithms first have to register to ZooKeeper by using the `initialize(SolverId)` method of the `IslandModel`. The registration adds an entry to ZooKeeper, that is visible to all other algorithms that use the island model API. The path to the registration is defined by the fixed string `/biohadoop/solvers`, followed by the simple class name of the algorithm and the `solverId` as subpath. For example lets assume that the full class name of our algorithm is `org.example.Sum` and that its `solverId` is `08b1bc4c-e6d9-44ec-ba5d-24801f6c2339`. Then the path, at which the algorithm registers itself to ZooKeeper is

```
/biohadoop/solvers/Sum/08b1bc4c-e6d9-44ec-ba5d-24801f6c2339
```

The registration data consists, again, of the `solverId` and a URL, where the current solution for the algorithm can be found, both serialized using the JSON format. An example for this data can be found in listing 1.15, where the relevant information is on line 1.

Listing 1.15: Example for ZooKeeper registration data

```
1 {"solverId":{"id":"08b1bc4c-e6d9-44ec-ba5d-24801f6c2339"},"
    url":"http://example.org:30000/rs/islandmodel"}
2 cZxid = 0x2e
3 ctime = Wed Sep 24 23:44:22 CEST 2014
4 mZxid = 0x2e
5 mtime = Wed Sep 24 23:44:22 CEST 2014
6 pZxid = 0x2e
7 cversion = 0
8 dataVersion = 0
9 aclVersion = 0
10 ephemeralOwner = 0x148a93c8d61000f
11 dataLength = 106
12 numChildren = 0
```

When the data merging between islands should be done, the defined `RemoteResultGetter` gets a list of all the algorithms that are registered at ZooKeeper, along with their registration data. With this information, that contains also the URL where the current solution for the islands can be found, the `RemoteResultGetter` chooses which remote solution should be used for the subsequent merge.

The remote solutions are made available as HTTP resources at the URL that is part of the registration information, where the complete URL consists of the URL from the ZooKeeper registration with the `solverId` appended, for example:

```
1 http://example.org:30000/rs/islandmodel/08b1bc4c-e6d9-44ec-
    ba5d-24801f6c2339
```

In the end, the whole process of registration to ZooKeeper, retrieval of this information from ZooKeeper and publishing the result of the computation at a defined URL is done by the island model API behind the curtains, so an algorithm author doesn't have to bother with it. The only things an algorithm author has to do is to

- provide its data using the `DataClient.setData(SolverId, Option<T>, T)` method
- use the provided methods of the `IslandModel` object when the merging should be done

- provide an implementation of `RemoteResultGetter` and `DataMerger` to define which data should be merged and how the merging should be done

By using ZooKeeper as central registry for the island model, and by exchanging the data between the islands through HTTP, it doesn't matter if the algorithms run in the same Biohadoop instance (which is possible by defining several instances in Biohadoops configuration file), or if the algorithms run in different Biohadoop instances. The only advantage of running several algorithms in the same Biohadoop instance is, that it guarantees that they all run at the same time. When running several Biohadoop instances on Hadoop, it isn't guaranteed that they also run at the same time, as Hadoop decides when to launch a program. If the algorithms don't run at the same time, then there is no data exchange possible between them, so the island model is reduced to one single island, which means that it makes no difference to running the algorithm without the island model.

1.6. Configuration

Biohadoop uses a configuration file in JSON format. This has the advantage, that it isn't hard for humans to understand and modify its content. It usually is also smaller in size than e.g. XML and parsers exist for a broad range of programming languages. To provide support for automatic schema validation, JSON Schema [9] is used. The full JSON Schema [9] for the configuration file can be found in the appendix A.1.

The path to the configuration file must be given on invocation of Biohadoop as its first parameter (more information on how to run Biohadoop can be found in **??**). If the path is empty or wrong, Biohadoop stops immediately with an `IllegalArgumentException`. If a file is found, its content is parsed and Biohadoop tries to build a configuration object out of it. If there was an error during parsing, Biohadoop stops with an `IOException`, `JsonParseException` or `JsonMappingException`, depending on the reason of the error. Supposing everything went right, the configuration is exposed through the static class `at.ac.uibk.dps.biohadoop.hadoop.Environment` and its `getBiohadoopConfiguration()` method.

