



Universiteit
Leiden
The Netherlands

Computer Science & Economics

Parallel algorithm portfolios in Sparkle

Richard Middelkoop

Supervisors:

Koen van der Blom & Holger Hoos

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)

www.liacs.leidenuniv.nl

03/08/2021

Abstract

A solver is an implementation of an algorithm for solving a given problem. This specific characteristic of a solver is what makes it useful in practical applications. Therefore, reducing the time a solver needs to solve a problem can be of substantial value to a wide array of real-world applications. Solver designers try to accomplish this, by improving upon the current state of the art of solvers. Currently, when solver designers want to create a new solver, they need to create a new environment to be able to test the new solver. Solver designers can also combine different software platforms to test the new solver; nevertheless, this can require pre- and post-processing between the different software. Since both cases are time-consuming it is of value to have a platform that supports solver designers in their development process. A platform that helps solver designers is Sparkle. The platform Sparkle is designed for the evaluation of empirical algorithms/solvers and can support solver developers in the evaluation and testing process of developing a solver. This thesis will introduce parallel portfolios to Sparkle. Parallel portfolios are a set of independent, non-communicating solvers which solve problem instances in parallel. By introducing parallel portfolios to Sparkle, this thesis will show how parallel portfolios, support solver developers in their development process and show the potency of parallel algorithm portfolios. How this thesis will do this is by providing insight into the design and implementation of parallel algorithm portfolios into Sparkle. Furthermore, it will use experiments to show the practical capabilities of parallel portfolios. The results of the experiments show that parallel portfolios can be easily utilised in Sparkle, achieving a significant saving in wall-clock time. However, the conducted experiments quantified the overhead which occurred during the experiments these include; a delay when placing the jobs on the cluster, and an increase in solver overhead when scaling the portfolio above 64 solvers which are 1/8th of the cluster size. Concluding, the paper will discuss further applications of parallel algorithm portfolios and possible improvements, or enhancement of the design.

Contents

1	Introduction	1
2	Theoretical Framework	2
2.1	Related work	2
2.2	Main concepts	3
2.2.1	Parallel algorithm portfolio	3
2.2.2	Two types of algorithm: Optimisation and decision	3
2.2.3	Randomised solvers	4
2.2.4	How to evaluate a result	4
2.2.5	Status codes	4
3	Design	5
3.1	Design choices	5
3.2	User guide	5
3.3	System overview	7
4	Implementation	9
4.1	Command-line interface	9
4.2	Data flow	11
4.3	User configuration	11
5	Experiments	12
5.1	Portfolio vs. single solver	13
5.2	Scaling experiment	15
6	Discussion	20
7	References	23
	Appendices	24
A	Listings of the User guide	24
B	Results of experiment 5.1	27
C	Results of experiment 5.2	29
D	Grace specification	30
D.1	Hardware	30
D.2	Software	30
E	Example sparkle report	31

1 Introduction

Nowadays, the involvement of algorithms can be seen almost everywhere. Therefore, the speed at which algorithms perform their task is crucial since it can influence everyday tasks. For instance, think about your route planner which uses algorithms to calculate a new route after you have taken the wrong exit or the finding of an alternative route due to traffic. Solver developers develop algorithms to improve the performance of the current state of the art of solvers. Currently, when solver designers want to design a new solver, they have little support for managing administrative and trivial tasks within the designing process. This can take up a lot of valuable time. To try and combat this, solver designers combine different software platforms that support them in different "uninteresting" tasks. However, this can cause a lot of pre- and post-processing between the software platforms. Since both of these cases are time-consuming, it would be of value if there was a platform that would support them through the complete process. There is a platform, currently under development, which tries to achieve this, called Sparkle. In 2015, a technical report about *Sparkle* [1] was published by Hoos and in 2019 an additional functionality, algorithm configuration, was implemented by van der Blom et al. [2]. This report outlines the functionalities, as well as possible extensions of the Sparkle platform.

Within this thesis one such extension, parallel algorithm portfolios [3], will be introduced which can serve as an important tool for solver developers, it will support the users of the extension with construction, execution and empirical evaluation of the portfolio. Considering that the state of the art does not consist of a single best algorithm, instead, it is a set of complementary algorithms. By combining the algorithms they create the state of the art. The reason for this is that solvers perform differently on different problems; therefore, one solver might outperform one solver on one instance but gets outperformed on the next. For solver developers a parallel algorithm portfolio would enable them to test a set of algorithms in parallel, this could, in turn, be used to see if their portfolio performs, as well as they would hope. Furthermore, for solver users, the extension would enable them to use their parallel computing resources to improve the performance of their experiments. Additionally, the simple design of Sparkle allows for easy comprehension for novice solver users, this enables more people to solve problems with solvers. In conclusion, the Sparkle platform intends to support solver developers, and incentivise them to focus their energy on their work of improving the state of the art [1]. Sparkle tries to achieve this with multiple readily available tools supporting solver developers, and if Sparkle gains a tool that allows for the constructing, testing, and evaluation of a set of solvers on instances, it would support solver developers.

In short, this thesis will do exactly this, it will showcase the design and implementation of parallel algorithm portfolios into Sparkle. Furthermore, it will perform several experiments showing the potency of parallel algorithms portfolios and the performance of the implementation. Parallel algorithm portfolios use parallel computing resources to improve the wallclock execution time. This is done by running multiple jobs in parallel. Additionally, this thesis will discuss further improvements of the implementation and additional options which can further support solver designers. The structure of the thesis is as follows, Chapter 2 explains the most relevant works, as well as the relevant technical terms used in the design of the system. Chapter 3 shows the overall design of the system and a user guide showcasing the main use cases. This will be done by listing the steps a user needs to take to complete the use case in Sparkle commands. In Chapter 4 a further insight into the design of Sparkle is given by listing all Sparkle commands and their settings. Furthermore, parallel algorithm portfolios introduce several global variables which are also shown

in the chapter. Chapter 5 demonstrates the practical capabilities of parallel algorithm portfolios by performing several insightful experiments on Leiden’s Grace cluster. In Chapter 6 the work is summarised, and future work is discussed on the implementation and parallel portfolios as a whole.

2 Theoretical Framework

In this chapter the theoretical background required to understand this thesis will be discussed. This consists of the most relevant papers in Section 2.1 as well as the relevant technical terms in Section 2.2.

