Multijob: A Framework for efficient Distribution of Evolutionary Algorithms for Parameter Tuning

Robin Mueller-Bady
Martin Kappes
Lukas Atkinson
Frankfurt University of Applied Sciences
Frankfurt, Germany
[mueller-bady,kappes]@fb2.fra-uas.de
fra-uas@LukasAtkinson.de

Inmaculada Medina-Bulo
Universidad de Cádiz
Puerto Real, Spain
inmaculada.medina@uca.es

ABSTRACT

An important challenge in designing evolutionary search heuristics is the statistically significant evaluation of different configurations. The goal is to find an optimal algorithm design with respect to its parameters, i.e., parameter tuning. In this paper, we propose an open source software framework, called *Multijob*, allowing to simplify and automate EA configuration and parameter tuning. Additionally, the framework offers a workflow for distributed execution of the preconfigured algorithms in heterogeneous computing clusters or grids.

The framework uses features of the Unix-based command line utility GNU Parallel, which enables the pausing and resuming of jobs, estimation of experiment completion time, etc. It is highly dynamic due to its language-agnosticism and flexible with respect to parameters and configurations of specific EAs. The possibility of distributing computing time among (heterogeneous) hardware, only requiring access over secure shell (SSH) and a proper environment for job execution, makes *Multijob* a noteworthy utility for improving efficiency of statistically significant parameter testing and tuning.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**; • **Computer systems organization → Grid computing**; • **Software and its engineering →** *Massively parallel systems.*

KEYWORDS

Evolutionary computation, Parameter tuning, Grid computing

ACM Reference Format:
Robin Mueller-Bady, Martin Kappes, Lukas Atkinson, and Inmaculada Medina-Bulo. 2017. Multijob: A Framework for efficient Distribution of Evolutionary Algorithms for Parameter Tuning. In *Proceedings of GECCO '17 Companion.* ACM, New York, NY, USA, 8 pages. https://doi.org/http://dx.doi.org/10.1145/3067695.3082476

1 INTRODUCTION

Finding the optimal configuration for an evolutionary search heuristic is itself a computationally hard problem. It can be observed that most parameters for an evolutionary algorithm (EA) interfere with each other. Therefore, parameter tuning has to be done and evaluated observing the overall evolutionary system and its output.

According to the monothetic analysis in design of experiments theory, i.e., changing only one variable at a time, the amount of possible EA configurations increases massively when multiple parameters need to be considered. This effect is also known as "combinatorial explosion". Additionally, as evolutionary algorithms can be classified as probabilistic search heuristics and thus are non-deterministic, measurement uncertainty has to be reduced while providing statistical replicability. This is done by repeating experiments up to a level where statistical significance can be shown using appropriate statistical tests and methods.

As an example, an EA might be parameterized over its mutation-, recombination-, and selection operations. The EA has 3 discrete parameters with 4 different values each to specify a particular operation. Additionally, each operation had a trigger probability ranging from 0% to 100% in steps of 1%. This simplified EA will result, in case an exhaustive search of parameters is performed on a significance level using 100 repetitions per configuration, in more than 6.6 billion distinct configurations to be executed and evaluated. Assuming a runtime of 1 second per EA execution, which is far from being realistic for many search heuristics solving computationally hard problems, experiments would require more than 200 years to be executed serially.

In this paper, we propose a framework called *Multijob*, which helps in multiple ways to configure and evaluate experiments for new EAs. *Multijob* does not reduce the number of distinct configurations in an exhaustive search, thus, does not replace a proper design of experiments beforehand. However, it eases the configuration, execution, and result handling of EA configurations including repetitions for statistical significance, while handling adequate distribution of jobs in a (possibly) heterogeneous computing cluster or grid. The framework consists of a library for EA configuration generation, and a workflow for distributed execution of these configurations.

The *Multijob* library[1]is written in the Python 3 language[2], and is able to create a list of possible EA configurations according to the experimental design and the significance level that should be achieved. This is done by either providing the discrete parameter values to be tested, or by letting *Multijob* select concrete values from a range given by a start, stop, and (optional) step argument. The library then generates all possible combinations and exports them for processing. The library is also able to decode EA configurations that are provided as command line arguments.

The *Multijob* workflow suggests a strategy for distributed execution of the previously generated EA configurations in a heterogeneous computing cluster or grid using GNU Parallel [16]. This provides information about the overall progress of the experiment such as the estimated time until completion, manages logging and redirection of output and error streams, can pause and resume experiments, and transfer data to and from computing nodes.

In total, the proposed framework provides support for the whole process of evaluation of empirical experiments, from design and configuration of the EA to distributed execution of those configurations and gathering of (individual) outputs and results.

The remainder of this paper is structured as follows. In the following section, the proposed software is set into the context of related work and other software projects. It is followed by a description of the *Multijob* library, and a section about the GNU Parallel-based workflow. We then present an empirical experiment showing the efficiency of the proposed library. Finally, the paper summarizes its results in a conclusion and provides an outlook to possible future features and improvements.

2 RELATED WORK

According to the "no free lunch" theorem [18], each algorithm finding a solution to a specific problem is necessarily worse on other problems. This leads to the fact, that each specific (optimization) problem requires dedicated attention in algorithm design. There exist many software projects targeting this problem, offering features to reduce runtime of individual EA runs or whole experiments by distributing computation and providing libraries containing several well-known and researched operators and schemes.