Note that the exposure is restricted to the JVM that runs Biohadoop. But this should be enough for most purposes. The algorithms can directly access the `Environment`, although this should be seldom necessary, as the algorithm get their parameters as arguments. Adapters are also able to access the `Environment`. Both mentioned components run in the same JVM as Biohadoop does.

When it comes to the workers and the `AsyncComputable` methods that they run, things are a little bit different. The workers run on different JVMs than Biohadoop does (except the local running workers, that run as separate threads), so they don't have direct access to the `Environment` object. Nevertheless, the path to the configuration file is provided also to the workers, such that they could read and parse the configuration file. But at the time of this writing, there was no reason for a worker or `AsyncComputable` to access the configuration file directly, therefor the access is currently not implemented. Changing this at a later stage is not a big task.

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

- `communicationConfiguration`: defines lists for additional adapters and all workers, that Biohadoop should start. Additional in the sense, that adapters for dedicated pipelines must be configured right here, whereas adapters for the default pipeline don't have to be configured. The workers must be also defined here, for all pipelines including the default pipeline, since there is no useful default configuration for workers.
  - `dedicatedAdapters`: a list of additional adapters. Each element in the list is of type `at.ac.uibk.dps.biohadoop.tasksystem.adapter.AdapterConfiguration`, that contains the canonical class name of an `Adapter` and a string that defines to which pipeline that adapter belongs.
    - `adapter`: canonical class name of the `Adapter`
    - `pipelineName`: string that defines to which pipeline that adapter belongs
  - `workerConfigurations`: a list of workers: Each element in the worker list is of type `at.ac.uibk.dps.biohadoop.tasksystem.worker.WorkerConfiguration`, that contains the canonical class name of a `Worker`, a string that defines to which pipeline that worker belongs and a counter that indicates how many instances of this worker should be started.
    - `worker`: canonical class name of the `Worker`
    - `pipelineName`: string that defines to which pipeline that worker belongs
    - `count`: number of instances of this worker
- `globalProperties`: a map with strings as keys and values. This properties are used for global settings, that should be available in different places.

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

- `solverConfiguration`: a list of algorithms that should be run by Biohadoop. Each element in the list is of type `at.ac.uibk.dps.biohadoop.solver.SolverConfiguration`, that contains the canonical class name of an `Algorithm`, a freely selectable name for it, and a map of properties, that are passed to the algorithm as argument.
  - `algorithm`: canonical class name of the `Algorithm`
  - `name`: freely selectable name, is shown for example in the log files
  - `properties`: a map with strings as keys and values. This properties are used for this instance of the algorithm and should be considered private to it. This properties are provided as second argument to the algorithm, when its `compute(SolverId, Map<String, String>)` method is invoked.

Listing 1.16 illustrates a very simple configuration file for the `Sum` algorithm, introduced in section 1.1. The implementation can be found in [2]. The algorithm sums up all values of an integer array. It takes as input arguments the the number of chunks (`CHUNCKS`) of integer arrays. As second input argument it takes the size of each chunk (`CHUNCK_SIZE`). The whole integer array has then the size `CHUNCKS` $*$ `CHUNCK_SIZE`. The chunks are submitted to the task system, the sum for each chunk is computed on workers and then returned back to the algorithm, where the final summation for all chunks is done.

Listing 1.16: Simple configuration file for the `Sum` algorithm introduced in section 1.1

```
1 {
2   "includePaths" : [ "/biohadoop/lib/", "/biohadoop/conf/"
      ],
3   "communicationConfiguration" : {
4     "dedicatedAdapters" : [ ],
5     "workerConfigurations" : [ {
6       "worker" : "at.ac.uibk.dps.biohadoop.tasksystem.worker
          .KryoWorker",
```

```
7         "pipelineName" : "DEFAULT_PIPELINE",
8         "count" : 0
9       }, {
10        "worker" : "at.ac.uibk.dps.biohadoop.tasksystem.worker
              .LocalWorker",
11        "pipelineName" : "DEFAULT_PIPELINE",
12        "count" : 1
13      }, {
14        "worker" : "at.ac.uibk.dps.biohadoop.tasksystem.worker
              .RestWorker",
15        "pipelineName" : "DEFAULT_PIPELINE",
16        "count" : 0
17      }, {
18        "worker" : "at.ac.uibk.dps.biohadoop.tasksystem.worker
              .SocketWorker",
19        "pipelineName" : "DEFAULT_PIPELINE",
20        "count" : 0
21      }, {
22        "worker" : "at.ac.uibk.dps.biohadoop.tasksystem.worker
              .WebSocketWorker",
23        "pipelineName" : "DEFAULT_PIPELINE",
24        "count" : 0
25      } ]
26    },
27    "globalProperties" : { },
28    "solverConfigurations" : [ {
29      "name" : "SUM",
30      "algorithm" : "at.ac.uibk.dps.biohadoop.algorithms.sum.
          SumAlgorithm",
31      "properties" : {
32        "CHUNKS" : "20",
33        "CHUNK_SIZE" : "10"
34      }
35    } ]
36  }
```