2.1 Related work

In 1997 as the first ones, Huberman et al. provided a common method to combine existing algorithms. These algorithms were in no form superior over one another, instead, they rivalled each other. By combining them they performed better than any of the components separately [4]. This method revolves around heuristic algorithms which were developed to combat extremely hard computational problems. These algorithms had their flaws, and performed significantly different from one problem instance to another, even rerunning an algorithm on a problem instance could result in different outcomes. Ultimately, the method revolved around constructing portfolios that combine different algorithms. *“The portfolio is constructed simply by letting both algorithms run concurrently but independently on a serial computer”* [4].

This method can now be described, as ‘algorithm portfolio’ when looking at the paper written by Gomes and Selman [3]. This paper was written four years later and concerns the study in which a portfolio approach compared to a single algorithm approach, provides a strong improvement regarding overall performance. Furthermore, Gomes and Selman state that *“a good strategy for designing a portfolio is to combine many short runs of the same algorithm”* [3]. The same algorithm is only varied by changing the initial seed of the algorithm, which is used by the random number generator. The authors mention that this method, along with several other methods, use an algorithm portfolio to improve the overall performance.

Gomes and Selman discuss the difference between the theory and practice in their paper *Algorithm portfolio design: Theory vs. practice* [5]. In the paper, the authors create a portfolio to demonstrate the conditions needed for a portfolio approach to have a substantial performance advantage over more established methods. Within a portfolio design, they consider a setup consisting of separate, non-communicating runs. From the concrete empirical results, the conclusion has been drawn that the portfolio approach outperforms the best-established methods in *“hard combinatorial search and reasoning problems”* [5].

Although there are no publicly accessible frameworks, like Sparkle, to relate to there have been multiple usages of algorithm portfolios in experiments [6, 7, 8, 9]. For example, in the paper from Lindauer et al. [7] several frameworks and methods are combined to perform their experiments. Within this thesis, we will implement this in a publicly accessible framework, which is called Sparkle [1]. Sparkle is publicly available as of 02-07-2021 on [bitbucket](#) and is being continuously developed. To use the framework, one must simply follow the installation instructions. Thereafter, the platform can be used by adding the desired instances and solvers and by selecting the desired procedure. This procedure sends jobs to any computing environment, which uses the Slurm workload

manager. Lastly, the workload manager distributes these jobs for execution. One of the goals of Sparkle is that the jobs can also be managed through other environments, which do not use Slurm. Summarising, in this work we will extend Sparkle with a new functionality, parallel algorithm portfolios.

2.2 Main concepts

The remainder of the chapter will explain the core concepts used in the design and implementation of the parallel portfolio.

2.2.1 Parallel algorithm portfolio

Gomes and Selman state that “*A portfolio of algorithms is a collection of different algorithms and/or different copies of the same algorithm running on different processors*” [3]

Gomes and Selman introduce three different types of Algorithms Portfolios; “(1) *running on a parallel machine, (2) running interleaved on a single processor, and (3) running an algorithm “restart” strategy*” [3]. The focus of this project is the type that uses a parallel machine. This means that multiple algorithms are carried out simultaneously.

In the context of this project, this can further be specified, as the use of several CPU cores. Each core containing different algorithms or sometimes the same algorithm but with different pseudo-random seeds. These algorithms try to solve a problem instance simultaneously in the same time span.

2.2.2 Two types of algorithm: Optimisation and decision

Sparkle differentiates between two types of algorithm: Optimisation and decision. The performance of these two types are measured in different ways and require a different setup in Sparkle.

Optimisation algorithms are algorithms that solve optimisation problems. The solution to the problem is the best answer to the input. Considering this, each solver within a portfolio is required to return their solution after a given time span. Afterwards, the best solution is chosen. An example of an optimisation algorithm is the shortest path algorithm, the algorithm tries to find the shortest route between point A and point B of a given problem.

Decision algorithms are algorithms that solve decision problems. The solution of the problem can only be one of two outcomes; yes, or no. This holds true for all decision algorithms; however, the literal answer of an algorithm can vary.

A specific type of decision algorithm is an SAT solver, these solvers can solve satisfiability problems. A satisfiability problem is a problem that contains a list of requirements that need to be met in order for that item in the list to be satisfied. If there is a way in which all items on the list can be satisfied then the complete problem is satisfiable. For example, look at a list of sports people want to have covered on the network channel of the Olympic games. If there is a selection of sports which a network channel can air which satisfies everyone, the satisfiability problem would be found satisfiable by an SAT solver. The decision algorithms in Chapter 5 is an SAT solver and considers SAT or UNSAT. These answers are status codes; see Section 2.2.5 for a list of status codes used in Sparkle.

The two types of algorithms have different setups because the solvers are stopped at different moments in time. For decision algorithms, this moment is when one solver within a portfolio finds

an answer to the problem instance it is running on. At that point, the answer is found and there is no reason to allow the other solvers to find the same answer on their own, thus, they are cancelled at that point. For optimisation algorithms, the moment a solver should be stopped is when it reaches the cutoff-time which is specified for the experiment. This can also occur for decision algorithms and that is if no solvers within the portfolio find an answer to the problem within the cut-off time.

2.2.3 Randomised solvers

Solvers can have randomised behaviour; this is created by using a seed. Solvers in Sparkle have a wrapper that is used internally in Sparkle to instruct and communicate with the solver of the user. One of these instructions is a seed number, the seed takes input in the form of an integer and is used as input for a pseudo-random number generator. By varying the seed number, different variations of the solver can be created. Consequently, when running a solver with a randomised seed number, random behaviour can be mimicked. Using multiple copies of the solver can result in better performance, this is because the chance of getting “lucky” is increased when there are more copies of a solver making decisions based on their seed number. In the case of decision algorithms, this means that the running time will be reduced when running multiple copies in parallel, in the case of optimisation algorithms this will result in a more optimised solution.

2.2.4 How to evaluate a result

For decision algorithms when a solver is finished a status code is returned, as well as a PARx score. The status code gives insight into the outcome of the process. For example, the status code `TIMEOUT` is returned when a solver fails to solve an instance within the allotted time, this is called the `cutoff-time`. The PARx score gives insight into the duration of the process, when a solver finishes before the cutoff-time, which can either be in wallclock- or CPU time, the finishing time is the score. If, however, the solver is not able to finish before the cutoff-time the score is the cutoff-time multiplied by the x in PARx. In this way, the PAR10 score of a solver, which does not finish before the cutoff-time, will be scored the cutoff-time times ten.

The PARx score is then used to find the single best solver. The single best solver is the solver which has the lowest PARx score over a set of problem instances. This can be derived from the table of solvers which show their PARx score on each instance and the solver with the lowest score is the single best solver (SBS).