HeuristicLab [5, 17] is a Microsoft .NET based framework, offering an comprehensive amount of different predefined problems and algorithms, such as genetic programming, machine learning, and multiple different single- and multi-objective evolutionary algorithms. The provided API eases extension of the framework using own problems and heuristics. HeuristicLab aims at executing, analyzing, and understanding those problems and their solving strategies, supported by an extensive graphical user interface and plotting functionality. Within the framework, it is possible to distribute executions of independent algorithms using a Hive server. However, the primarily Microsoft Windows based server infrastructure may be an obstacle for users familiar with other operating systems or without access to proper licensing of the corresponding software.

[1]The *Multijob* source code is available at https://github.com/fg-netzwerksicherheit/multijob and the documentation at https://fg-netzwerksicherheit.github.io/multijob/.
[2]https://docs.python.org/3/

The jMetal framework [4, 14] is a Java-based software project aiming at rapid prototyping of several different multi- and single-objective meta heuristic algorithms. It supports several predefined algorithms, different quality indicators, and an API to develop your own algorithms. However, the native parallelization methods are currently limited to use Java threads on the same CPU. Distribution of computation would require starting multiple instances of the EA under test.

Distributed Evolutionary Algorithms in Python (DEAP) [3, 7] is a framework written in Python which makes extensive use of dependency injection and higher-order functions, making it highly adaptable. Some basic algorithms are already implemented, and the API simplifies creation of new algorithms and methods. While implementation speed is fast in Python and supports rapid prototyping, a major drawback of the framework is the slow execution speed of the language. This is mostly due to the CPython implementation being an interpreter, and further limited by the threading model. Python's global interpreter lock (GIL) prevents threads from running in parallel, which makes them unsuitable for speeding up CPU-bound tasks. Process level parallelism can be used instead, which requires proper interprocess communication, e.g. over pipes or sockets. Due to the strongly enforced dynamic typing in DEAP, enabled through the before mentioned dependency injection and higher-order functions, using specific operators and types in the algorithms is challenging, even using the recommended parallel execution frameworks, e.g., SCOOP [9].

Evolving Objects (EO) [10] is a C++ library which implements several distinct search heuristics, variation-, replacement-, and selection-operators. Furthermore, it implements different visualizations for displaying results and a parallelization using OpenMP [2] and OpenMPI [8]. Beside the fast execution speed and parallelization using the techniques mentioned before, EO is restricted to the C++ programming language, which may be a drawback for researchers aiming for rapid prototyping or working in interdisciplinary fields, as C/C++ require a deeper understanding of the underlying architecture than other languages.

Several other software projects exist, targeting the same issue as the proposed *Multijob* framework. There exist frameworks using Apache Hadoop[3] as a platform for distributed computing [1, 6, 15]. While this promotes rapid prototyping using Java and would allow using any possibly existing and maintained Hadoop cluster, most of the evolutionary algorithms do not require a reduction step and simply omit it. Furthermore, the computational overhead due to the shuffle phase of the MapReduce paradigm and a maintenance of a synchronized Hadoop file system (HDFS) in the cluster require a computationally expensive evaluation function in order to be effective. Thus, an efficient usage of Hadoop for distributed computing of EAs applies to a specific set of problems.

Finally, an architecture for using a combination of Node.js and browser-based JavaScript, NodIO, has been proposed for distributed evaluation of EAs [12]. In the publication it has been shown that performance of the JavaScript implementation is around 30% slower than its Java or Matlab counterpart. Therefore, NodIO may also be useful for specific EAs or rapid prototyping, as the language encourages a fast implementation speed.

[3]http://hadoop.apache.org/

All before mentioned frameworks, architectures, and techniques serve specific goals. While some of them focus on rapid implementation of ideas, others serve the purpose to analyze and understand problems, operators, and algorithms. Furthermore, all of them implement a parallelization or distribution mechanism, custom tailored to the specific purpose the framework serves. The proposed *Multijob* framework sidesteps some of the restrictions of other frameworks by focusing on process-level distribution and parallelization, without prescribing any language or framework for the EA implementation. The EA can be written with any technology that can parse command line parameters. Any number and kind of computation nodes can be used, as long as they can be reached over SSH and have the environment for execution of the specific EA prepared. As the EA and any data files are uploaded to the nodes during the experiments, *Multijob* reduces the administrative effort and allows to easily run and quickly modify distributed experiments running on a cluster.

3 THE HITCHHIKER'S GUIDE TO *MULTIJOB*

In the following, an overview of the proposed *Multijob* framework and its architecture will be given, followed by a detailed description of the individual subparts.

3.1 Overview

As can be seen in Figure 1, *Multijob* is composed of 2 major components: the Job builder and the GNU parallel processor. Both components are connected using an asynchronous queue data structure. The *Multijob* job builder is responsible for creating a set of EA configurations, given a template of an algorithm, the parameters under test, and the level of statistical significance described by the number of repetitions per configuration. In the next step, the generated configurations are enqueued in the processor queue, where the processor distributes them for execution in the defined computing cluster. Distribution is done using either the local machine and its resources in terms of processing power, memory, and storage, or, which may be more attractive with respect to efficiency, one or multiple remote machines, which are accessible via SSH. For both, the local and the remote case, it is necessary that the EA configuration is executable on the target machine, i.e., has all necessary libraries and other requirements available. Finally, the processor aggregates all result data, e.g., found optima, runtime statistics, logs etc., and transfers it back to the machine running *Multijob*. In the following, the individual steps will be described in more detail, explaining the technical details of the underlying architecture.