At line 2 we have the configuration of the paths, that have to be included when running Biohadoop. This setting is only important if Biohadoop runs in a Hadoop environment. It is not important if it runs in local mode, for example during development. When running on Hadoop, the files in this directories are made available to Biohadoop.

At line 3 starts the communication configuration, which consists of a list of dedicated adapters and a list of all workers. The definition of dedicated adapters at line 4 is empty, as `Sum` uses only the default pipeline for computation.

If dedicated pipelines were used, than the adapters for this pipeline must be configured here.

Beginning with line 5 we have the declaration of the workers, that Biohadoop should launch. Each entry is defined by the a worker class name, a pipeline to which this worker is assigned, and the number of workers of this instance, that should be launched. We see here for example, that 5 workers of different types are configured, but only the `LocalWorker` at line 10 should be instantiated once. All other workers have a count of 0, which means that no instances of these workers are launched.

At line 27 we have the definition of the global properties. Again, `Sum` doesn't take usage of this configuration possibility. Other algorithms instead use this place for example to specify the configuration for ZooKeeper.

At line 28, the definition for the algorithm starts. In this case, only one algorithm should be started, it is the algorithm with class `SumAlgorithm`. The name of the algorithm is `SUM`, which can be seen e.g. in the log output of Biohadoop. At line 31 there is the configuration of the properties, that are submitted to the `compute(SolverId, Map<String, String>)` method of the algorithm. Those are the properties described before (`CHUNCKS` and `CHUNCK_SIZE`).

It is not always convenient to write the configuration for an algorithm by hand, although it is possible. Biohadoop provides two builder classes, to make it more easy to produce config files. The builder in `at.ac.uibk.dps.biohadoop.hadoop.BiohadoopConfiguration` offers methods to configure the top level elements of a configuration file (`communicationConfiguration`, `globalProperties`, `includePaths`, `solverConfigurations`). The configuration for solvers is simplified by the builder in `at.ac.uibk.dps.biohadoop.solver.SolverConfiguration`. The result of this solver configuration can then be handed over to the `BiohadoopConfiguration` builder. Listing 1.17 shows how the configuration for the `Sum` algorithm can be performed by using the mentioned builders. Note that the outcome of this listing is exactly the config file of listing 1.16.