For optimisation algorithms when a solver is finished it also returns a status code and a performance value. On the contrary, to the decision algorithm, this value is an answer for the problem instance. When processing the results of an experiment the best (lowest) value returned by the solvers on an instance is chosen.

2.2.5 Status codes

Within the context of this thesis, status codes are variables returned by the `run_solver` program and they give insight into the status of the job executed by the `run_solver`. These codes are single words or abbreviations of a word/words, the three most frequent status codes are explained below. `SAT`: This is a code used for a decision algorithm, more specifically an SAT solver, and is short for satisfiable. The meaning of the code is that problems instance on which the algorithms is executed has found a solution which is positive.

UNSAT: Similar to SAT, UNSAT is a code used for SAT solvers, and is short for unsatisfiable. The meaning of the code is that the problem instance on which the algorithm is executed has finished and a solution does not exist (according to this solver).

TIMEOUT: This is a code also used for decision algorithms and means that the solver was unable to find an answer before the `cutoff-time` is reached.

3 Design

As mentioned in Section 2.2.1 in its foundation the design of the parallel portfolio extension, in Sparkle, allows for several solvers to be run on a parallel machine on a singular instance. However, Sparkle does not limit itself to only run a script for such a machine, it also handles the pre-processing and post-processing of a portfolio. Therefore, in the design for Sparkle, the code is separated into three segments: the construction of the portfolio, the running of the portfolio, and the generating of the report.

3.1 Design choices

For the first segment of the design, the construction of a portfolio, a choice has been made to only include the solvers within the construction, and leave the specification of which instances that are going to be used to the second segment of the process. So, for the construction of the portfolio a user only has to submit the solver(s) with the `add_solver` command and the portfolio name with `construct_sparkle_parallel_portfolio --portfolio-name my_portfolio` to complete the first step.

For the running of the portfolio, the type of solvers within the given portfolio have to be specified with the `run_sparkle_parallel_portfolio --performance-measure my_performance_type` command in order for the portfolio to be handled in the desired manner. Furthermore, the cutoff-time can be specified to override the default cutoff-time value. This can be done by adding the option `--cutoff-time my_cutoff_time` to the previous command. The default value is stored in a settings file and can also be changed within that file.

The third section needs to be called with the `generate_report` command when the running of the solvers has been completed and will require no additional input to generate a pdf file containing the results of the conducted experiment.

3.2 User guide

The two main use cases within Sparkle for parallel portfolios are portfolios containing decision algorithms and portfolios containing optimisation algorithms. In Appendix A the complete examples are shown of a step-by-step guide to be able to execute all necessary commands from start to finish. In this section, we will show a selection of these commands and explain their function in more detail. The first command `add_instances` is used, but has not been modified, the commands `add_solver` and `generate_report` are existing commands but have received modifications. The other two commands `construct_sparkle_parallel_portfolio` and `run_sparkle_parallel_portfolio` are newly implemented.

1. <code>Commands/add_instances.py</code>

The first command is used to add instances to the folder from where the `run_sparkle_parallel_portfolio` can locate them, this is the `Instances/` folder. Note that the command must be followed by the main folder containing single instances and/or sub folders containing a single instance. For example, `Commands/add_instances.py path/to/instance_folder/`.

```
2. Commands/add_solver.py
```

The second command is used to add a solver to the Sparkle platform and can be combined with the option `--solver-variations` to add multiple variations to the platform. This will be done by denoting the number of variations in the solver list (located in `Reference.Lists/Sparkle_solver_list.txt`).

```
3. Commands/construct_sparkle_parallel_portfolio.py
```

The third command is used to construct the portfolio by combining solvers into a portfolio. There are several ways in which these solvers can be selected. The first method is without using the `--solver` option, then ALL the solvers in the solver list will be used. The other methods involve the use of the `--solver` option. The option has to be followed by a space-separated list of solver paths. It is possible to add multiple variations of a solver to a portfolio, if the solver uses a seed number within their solving process, this number can be used to alter the behaviour of the solver. If, no solver variations are specified, then the number of variations will be selected from the solver file. To override this, a solver path has to be followed by a comma and the desired number of solver variations. All constructed portfolios in Sparkle must have a unique name and can be named with `--nickname` option.

```
4. Commands/run_sparkle_parallel_portfolio.py
```

The fourth command is used to run the portfolio, the command requires three different options: `--instances`, `--portfolio-name` and `--performance-measure`. The option `--instances` can be used to specify the instances on which the portfolio will run. If, however, the option is not used then all instances used in the `Instance/` folder will be used. The second option can be used to specify the portfolio which will be used. If the option is not used than the latest constructed portfolio will be chosen. The third option is used to specify which type of solvers are contained within the portfolio, this can either be `RUNTIME` for decision algorithms or `QUALITY_ABSOLUTE` for optimisation algorithms.

```
5. Commands/generate_report.py
```

The fifth and final command is used to generate a report. This will generate a report of the latest experiment which was executed. No further options have to be specified. However, do note that generating a new report overrides the previously generated report, so be sure to save the previously generated report into another location before generating a new report. Within the report one can find two chapters, the first chapter contains a short introduction. The second chapter lists the used solvers and instances, as well as the experimental setup. Lastly, it contains an empirical evaluation of the results of the experiment including a graph of the single best solver versus the portfolio itself, in Appendix E an example report is shown.

These commands can be used for decision algorithms, as well as optimisation algorithms. The only option which will have to be differentiated is the `--performance-measure` option for the run sparkle command (command 4). Besides this, the selected instances and solvers will have to be varied as well.