Multijob separates the parallelization and configuration of multiple EA runs from the EA itself. The EA is implemented in an independent executable (the target executable), which is then invoked by GNU Parallel. By using process-level parallelization, significant flexibility is gained: The target executable does not have to use the same language as the rest of the toolchain, and the processing can be distributed across multiple servers. However, this also requires that all data flow is in a technology-agnostic format. Using command line parameters satisfies this and is highly debuggable. The *Multijob* data flow (as illustrated in Figures 1 and 2) has four stages:

First, the job configurations are generated from the provided parameter ranges. This produces a list of `Job` objects.

Then, these `Job` objects are encoded as command line arguments. This produces a shell script (called `jobs.sh`), which effectively is a job queue. The shell script may use environment variables as placeholder for the target executable, so that the same configuration can be run with multiple executables.

Next, the `jobs.sh` is processed concurrently by GNU Parallel which can run the jobs locally or on remote servers via SSH. The placeholder for the target executable is now resolved by an environment variable. If the job is executed on remote servers, GNU Parallel will need to transfer the target executable and any required input files.

Finally, the command line arguments are decoded inside the target executable. The evolutionary algorithm is then executed with these parameter values, and results are written to output files that can then be analyzed. If the job was executed on a remote server, GNU Parallel will transfer these results back to the controlling host first. Now that all jobs have run, a statistical analysis can be performed over the results.

3.2 Job Generation

Generating the job configurations is easy with the `JobBuilder` class. For each parameter, a range of one or more values can be provided either explicitly, or implicitly as an uniformly distributed range. The number of resulting configurations is the Cartesian product of all parameter value sets. The `JobBuilder` can calculate the number of configurations before the configurations are actually generated. A user can then abort the script if too many configurations would be generated. In our experience, most experiments will not go beyond a few hundred distinct configurations, although *Multijob* does not have or impose any limit itself.

The resulting `Job` objects contain a callback function that is to be invoked when the job is executed, and a table of concrete parameter values. When generating jobs for a shell script, the function cannot be encoded, and will be provided by the target executable instead. A placeholder like `lambda **kwargs: None` will have to be used instead. Each distinct configuration has an ID, repetitions of the same configuration are distinguished by a repetition-id. These IDs are particularly useful for generating unique names of output files.

Repetitions of each configuration are necessary to gain statistical confidence for the results. These repetitions are represented as separate job objects. Two repetitions of the same configuration differ only in their repetition-ID.

An example of job generation is shown in Listing 1. There, a `JobBuilder` object is created which then receives several parameters ("use_injection" set to True and False, "mxpb" defined as $\{0.0, 0.1, \ldots, 1.0\}$, and "popsize" as static values 10, 20, 50, and 100). The Cartesian product of this configuration leads to 88 different configurations. Using the "repetitions" parameter in Line 9 will execute each configuration 20 times for statistical significance, each with a separate ID, leading to 1760 total jobs.

3.3 Encoding and Decoding of Command Line Arguments

To communicate the job parameters to the target executable, a language-agnostic serialization method is required. *Multijob* represents the fields of a `Job` object as simple strings, without using a