Listing 1.17: Example on how to produce a configuration file for the `Sum` algorithm using the provided builders. The output of this program can be found in listing 1.16

```
1 public class SumConfigWriter {
2
3   private static final Logger LOG = LoggerFactory
4       .getLogger(SumConfigWriter.class);
5
6   // Define where to write the resulting config file
```

```
7   private static String CONF_OUTPUT_DIR = "/sdb/studium/
       master-thesis/code/git/biohadoop-algorithms/conf";
8   // Define name of the config file
9   private static String CONF_NAME = "biohadoop-sum";
10  // Define full path for config file, that is used for
11  // running Biohadoop locally (e.g. for development)
12  private static String LOCAL_CONFIG_NAME = CONF_OUTPUT_DIR
       + "/" + CONF_NAME
13      + "-local.json";
14  // Define full path for config file, that is used for
15  // running Biohadoop in a Hadoop environment
16  private static String HADOOP_CONFIG_NAME = CONF_OUTPUT_DIR
       + "/"
17      + CONF_NAME + ".json";
18
19  private SumConfigWriter() {
20  }
21
22  public static void main(String[] args) throws IOException,
23      ClassNotFoundException, InstantiationException,
24      IllegalAccessException {
25    // Build solver configuration
26    SolverConfiguration solverConfiguration = new
         SolverConfiguration.Builder(
27        "SUM", SumAlgorithm.class)
28        // Add properties that are private to this algorithm
29        // instance, and that are provided as arguments when
30        // the algorithm is started
31        .addProperty(SumAlgorithm.CHUNKS, "20")
32        .addProperty(SumAlgorithm.CHUNK_SIZE, "10").build();
33
34    // Build Biohadoop configuration
35    BiohadoopConfiguration biohadoopConfiguration = new
         BiohadoopConfiguration.Builder()
36        .addLibPath("/biohadoop/lib/")
37        .addLibPath("/biohadoop/conf/")
38        .addSolver(solverConfiguration)
39        // Add the workers, that should be launched by
40        // Biohadoop to compute the tasks. Note that the
41        // workers, defined by the addWorker(String, int)
42        // method, are part of the default pipeline. Also
43        // note, that if a worker has a count of 0, no
44        // instance of it is started
45        .addWorker(KryoWorker.class, 0)
46        .addWorker(LocalWorker.class, 1)
```

```
47          .addWorker(RestWorker.class, 0)
48          .addWorker(SocketWorker.class, 0)
49          .addWorker(WebSocketWorker.class, 0).build();
50
51      // Save configuration file for Biohadoop running in a
52      /// local environment
53      BiohadoopConfigurationUtil.saveLocal(
            biohadoopConfiguration,
54          LOCAL_CONFIG_NAME);
55      // Save configuration file for Biohadoop running in a
56      // Hadoop environment
57      BiohadoopConfigurationUtil.saveLocal(
            biohadoopConfiguration,
58          HADOOP_CONFIG_NAME);
59
60      // Read results to test if everything is ok
61      LOG.info(BiohadoopConfigurationUtil.readLocal(
            LOCAL_CONFIG_NAME)
62          .toString());
63      LOG.info(BiohadoopConfigurationUtil.readLocal(
            HADOOP_CONFIG_NAME)
64          .toString());
65    }
66  }
```

1.7. BioOozie

- where fits oozie in the picture?
- advantage of using oozie / problems with oozie (not guaranteed that oozie starts parallel task in parallel)
- custom tag / extension
- configuration / why using json rather than XML
- extension possibilities (extend tags in a way that XML configuration can be submitted to biohadoop (e.g. make json out of xml configuration, save to file, start biohadoop))

Appendix A.

A.1. JSON Schema for Biohadoops configuration file

Listing A.1: JSON Schema [9] for the Biohadoop configuration file

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

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

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

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

A.2. Example algorithm: Sum

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

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

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

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

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

A.3. Example algorithm: Sum - AsyncComputable

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

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

A.4. Biohadoop quickstart

The quickstart uses the pre-configured Hadoop environment, that can be found in appendix A.5. This environment is installed during the following steps. In addition, some example algorithms are installed, that can be found at [2].

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

A.4.1. Build and start the Hadoop environment

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

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

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

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

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

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

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

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

A.4.4. Run Biohadoop in the Hadoop environment

To run the Echo example in Hadoop , the following command must be issued in the red terminal - if there is no red terminal, please check [10]. This example assumes that the jar `biohadoop-0.3.0-SNAPSHOT.jar` is used:

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

Other examples of Biohadoop programs can be found at [2]. You can use them to try out Biohadoop and as a template for your own experiments. Good luck and have fun :)

A.5. Pre-build Hadoop environment using Docker

This section provides a way to run a pre-configured Haddop environment using Docker [11]. Docker uses its so called Dockerfiles for its configuration. The here presented solution installs Apache Hadoop 2.5.0, Apache Oozie 4.0.1 and

Apache ZooKeeper 3.4.6 as a cluster environment. The solution is based on `sequenceiq/hadoop-docker` [12].

A.5.1. Build the Hadoop environment

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

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

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

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

A.5.2. Run Hadoop

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

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

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

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

```
1 jps
```

On the master node, it should output:

- DataNode

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

On the slave nodes it should output:

- DataNode
- NodeManager

A.5.3. Stopping Hadoop

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

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

A.5.4. SSH access

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

List of Figures

List of Tables

Listings

Bibliography

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

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

[3] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[4] Jim Shore. Fail fast [software debugging]. *Software, IEEE*, 21(5):21–25, 2004.

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

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

[7] Kryo. https://github.com/EsotericSoftware/kryo. last access: 04.08.2014.

[8] Kryonet. https://github.com/EsotericSoftware/kryonet. last access: 04.08.2014.

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

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

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

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