An example experiment that uses all five commands explained in this section is shown below.

```

1. Commands/add_instances.py --run-solver-later --run-extractor-later
Examples/Resources/Instances/PTN/
2. Commands/add_solver.py --run-solver-later --deterministic 0
Examples/Resources/Solvers/CSCCSat/
2. Commands/add_solver.py --run-solver-later --deterministic 0
Examples/Resources/Solvers/MiniSAT/
2. Commands/add_solver.py --run-solver-later --deterministic 0
Examples/Resources/Solvers/Pb0-CCSAT-Generic/
3. Commands/construct_sparkle_parallel_portfolio.py --nickname
user_guide_example --solver Solvers/Pb0-CCSAT-Generic Solvers/MiniSAT/
4. Commands/run_sparkle_parallel_portfolio.py --instance-paths Instances/PTN/
--portfolio-name user_guide_example
5. Commands/generate_report.py

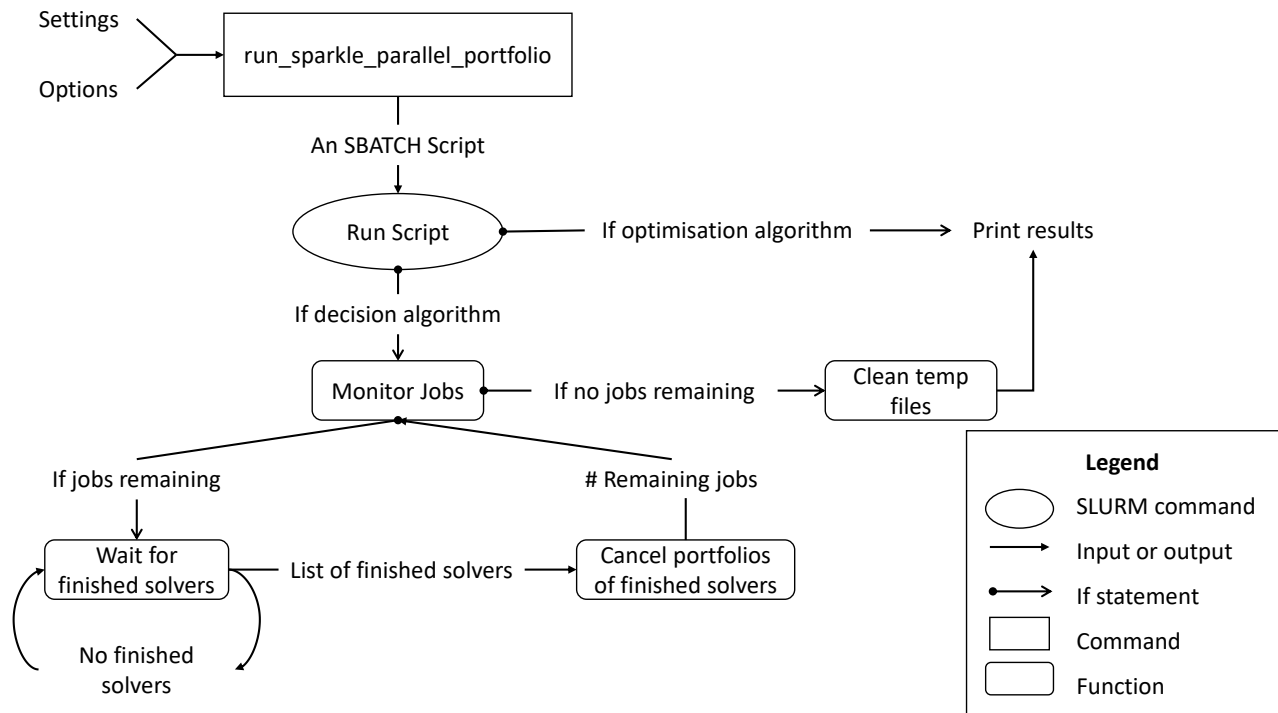
```

3.3 System overview

In this section, the system is shown in a visual summary. In the figure below, Figure 1, the three main segments are shown accompanied by the general purpose of the section and the necessary steps to execute that section.

Figure 1: A visual overview of the three main segments of parallel algorithm portfolios within Sparkle



Figure 2: The overview of the processes within `run_sparkle_parallel_portfolio`

In Figure 2 the process behind the execution section of the system overview is portrayed. The process starts at the top left with the user sending the Sparkle command, possibly with options. First, the `run_sparkle_parallel_portfolio` command gathers the settings and options, using the default values if some are not specified. This information is sent to the helper file of the command, and within this file, the rest of the process is managed.

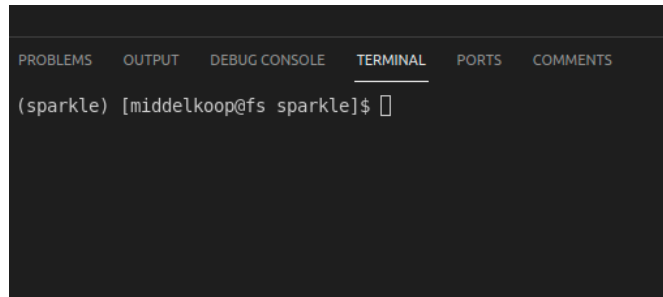
The helper file firstly constructs an `SBATCH` script using the given information from the primary file. An `SBATCH` script is a small script used for sending commands through `SLURM` to the Grace cluster and contains all the information `SLURM` needs, to send all the jobs to the cluster. Secondly, the helper file runs the generated script by sending a `SLURM` command to the terminal. In the case of optimisation algorithms this is the last step of the process and after the process finishes running the results are printed. However, for decision algorithms monitoring is required. The reason for this is once a solver has made a decision on an instance the other solvers have to be cancelled. This is done by three functions that operate in a loop. The monitor jobs function checks if the loop has to continue with another cycle or if all jobs are finished, and the process can end. Within the loop, the first function continuously checks if a solver is finished, and when this occurs the list of finished solvers is passed to the second function of the loop. The second function checks if all the other solvers, which are run on the same instance of the finished solver, have had at least the same amount of running time. If this is true, the solver is cancelled. Otherwise, the function calculates the time the solver still has and sends a delayed cancel command which executes after this time has passed.

When the monitor jobs function finds that no jobs are remaining, the final function of the process clears all temporary files, ends the process, and the results are printed.

Note that the decision algorithm section of Figure 2 is explained with the `--process-monitoring`

set to `EXTENDED`. The reason for this is that this option was added only after the experiments were conducted. The current default setting of the `run_sparkle_parallel_portfolio` option `--process-monitoring` is `REALISTIC` and will cancel solver within a portfolio containing a finished solver regardless of whether that solver has gotten to run at least the same amount of time as the finished solver. Additionally, the `EXTENDED` option has the most monitoring processes, which would also result in the largest increase in possessing overhead. Therefore, the results from the experiments show a larger increase in overhead than when a user would only be interested in the outcome of the experiment, this is because for this scenario the `REALISTIC` option would suffice.

Terminal: Sparkle uses a command-line interface, meaning that to use Sparkle you will have to use a terminal. Depending on the software you use to run Sparkle, the terminals look and lay-out can vary but its functionality will remain the same.