Figure 1: The *Multijob* data flow

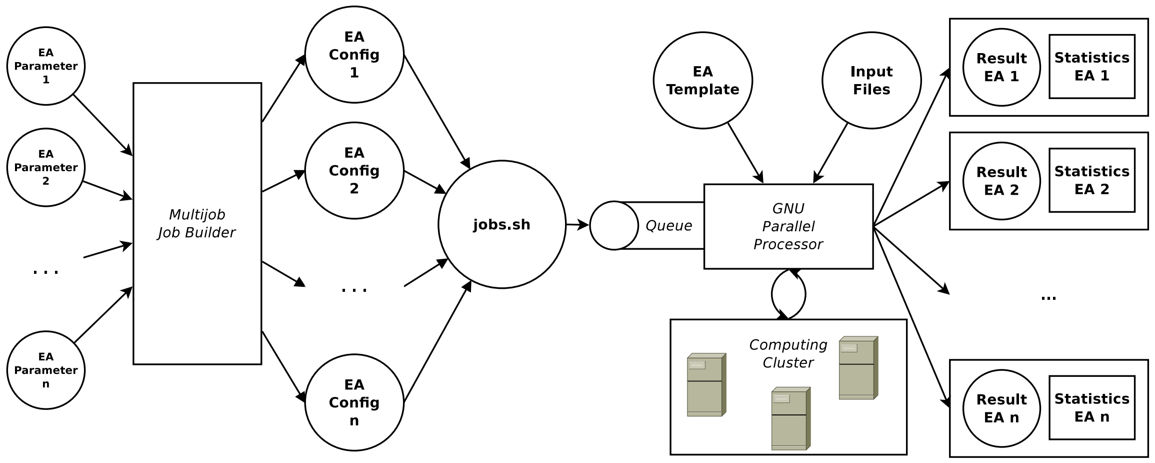
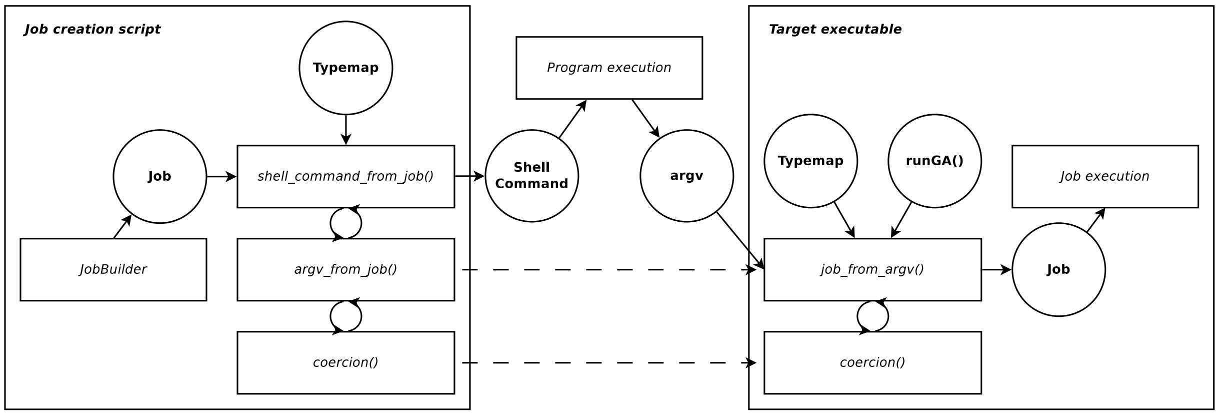


Figure 2: Encoding and decoding the job configuration as command line arguments



```
from multijob.job import JobBuilder
b = JobBuilder()
b.add('use_injection', True, False)
b.add_range('mxpb', 0.0, 1.0, 0.1)
b.add('popsize', 10, 20, 50, 100)
if b.number_of_jobs() > 1000:
  raise RuntimeError('too many jobs')
jobs = b.build(lambda **kwargs: None,
               repetitions=20)
#=> list of 1760 jobs
```

Listing 1: Job generation example

```
from multijob.commandline import \
  shell_command_from_job
for job in jobs:
  print(shell_command_from_job('$RUN_GA', job))
```

Listing 2: Shell script generation

complete serialization format such as JSON. Most EA parameters we are interested in tuning are simple values like integers or floating point numbers, so using Python's default string representation is sufficient.

Should a parameter have a more complex format, a custom coercion for that parameter can be provided via the `typemap` option. A typemap is a dictionary that maps parameter names to coercion functions, so that the default coercion can be overridden.

The full shell command for a job needs to invoke some executable, but the target executable is not known when the job configurations are generated. Therefore, the shell script `jobs.sh` with all jobs generally uses the environment variable `$RUN_GA` as a placeholder.

A sketch of the encoding workflow is depicted in Figure 2. An example of shell script generation can be seen in Listing 2. This produces commands in text format such as `$RUN_GA -id=42 -rep=3 - mxpb=0.5 popsize=20 use_injection=True` per line.

In the target executable, the command line parameters need to be decoded into usable data types. In Python, the command line arguments can be accessed through the `sys.argv` list.

For decoding, a `typemap` is required. The typemap is a dictionary associating coercion functions with parameter names. For decoding,

```
1  import sys
2  from multijob.commandline import \
3    job_from_argv
4
5  def runGA(popsize, mxpb, use_injection):
6    ...  # your EA here
7
8  TYPEMAP = dict(
9    popsize='int',
10   mxpb='float',
11   use_injection='bool')
12
13 # skip the executable name argv[0]
14 job = job_from_argv(
15   sys.argv[1:], runGA, typemap=TYPEMAP)
```

Listing 3: Parameter decoding

```
1  import csv
2
3  def save(res):
4    fname = "result_{}_{}.csv".format(
5      res.job.job_id,
6      res.job.repetition_id)
7    with open(fname, 'w') as f:
8      csv_writer = csv.writer(f)
9      for row in res.result:
10       csv_writer.writerow(row)
11
12 save(job.run())
```

Listing 4: Handling results

these coercions read a value from the string in the command line argument. For common types, named coercions can be used. In some cases, custom coercion functions will have to be created.

During decoding, the job parameters are also associated with the `runGA()` function containing the actual evolutionary algorithm we want to run. The result of the decoding with the `job_from_argv()` function is a `Job` object that has the same parameter values and IDs as one of the jobs generated by the job creation script.

The overview of the decoding process is visualized in Figure 2. An example of parameter decoding is shown in Listing 3. In Line 5, the EA is defined where the usual recombination / mutation / selection process takes place. Line 8 defines the typemap to decode the different parameters given according to their data type, while the final decoding is done in Line 14.

3.4 Running the Job and Writing of Files

Aggregating the parts of the workflow described before, the job can now be executed. The `runGA()` function is expected to return the relevant results, which then have to be written to a file. The filename should use the ID and repetition-ID in order to distinguish separate jobs.

An example is shown in Listing 4, having a `job` object to be run in Line 12, whose results are processed into a *.csv file using the function `save()` as defined in Lines 3 to 10.

4 DISTRIBUTED EXECUTION WITH GNU PARALLEL

GNU Parallel provides a flexible toolkit to run a set of processes in parallel either on the local computer, on a remote host, or distributed across multiple remote hosts.

When we run the job generation script with *Multijob*, we get a `jobs.sh` file describing all jobs. This file could be executed sequentially as a true shell script with the Linux Bash shell. Instead, GNU Parallel views each line as a separate record, but can also execute each record as a shell script snippet. The difference is that GNU Parallel will execute multiple lines in parallel and possibly out of order. In its simplest form, this can be done by piping the jobs into GNU Parallel, and resolving the `RUN_GA` variable to a target executable: “`RUN_GA="python runGA.py" parallel <jobs.sh`”.

Using GNU Parallel in this manner has a number of advantages over other ways to run the jobs, in particular over language-native parallelization methods. For Python, the primary way to parallelize CPU-intensive tasks is the `multiprocessing` module.[4] Compared with it, GNU Parallel seems to be more correct and more mature, especially regarding the premature termination of the jobs with SIGINT (Ctrl-C).

Another interesting feature is that by default GNU Parallel gathers the output of each subprocess, and prints the output upon completion. This can make it much easier to debug the processes. If the performance overhead of this feature is undesirable it can be turned off with `-line-buffer`, but that depends mostly on the output characteristics of the experiment.

For long-running experiments there is a chance that the experiment is interrupted. With the `-joblog` option flag, GNU Parallel will keep track of the incomplete and successfully completed jobs. When after an interruption the experiment is restarted with the `-resume` or `-resume-failed` option, GNU Parallel will continue where the previous run stopped and will not repeat existing results. Only the work of the currently running jobs is lost when the experiment is aborted. This encourages starting experiments early, and optimizing them while they are running.

GNU Parallel can estimate the remaining time of the experiment with the `-eta` option. While this is notoriously imprecise especially before the first set of jobs has completed, we find that the runtime estimate lets us make better decisions as to when an experiment should be performed. For a swift experiment, waiting for the result may be acceptable. If an experiment turns out to take longer, running it over several hours or days, blocks less time.

The most valuable feature is the ability to distribute the jobs across multiple hosts via SSH. As a precondition, this requires that all hosts have a comparable environment. In particular, any interpreters and libraries required by the target executable must be installed. Depending on the problem structure, it may also be necessary to upload data files to a known location on each used host. In the following examples, we will assume that the necessary files are uploaded by GNU Parallel for each job, and that the target executable requires a *venv* Python virtual environment[5] to run.

As it is necessary to prepare the environment on the remote host first, GNU Parallel should not invoke the target executable

[4]https://docs.python.org/3/library/multiprocessing.html
[5]https://docs.python.org/3/library/venv.html

```
#!/bin/bash
# usage: remote-job.sh ID SHELL_COMMAND
set -e
seq_id="$1"
target_exe="$2"

# unpack data files
tar xzf data.tar.gz

# initialize environment
source path/to/venv/bin/activate

# run command
RUN_GA='python runGA.py'
eval "$target_exe >logfile_${seq_id}.txt 2>&1"

# pack results
tar czf result-${seq_id}.tar.gz \
        result_*.csv logfile_*.txt
```

Listing 5: remote-job.sh

```
 1 #!/bin/bash
 2 
 3 tar czf data.tar.gz data/
 4 
 5 parallel \
 6   --sshloginfile hosts.txt \
 7   --workdir ... \
 8   --transferfile data.tar.gz \
 9   --transferfile remote-job.sh \
10   --transferfile runGA.py \
11   --return "result-{#}.tar.gz" \
12   --cleanup \
13   --eta \
14   --joblog .joblog \
15   ./remote-job.sh "{#}" "{}" <jobs.sh
16 
17 parallel tar xzf ::: result-*.tar.gz
18 rm data.tar.gz result-*.tar.gz
```

Listing 6: Typical session using GNU Parallel

directly. Instead, we create a small wrapper script `remote-job.sh` to manage these problems. It is given by GNU Parallel as command line arguments the GNU Parallel job ID (unrelated to the *Multijob* job ID), and the shell script snippet generated by *Multijob* which contains the EA parameters. The script will then unpack any transferred data archives, will initialize the required environment, will execute the target executable, and will pack the output files into a result archive that will be transferred back by GNU Parallel. An example is shown in Listing 5.

To drive the experiment, it is convenient to pack the data files into an archive, invoke GNU Parallel with the required options, and then unpack the result files. However, transferring all individual files would be possible as well.

First, GNU Parallel needs to know which hosts to log in to via SSH. These logins can be specified as arguments to the `-sshlogin` option, or can be listed line by line in a file that is given to the `-sshloginfile` option. It is necessary that GNU Parallel can log into these hosts without a password. In particular, this means that a public/private SSH key pair is generated, the public key uploaded to the remote host, and the private key added to the *ssh-agent* keyring session with *ssh-add*. It may also be necessary to increase the maximum number of simultaneous SSH connections to the host by editing the *sshd* configuration, as GNU Parallel will maintain one connection per available CPU on the remote host unless it is explicitly throttled.