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS
(sparkle) [middelkoop@fs sparkle]$ █

```

The terminal will always have a single input line on which you can enter a command. These commands will dictate Sparkle what actions to perform. However, be precise, because if you mistype the command, it will not execute. If there is a command which is unclear add the `--help` option behind the command and it will show additional insight into the command, see Section 4.1 for more details.

4 Implementation

The implementation of the design is divided over six different python files, including three command files, which can be called by the users, and three accompanying helper files to process the input given by the command files. The aim of the implementation was to be as integrated into Sparkle as possible and to use readily available code within Sparkle with only slight changes when needed. Therefore, the `generate_report` command uses the general generate report python file used by multiple processes and only an additional helper file was made to facilitate the parallel portfolio report.

4.1 Command-line interface

All three sections of the design process have their command and are accompanied by several options, which can be used to further specify the command. Most options only have to be used when the user wants to use non-default values. The default values are stored in a settings file and can be changed to alter the default values. Additionally, the standard is that the latest process will be selected in case of the report generation. For the construction or running of the portfolio, all added instances and solvers within Sparkle will be selected.

All commands for Sparkle can be executed within a command-line interface, this means that Sparkle functions can be called by using text input. These text inputs and their options are highlighted below.

`Construct_sparkle_parallel_portfolio` is the command for the construction of a portfolio. The construction process will use either the default values or the given options to determine what solvers need to be added to the portfolio. These solvers will be listed in a text file within a folder holding the portfolio name.

```
Commands/construct_sparkle_parallel_portfolio.py
--help: shows a help message explaining the command and its options.
the option also exits the command.
--nickname: Give a nickname to the portfolio, the default name of
a portfolio is sparkle_parallel_portfolio.
--solver: Specify the list of solvers, add ",solver_variations"
to the end of a path to add multiple instances of a single solver.
For example --solver Solver/Pb0-CCSAT-Generic,25 to construct a
portfolio containing 25 variations of Pb0-CCSAT-Generic.
--overwrite: Allows overwriting of the directory, default true if
the --nickname option is NOT specified otherwise constructing a
portfolio with a name of an already existing portfolio will throw
an error if --overwrite True is not used.
--settings-file: specify the settings file to use in case you want to use
one other than the settings file
```

`Run_sparkle_parallel_portfolio` is the command for the executing and monitoring of the portfolio. The running process will use either the default values or the given options to determine what problem instances need to be run on which portfolio, and how the performance is measured. During the process, several files will be created which `generate_report` will use to show the results and performance of the portfolio.

```
Commands/run_sparkle_parallel_portfolio.py
--help: shows a help message explaining the command and its option.
the option also exits the command.
--instance-paths: Specify the instance_path(s) on which the portfolio
will run. This can be a space separated list of instances contain instance
sets and/or singular instances. For example --instance-paths
Instances/PTN/Ptn-7824-b01.cnf Instances/PTN2/
--portfolio-name: Specify the name of the portfolio, if the portfolio is
not in the standard location use its full path, the standard location is
Sparkle_Parallel_Portfolio/. If the option is not used the latest
constructed portfolio will be used.
--process-monitoring: Specify whether the monitoring of the
portfolio should cancel all solvers within a portfolio once a
solver finishes(realistic), or allow all solvers within a
portfolio to get an equal chance to have the lowest amount of
running time on an instance(extended). The default option is
realistic.
--performance-measure: The performance measure, e.g. runtime (for
decision algorithms) or quality_absolute (for optimisation
algorithms)
--cutoff-time: The duration the portfolio will run before the
solvers within the portfolio will be stopped.
```

`Generate_report` is the command for the generation of a pdf report containing the results of the latest parallel portfolio experiment.