On the remote host, the jobs will be executed in some working directory. A particular directory can be specified if required by the experiment, but generally a disposable temporary directory is preferred. This can be configured with the "`-workdir ...`" option, where the triple dot is a special value that requests this kind of temporary disposable working directory.

We can specify with `-transferfile` all files that should be transferred to the remote working directory from the controlling host. These are the target executable (here: `runGA.py`), the wrapper script `remote-job.sh`, and any data files `data.tar.gz`. If files should be returned from the remote host, these can be given to the `-return` option. Since we return files from many hosts, the returned file names should be distinct. This can be achieved by including the GNU Parallel job ID in the filename pattern with the `{#}` symbol. Finally, the transferred files can be deleted, which is done with the `-cleanup` flag.

When the `remote-job.sh` script is executed, it must receive the ID (`{#}` symbol) and the job shell command argument (`{}` symbol).

Unless explicitly limited, GNU Parallel will use as many hosts as available, and will launch as many jobs on each host as CPU cores are available on that host. Once the experiment has completed, the returned result archives can be unpacked and the results can be analyzed.

A typical session is shown in Listing 6. Since GNU Parallel is a general utility, it can also be used to unpack the returned TAR archives in parallel as shown in Line 17.

5 EXPERIMENTS

In order to study the effectiveness of the approach, several experiments have been conducted. A predefined evolutionary algorithm using different configurations has been executed on a computing server having the following specifications: 2x Intel Xeon CPU E5-2690 v4 @ 2.60GHz CPU, 28 total cores, 56 total threads, 126GB RAM. The experiments aim at simulating typical parameter tuning experiments, where one (or multiple concurrent) parameters are researched including an adequate amount of statistical significance through experiment repetition.

Performance is measured using real execution time of the experiments using a doubling amount of jobs, starting at 1, to be simultaneously executed on the server. As optimization heuristic, we picked the EA as proposed in [13]. However, as the evaluation of the experiments is problem-independent, any other configurable EA may have been used. This algorithm is used to optimize the number and positions of monitors in a given dynamic communication network topology model. The computational effort of this algorithm is low for the given problem instance and it requires several files to be transferred to/from the computing nodes, which is why we decided on this EA as test case. For comparison reasons, the same EA with a changing configuration is executed over all experiment instances. As problem instance for all configurations, the network model of the "National Research and Education Network (NREN) Europe" [11] is used, which consists of 1 157 nodes and 1 465 edges.

The termination condition of all EAs is the number of evaluations. The first two experiments are designed to terminate swiftly, using a maximum number of evaluations of 1 000. Each can be observed to terminate within several seconds. For the third experiment, the number of evaluations is increased to 10 000, which will result in a runtime per job of several minutes. This is done in order to differentiate between total runtime due to job runtime or total runtime due to communication overhead, i.e., file transfer, SSH connection handling, etc. In the following, the different configurations will be described.

The first EA configuration aims at simulating a check for statistical significance without performing any parameter tuning, thus, one EA configuration is repeated 100 times. The second configuration alters the crossover rate from 0.1 to 1.0, increasing it by a step size of 0.1. Again, in order to simulate checks for significance, each experiment in the configuration is repeated 100 times. The third configuration aims at tuning the same parameter as the second one, except for a longer job runtime due to the increased number of evaluations. For the fourth experiment, the same configuration as for experiment 3 is used. However, another computing node is added (2x Intel Xeon CPU E5-2650 v3 @ 2.30GHz CPU, 20 total cores, 40 total threads, 62GB RAM) and computation is done in a network-based computing cluster. Therefore, the maximum number of parallel jobs for this experiment is increased to 96, while the minimum number is increased to 2, i.e., one job per machine.

The numerical results of the experiments are shown in Table 1. Visualization of the results are shown in Figures 3 and 4. As can be observed in the obtained results, runtime of the jobs nearly halves in case the double amount of concurrent jobs is used. This indicates that speedup for the given experiment is close to linearity (the optimal case). Furthermore, the results imply that the transmission and job scheduling overhead are negligibly low, as indicated by the comparison of the job runtimes of the different experiments and the overhead a cluster configuration may have introduced. As shown, *Multijob* offers the expected linear job execution speedup for all of the given experiments.

Table 1: Experimental Results for Efficiency Study of the *Multijob* library

Experiment configuration	Concurrent jobs	Execution time
	1	473 sec
	2	231 sec
⋆ Repetitions: 100	4	114 sec
⋆ Number of evaluations: 1 000	8	58 sec
⋆ Total number of jobs: 100	16	32 sec
	32	19 sec
	56	15 sec
	1	4 888 sec
⋆ Crossover rate ($\{0.0, 0.1, \ldots, 1.0\}$)	2	2 399 sec
⋆ Repetitions: 100	4	1 178 sec
⋆ Number of evaluations: 1 000	8	583 sec
⋆ Total number of jobs: 1 100	16	293 sec
	32	151 sec
	56	125 sec
	1	20 991 sec
⋆ Crossover rate ($\{0.0, 0.1, \ldots, 1.0\}$)	2	10 454 sec
⋆ Repetitions: 100	4	5 228 sec
⋆ Number of evaluations: 10 000	8	2 610 sec
⋆ Total number of jobs: 1 100	16	1 319 sec
	32	764 sec
	56	763 sec
	2	12 371 sec
⋆ Crossover rate ($\{0.0, 0.1, \ldots, 1.0\}$)	4	6 190 sec
⋆ Repetitions: 100	8	3 128 sec
⋆ Number of evaluations: 10 000	16	1 625 sec
⋆ Total number of jobs: 1 100	32	833 sec
⋆ Computation distributed	64	542 sec
	96	536 sec