```
Commands/generate_report.py
  --help: shows a help message explaining the command and its option. the
  option also exits the command.
  --settings-file: specify the settings file to use in case you want to use
  one other than the settings file
```

4.2 Data flow

This section will explain how and when data is generated and stored during the process. During the construction of the portfolio, a sub-directory within the `Sparkle_Parallel_Portfolio` directory is created. The sub-directory contains a text file containing a list of paths to the solvers within the portfolio. If there are multiple solver variations of a solver within the portfolio, a number is added following the solver path separated by a space. As mentioned in Section 3.2 and Section 2.2.3 this number will be used to select several seed numbers, for five solver variations the first solver will have the seed number 1, the second seed number 2 etc. In this way, the variations are controlled random, every time the portfolio is run the solvers will behave the same.

During the running of the portfolio an `SBATCH` script is generated containing all jobs that need to be executed. In the case of an optimisation algorithm, the script can be submitted, and no monitoring is required. However, in the case of decision algorithms, the jobs need to be monitored. For the reason that once a solver finds a solution to an instance all other solvers on that instance have to be cancelled. This is accomplished by monitoring the number of jobs that are running, if this number reduces this indicates that there is a job that has finished. When this occurs, a result file is created and added to the `Performance_Data/Tmp_PaP` directory. The finishing time within this file is then used to cancel the jobs, which are running on the same problem instance. After all processes have finished running the jobs that have been cancelled, temporary processing files that have not been removed are now removed. After the running of the portfolio, the only files that remain are the results files that `generate_report` will use.

To generate the report, multiple files are created. Most importantly, one file containing the bibliography, and a pdf file in which several generated files are combined, including the generated graphs and generated text for the report. These files are located within the `Components/Sparkle-latex-generator-for-parallel-portfolio` directory. After completing all steps, as shown in Section 3.2, the files that remain are the constructed portfolio, the results files, and the report.

Furthermore, there are several log files in which log statements are kept with regards to the process, these will be placed within the `Output/` folder where the output of all experiments are located. Only the logs of the results are kept during the execution phase and can be found in the temp folder. It is important to be aware that these log statements are generated by the solvers provided and that these can be difficult to understand.

4.3 User configuration

The settings folder in Sparkle contains a file, which can be used to configure several default values and is called `sparkle_settings.ini` and the headers `[general]`, `[parallelportfolio]`

and `[slurm]` are of interest for parallel portfolios. The relevant variables will be listed below accompanied by an explanation of their meaning and options.

1. `performance_measure`, the default is `RUNTIME`, but other options are `QUALITY` or `QUALITY_ABSOLUTE`. `RUNTIME` will indicate decision algorithm behaviour and `QUALITY` or `QUALITY_ABSOLUTE` will indicate optimisation algorithm behaviour.
2. `target_cutoff_time`, the default is 300, this is the cutoff-time in seconds and can be any positive integer.
3. `penalty_multiplier`, the default is 10, and this is the multiplier for the PARx score. The variable can hold any positive integer.
4. `overwriting`, the default is `True`, the variable indicates that when a portfolio is constructed with the same name, as a portfolio that already exists, it will be automatically overwritten or that an error message should be raised. The variable must either hold `True` or `False`.
5. `process_monitoring`, the default is `REALISTIC`, this defines the way in which decision algorithm will be monitored when run. `REALISTIC` will mean that once a solver within a portfolio finishes, all the other solvers within the portfolio will be cancelled regardless of the running time of the other portfolio. The other option is `EXTENDED` in contrast with `REALISTIC`, the option will check if the other solvers have gotten at least the same amount of running time as the solver that finished.
6. `number_of_runs_in_parallel`, the default is 250, and the variable is the total maximum number of runs that an SBATCH script will run in parallel.
7. `clis_per_node`, the default is 32, and this number is the maximum number of concurrent runs a node can handle.

A user can configure these values either by changing these within the files or manually creating a different settings file containing the variables which need to be adapted. This file would then need to be referenced when executing a Sparkle command using the `--settings-file` option.

5 Experiments

In this chapter, different uses of the parallel portfolio will be demonstrated. Additionally, the performance of these uses will be measured by comparing the performance of a parallel portfolio and its comparison target. In experiment 5.1 a comparison of overhead is made; this will be done by comparing the overhead of a singular solver and a portfolio containing a singular solver. In experiment 5.2 the performance of the portfolios when scaled in size will be measured. This is done by comparing the running time of the portfolio to the solving duration. Additionally, the difference between the running time of the portfolio and the solver duration will be outlined by dividing it into two types of overhead: job overhead, and solver overhead. All experiments will use problems selected from the crafted16 benchmark set of the 2016 SAT Competition [10]. The choice of the instances is based on test results using five solver variations of PbO-CCSAT-Generic and selecting instances that performed between one second and fifty minutes. The reasoning for this is that

PbO-CCSAT-Generic [11] is part of the base installation of Sparkle. The solver can be found in the `Examples/` folder, making it easier to reproduce the experiments. The chosen instances can also be found in the `examples` folder, specifically, `Examples/Resources/Instances/` and are a selection from instances in `PTN/` and `PTN2/`. The selection criteria was used for the instance being solved in a time between one second and fifty minutes. The criteria was based on time constraints and interpretability. Since the Grace server only creates log statements of completed jobs rounded up to seconds, the results of an instance, which is solved in less than one second, would not show any improvement in solving time during the experiments. The cap of fifty minutes was chosen since the scaling experiment uses the entire grace cluster. This hinders other users of the platform, therefore, the cap is used to limit this. All selected instances are from Heule [12] and are named `Ptn-7824-b**.cnf`, with the two asterisks being `{01,03,04,05,06,07,09,11,13,15,18,20}`. The base data of the grace clusters' hard-/software specification can be found in Appendix D and are duplicated from the latest available version of the Grace user guide [13].

5.1 Portfolio vs. single solver

To illustrate that the running of the portfolio is not significantly slower than running a solver on an instance without a portfolio, a comparison will be made between a portfolio containing a single solver and a singular solver executed by the `run_solver.py --parallel` command. Both variations will use the instances mentioned in the paragraph above and will use the PbO-CCSAT solver. Additionally, the variations will be running on a single node in parallel with a `cutoff-time` of 3000 seconds. The performance of the variations will be measured by the solver duration and total duration. The solver duration is the *pure* duration containing only the running time of the solver. The total duration is the time measured from the moment a job is placed on the grace cluster until the job vacates its place. The difference between the two types of running time is the overhead of a job. The overhead contains several different delays; the processing time of grace, and the processing time of `run_solver`. In the overhead of the portfolio, a third delay is present, the delay of the processing time of the portfolio. To maintain some form of statistical consistency both variations are run three times. Using the results the average overhead time will be computed, and a comparison will be made between the experiment with and without a portfolio.

Table 1: Single solver experiment without portfolio

The shown time is in wallclock time

Instance	Total duration	Solving duration	Overhead
Ptn-7824-b20	3 seconds	1.32225 seconds	1.67775 seconds
Ptn-7824-b18	4 seconds	2.59785 seconds	1.40215 seconds
Ptn-7824-b13	23 seconds	22.1711 seconds	0.8289 seconds
Ptn-7824-b09	41 seconds	39.6904 seconds	1.3096 seconds
Ptn-7824-b11	67 seconds	65.5938 seconds	1.4062 seconds
Ptn-7824-b15	197 seconds	195.302 seconds	1.698 seconds
Ptn-7824-b06	1465 seconds	1463.6 seconds	1.4 seconds
Ptn-7824-b01	2482 seconds	2480.44 seconds	1.56 seconds
Ptn-7824-b07	3001 seconds	3000.07 seconds	0.93 seconds
Ptn-7824-b05	3003 seconds	3000.1 seconds	2.9 seconds
Ptn-7824-b04	3003 seconds	3000.09 seconds	2.91 seconds
Ptn-7824-b03	3007 seconds	3000.08 seconds	6.92 seconds

Table 2: Single solver experiment with portfolio

The shown time is in wallclock time

Instance	Total duration	Solving duration	Overhead
Ptn-7824-b20	3 seconds	2.41766 seconds	0.58234 seconds
Ptn-7824-b18	4 seconds	2.60105 seconds	1.39895 seconds
Ptn-7824-b13	23 seconds	22.3893 seconds	0.6107 seconds
Ptn-7824-b09	41 seconds	39.927 seconds	1.073 seconds
Ptn-7824-b11	67 seconds	65.9 seconds	1.1 seconds
Ptn-7824-b15	196 seconds	193.153 seconds	2.847 seconds
Ptn-7824-b06	1466 seconds	1465.71 seconds	0.29 seconds
Ptn-7824-b01	2477 seconds	2476.68 seconds	0.32 seconds
Ptn-7824-b07	3001 seconds	3000 seconds	1 seconds
Ptn-7824-b05	3002 seconds	3000 seconds	2 seconds
Ptn-7824-b04	3003 seconds	3000 seconds	3 seconds
Ptn-7824-b03	3005 seconds	3000 seconds	5 seconds

In Table 1 and Table 2 the results of the experiment are shown. For statistical consistency both experiments have been run two additional times, the results of these experiments can be found in Appendix B. The average overhead, which is computed by dividing the cumulative overhead by the number of instances, of the three non-portfolio experiments are 2.07855, 1.437201667 and 1.57429. In comparison, the outcome of the portfolio experiments, which are 1.6018325, 6.872275833 and 1.843208333, have a slightly higher overhead with an average increase of 0.2 seconds per instance between the two lower outcomes of both experiments. The exception is the highest outcome of the experiment, a possible explanation for this can be that at the time of running the experiment on

the grace cluster, the cluster was exceptionally busy which could affect the overhead. Note that both `run_solver` and `run_sparkle_parallel_portfolio` use the same seed number, which is seed one.

The second portfolio experiment, as shown in Table 8, has the highest average overhead. Note that the overhead remains about the same between different lengths of solving duration. To quantify the “business” during the experiment with high overhead about sixteen nodes were in use, which is almost 50% of the available nodes of the cluster and only our experiment was running on its node.

Command used for the experiment:

Run solver:

```
./Commands/run_solver.py --recompute --parallel
```

Parallel portfolio:

```
./Commands/construct_sparkle_parallel_portfolio.py
--portfolio-name exp-5.1 --solver Solvers/PbO-CCSAT-Generic/
./Commands/run_sparkle_parallel_portfolio.py
--instances Instances/pap_experiments_instances --portfolio-name exp-5.1
```

5.2 Scaling experiment

The scaling experiment will demonstrate the performance of the portfolio when scaling the number of independent copies of the solver, running with different random seeds. This will be done by using the twelve same problem instances as in Section 5.1 and the solver will be PbO-CCSAT-Generic [11]. The scaling will start at two solver variations and will be doubled until 512 solver variations will be reached. The variations are created by the seed of the solver, changing the seed number will change the behaviour of the solver. This is because solvers make decisions using a pseudo-random number generator and the seed is the input of the generator. In Sparkle, the default seed number is one, which will also be the starting seed for this experiment. The first scale, scaling 2, will use a portfolio containing PbO-CCSAT-Generic with seed 1 and seed 2, the second scale, scaling 4, will use seed 1,2,3,4 and this will extend linearly for all scales. The size of the portfolio for each experiment is the same size of jobs that ran in parallel, this was achieved by setting the `number_of_runs_in_parallel` to the size of the portfolio. Unlike experiment 5.1, this experiment is only ran twice due to time constraints. In the Tables 3,4,5,6 each repeat will be noted by the suffix “ - 1 ” or “ - 2 ”.

The performance of the portfolios will be measured by the solving duration, the total duration, the duration of job allocation (job overhead), and the solver overhead. The solving duration is the sum of the fastest solver duration on each instance. The total duration is the time between the first job is allocated on the grace cluster and the time the last job is finished. The overhead is the difference between the total duration and the solver duration and is split into job overhead and solver overhead. The job overhead is part of the overhead but contains only the time that Grace needs to allocate a job on a free slot. The solver overhead is the difference between the overhead and the job overhead.

The reason that the measurements are useful is that it tries to show the difference between the optimal (perfect) portfolio performance and the actual portfolio performance. The solver overhead can be used to compare itself with the overhead of a single solver, to see if scaling a portfolio

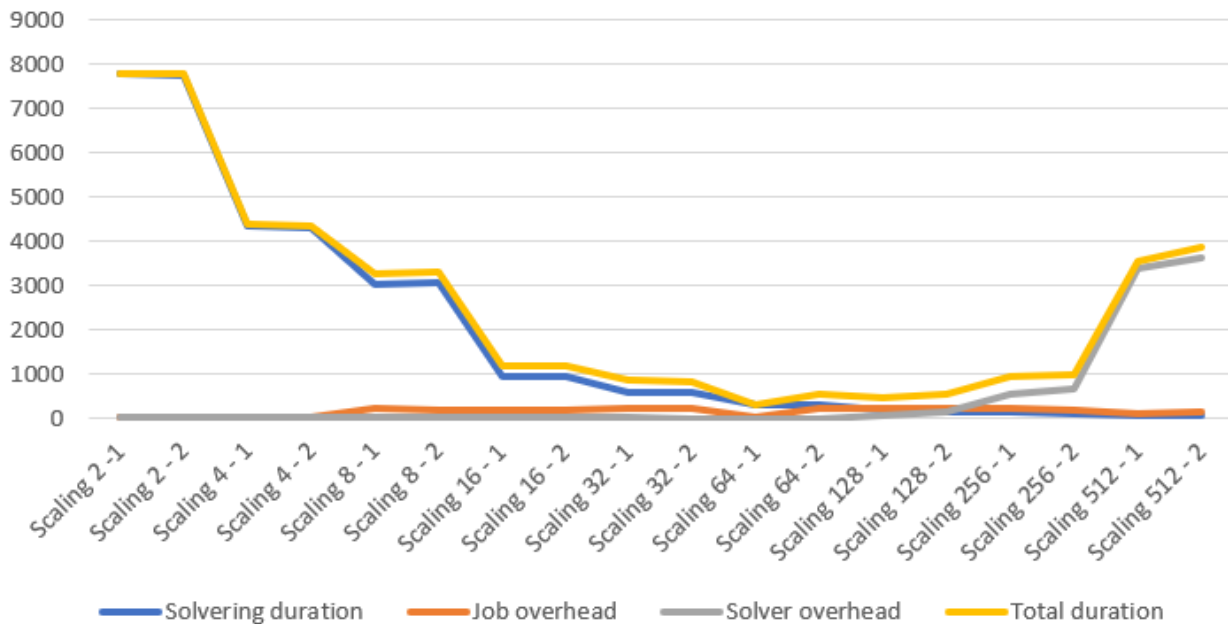
causes additional overhead, and if there is an ideal portfolio size in which the overhead and solving duration are optimised.

Table 3: The results of the scaling experiment

The shown time is in wallclock time(seconds)

Experiment	Solving duration	Total duration	Job overhead	Solver overhead
Scaling 2 - 1	7775.13976	7808	13	19.86024
Scaling 2 - 2	7763.02517	7797	14.5	19.47483
Scaling 4 - 1	4339.06393	4378	20	18.93607
Scaling 4 - 2	4294.0182	4332	20	17.9818
Scaling 8 - 1	3012.40122	3261	233.75	14.84878
Scaling 8 - 2	3087.40345	3300	198	14.59655
Scaling 16 - 1	969.23	1172	182.25	20.52
Scaling 16 - 2	969.61	1189	200.5	18.89
Scaling 32 - 1	603.23	855	222.96875	28.80125
Scaling 32 - 2	573.11	850	240.25	7.8287
Scaling 64 - 1	295.89	328	27.28125	4.82875
Scaling 64 - 2	291.98	546	246.28125	7.73875
Scaling 128 - 1	174.628304	456	219.59375	61.77625
Scaling 128 - 2	136.2	535	228.875	169.925
Scaling 256 - 1	144.96	936	237.453125	553.586875
Scaling 256 - 2	109.15	990	208.234375	672.615625
Scaling 512 - 1	66.55	3546	100.668	3378.782
Scaling 512 - 2	66.55	3870	160.572	3642.878

Figure 3: The results of the scaling experiment



The results from the experiment, as seen in Table 3 and Figure 3, show several trends. The solving duration decreases -as expected- when the portfolio size increases. On the contrary, the total duration decreases until scaling 64 and it even rapidly increases at scaling 256 and scaling 512. The reason for this is the increase in solver overhead, which increases from scaling 64 and onward. There are two possible explanations for the occurrence of an increase in solver overhead. First, the cluster could have issues with handling a large amount of *SCANCEL* commands, which is used to cancel jobs and executing the commands with a delay. Second, the job could take longer to cancel itself since a job is not removed from the cluster by the command. Instead, the job receives a signal which orders it to terminate itself, and a job could malfunction by either responding to the signal delayed or not at all. Another visible trend is that starting from scaling 8, the job overhead jumps from a value close to 15 seconds to a value around 200 seconds. To gain more insight into these trends the solving duration, as well as the job overhead and finally, the solver overhead will be further laid out by dividing the values over each instance.