6 CONCLUSIONS AND OUTLOOK

In this paper, we introduced a framework called *Multijob*, which has a library component to create different algorithm configurations for parameter tuning and a workflow component to use GNU Parallel for distributed execution of the jobs. As the results of the experiments indicate, *Multijob* is able to conveniently create job configurations and efficiently distribute execution of them. In the studied cases, a linear speedup can be observed, where the efficiency doubled with a doubled number of simultaneously executed jobs. Programming language-agnosticism is one of the major benefits of the presented framework, thus, it may be helpful for increasing productivity for a wide range of researchers from distinct fields using different toolchains for research.

So far, the library part of *Multijob*, responsible for generation of different configurations, is implemented as a Python module, although it was intentionally designed to be language-agnostic. As future work, it is planned to implement backends in other languages, so that EAs can be written in different languages. Primarily, creating C++ and Golang backends seem to be promising due to their efficiency. As opposed to offering different backends for programming required parameters on a code level, creation of files describing the requirements on a configuration level is another option, e.g., XML-based. Furthermore, as creation of distinct configurations for the defined EA is neither time critical nor does it imply a noticeable impact on computing power in our experiments, the necessity for programming knowledge may be eliminated creating a graphical user interface as wrapper for *Multijob*s builder library.

The *Multijob* workflow reduces the administrative overhead for running distributed experiments on a cluster or grid, but does not eliminate it. Besides a SSH connection to the remote host(s), a valid environment for the execution of the experiments must be present. In the future, configuration management might be subject to improvement, which would eliminate major drawbacks of using *Multijob* in dynamically changing computing clusters of entering and leaving nodes. A possible fix for this could be using Pythons virtual environments (venv) in case a Python-based EA is executed. However, a more generic approach would be to use containers like Docker, which will be subject to further research in the future.

ACKNOWLEDGMENTS

This work was supported in the framework of Hessen Modell-Projekte, financed with funds of the European Union (European Regional Development Fund - ERDF) and the State of Hessen in the context of the research project "Reactive network Optimization

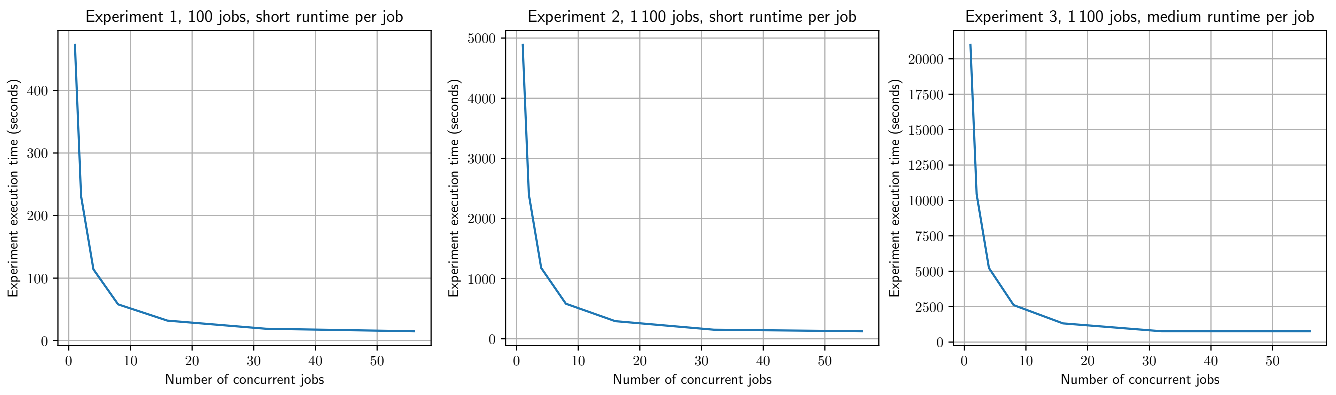


Figure 3: Experiment results indicating the job distribution efficiency of *Multijob* I

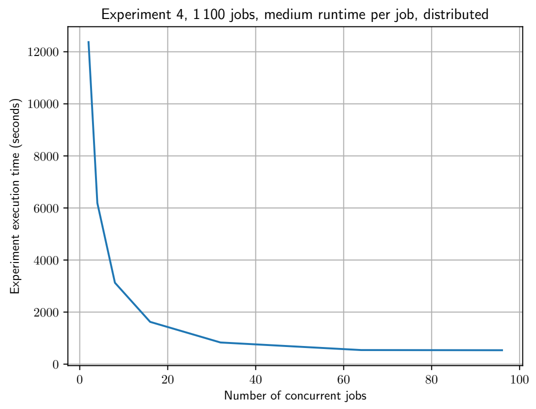


Figure 4: Experiment results indicating the job distribution efficiency of *Multijob* II

by Using SDN-Technology" (ROBUST) (HA project no. 473/15-15), and the Spanish Ministry of Economy and Competitiveness (National Program for Research, Development and Innovation) and with funds of the European Union (European Regional Development Fund - ERDF), project DArDOS TIN2015-65845-C3-3-R and Excellence Network SEBASENet TIN2015-71841-REDT. Responsible for the content are the authors.

REFERENCES

[1] Drona Pratap Chandu. 2014. A Parallel Genetic Algorithm for Generalized Vertex Cover Problem. *International Journal of Computer Science and Information Technologies (IJCSIT)* 5, 6 (2014), 7686–7689.

[2] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.

[3] Francois-Michel De Rainville, Felix-Antoine Fortin, Marc-Andre Gardner, Marc Parizeau, and Christian Gagne. 2012. DEAP - Enabling Nimbler Evolutions. *SIGEvolution newsletter of the ACM Special Interest Group on Genetic and Evolutionary Computation* 6, 2 (2012), 17–26. https://github.com/DEAP/notebooks

[4] Juan J. Durillo and Antonio J. Nebro. 2011. JMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software* 42, 10 (2011), 760–771. https://doi.org/10.1016/j.advengsoft.2011.05.014

[5] Achiya Elyasaf and Moshe Sipper. 2014. Software review: The HeuristicLab framework. *Genetic Programming and Evolvable Machines* 15, 2 (2014), 215–218. https://doi.org/10.1007/s10710-014-9214-4

[6] Filomena Ferrucci, M-Tahar Kechadi, Pasquale Salza, and Federica Sarro. 2013. A Framework for Genetic Algorithms Based on Hadoop. *CoRR, ArXiv e-prints* (2013). arXiv:1312.0086 http://arxiv.org/abs/1312.0086

[7] Felix-Antoine Fortin, Francois-Michel De Rainville, Marc-Andre Gardner, Marc Parizeau, and Christian Gagne. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (2012), 2171–2175.

[8] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H Castain, David J Daniel, Richard L Graham, and Timothy S Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 97–104.

[9] Yannick Hold-Geoffroy, Olivier Gagnon, and Marc Parizeau. 2014. Once you SCOOP, no need to fork. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. ACM, 60.

[10] Maarten Keijzer, J J Merelo, G Romero, and M Schoenauer. 2002. Evolving Objects: A General Purpose Evolutionary Computation Library. *Artificial Evolution* 2310 (2002), 829–888. http://www.lri.fr/{~}marc/EO/EO-EA01.ps.gz

[11] Simon Knight, Nickolas Falkner, Hung X. Nguyen, Paul Tune, and Matthew Roughan. 2012. I Can See for Miles: Re-Visualizing the Internet. *IEEE Network* 26, December (2012), 26–32. https://doi.org/10.1109/MNET.2012.6375890

[12] Juan-J. Merelo, Pedro Castillo, Pablo García-sánchez, Paloma de las Cuevas, and Mario García-Valdez. 2016. NodIO : A Framework and Architecture for Pool-based Evolutionary Computation. In *GECCO Comp '16: Proceedings of the 2016 conference companion on genetic and evolutionary computation companion*. 1323–1330.

[13] Robin Mueller-Bady, Martin Kappes, Inmaculada Medina-Bulo, and Francisco Palomo-Lozano. 2017. Optimization of Monitoring in Dynamic Communication Networks using a Hybrid Evolutionary Algorithm. In *GECCO '17: Proceedings of the 2017 conference on genetic and evolutionary computation*. (in press).

[14] Antonio J Nebro, Juan J Durillo, and Matthieu Vergne. 2015. Redesigning the jMetal Multi-Objective Optimization Framework. In *GECCO Comp '16: Proceedings of the 2016 conference companion on genetic and evolutionary computation companion*. 1093–1100. https://doi.org/10.1145/2739482.2768462 arXiv:arXiv:1508.06655v1

[15] Pasquale Salza, Filomena Ferrucci, and Federica Sarro. 2016. elephant56: Design and Implementation of a Parallel Genetic Algorithms Framework on Hadoop MapReduce. In *GECCO Comp '16: Proceedings of the 2016 conference companion on genetic and evolutionary computation companion*. 1315–1322. https://doi.org/10.1145/2908961.2931722

[16] O Tange. 2011. GNU Parallel - The Command-Line Power Tool. *The USENIX Magazine* 36, 1 (feb 2011), 42–47. https://doi.org/10.5281/zenodo.16303

[17] Stefan Wagner, Gabriel Kronberger, Andreas Beham, Michael Kommenda, Andreas Scheibenpflug, Erik Pitzer, Stefan Vonolfen, Monika Kofler, Stephan Winkler, Viktoria Dorfer, and Michael Affenzeller. 2014. *Architecture and Design of the HeuristicLab Optimization Environment.* Topics in Intelligent Engineering and Informatics, Vol. 6. Springer, 197–261. http://link.springer.com/chapter/10.1007/978-3-319-01436-4{_}10

[18] D H Wolpert and W G Macready. 1997. No Free Lunch Theorems for Optimization. *Trans. Evol. Comp* 1, 1 (apr 1997), 67–82. https://doi.org/10.1109/4235.585893