Table 4: Solving duration of the scaling experiment for each instance [1/2]

The shown time is in wallclock time(seconds)

Experiments	1	2	3	4	5	6
Scaling 2 - 1	617.226	40.0705	2.61909	66.0515	1047.56	112.903
Scaling 2 - 2	615.711	39.7278	2.61911	66.4632	1047.89	113.906
Scaling 4 - 1	142.909	40.0302	2.62047	61.6609	947.423	113.215
Scaling 4 - 2	141.338	39.7486	2.61393	61.6764	907.866	133.374
Scaling 8 - 1	143.99	2.47451	1.90187	13.2067	298.948	113.524
Scaling 8 - 2	144.623	2.45854	1.91279	13.1044	304.176	112.209
Scaling 16 - 1	143.656	2.47563	1.92776	13.1589	263.44	81.826
Scaling 16 - 2	143.031	2.46436	1.91032	13.1273	262.383	80.8171
Scaling 32 - 1	7.31694	2.47723	1.93571	1.75906	89.97	82.1803
Scaling 32 - 2	7.31619	2.51213	2.02872	1.73456	90.2146	81.139
Scaling 64 - 1	7.3091	2.47392	0.244388	1.7191	94.0032	63.5464
Scaling 64 - 2	7.43889	2.48951	0.246015	1.74337	47.2311	63.9643
Scaling 128 - 1	7.33814	2.49891	0.136804	1.73572	26.5555	63.5934
Scaling 128 - 2	7.33731	2.5096	0.136509	1.71898	13.2815	31.8469
Scaling 256 - 1	7.41834	1.83645	0.072849	1.73947	10.1743	40.2369
Scaling 256 - 2	7.32076	1.80777	0.073139	3.08754	13.4494	32.0764
Scaling 512 - 1	2.58685	0.80658	0.072139	1.74083	10.2211	40.3761
Scaling 512 - 2	2.62517	0.812128	0.071634	1.72417	10.1258	32.1944

Table 5: Solving duration of the scaling experiment for each instance [2/2]

The shown time is in wallclock time(seconds)

Experiments	7	8	9	10	11	12
Scaling 2 - 1	22.2122	1225.7	6.9538	1.33367	1632.47	3000
Scaling 2 - 2	22.4546	1225.44	6.93835	1.31511	1620.56	3000
Scaling 4 - 1	22.3551	658.014	6.95634	1.33992	1617.9	723.64
Scaling 4 - 2	22.4024	659.102	6.96444	1.31943	1619.16	718.403
Scaling 8 - 1	7.3732	635.442	3.07333	1.25461	1570.74	220.473
Scaling 8 - 2	7.38389	654.727	3.04489	1.25794	1622.38	220.126
Scaling 16 - 1	0.555941	26.504	3.08136	0.970107	224.834	206.805
Scaling 16 - 2	0.549536	26.488	3.04717	0.97028	225.199	209.62
Scaling 32 - 1	0.322493	53.4917	3.0689	0.486446	149.329	210.895
Scaling 32 - 2	0.320874	26.7195	3.06986	0.489347	149.425	208.14
Scaling 64 - 1	0.318797	27.0211	2.26361	0.281899	80.4603	15.5293
Scaling 64 - 2	0.323549	53.3798	2.2574	0.286102	81.4766	31.1446
Scaling 128 - 1	0.325075	26.9023	2.59016	0.163215	17.2172	30.5697
Scaling 128 - 2	0.325356	26.9294	2.28074	0.16464	34.0782	15.5945
Scaling 256 - 1	0.321753	18.5162	0.094808	0.167674	33.8516	30.5312
Scaling 256 - 2	0.331569	18.4717	0.096828	0.164086	16.5933	15.68
Scaling 512 - 1	0.107514	1.71828	0.095419	0.065699	8.1916	7.37746
Scaling 512 - 2	0.107306	3.12258	0.095153	0.066854	8.18758	7.42152

Tables 4 and 5 confirm the trend seen in Table 3, which is that the solving duration decreases when the portfolio size increases. Note that the running time of 3000 seconds is the set cutoff time of the experiment and that these instances have not been solved at that point. Further note, that when comparing the increase in overhead with the reduction of solving duration it might be beneficial to run a small-scale experiment with a short cutoff time e.g. 100 seconds. After the short experiment finishes only rerun the instances that were not solved on a larger size portfolio and with a longer cutoff time.

Table 6: Job overhead of each instance

The job overhead of Scaling 512 is excluded because the size of the delay causes the jobs to start asynchronously and cannot be compartmentalised to instances. The shown time is in wallclock time(seconds)

Experiments	1	2	3	4	5	6	7	8	9	10	11	12
Scaling 2 - 1	2	0.5	1	1	1	1	1	1	1	1	1	1.5
Scaling 2 - 2	3	1	1	1	1	1	1	1	0.5	0.5	1	1
Scaling 4 - 1	4	1.5	1.5	1.5	1	1.5	1.5	1.5	1.5	1.5	1.5	1.5
Scaling 4 - 2	4	0.75	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
Scaling 8 - 1	5	4.13	26	27	16	28.75	5	21.13	23	25.1	26.5	18.1
Scaling 8 - 2	5	3.13	26.5	27.1	15.13	6.13	6.13	22	22.13	25.5	28	8.13
Scaling 16 - 1	4	4.06	26	26	15	5	17	28	1.06	25	27	1
Scaling 16 - 2	5	4	25.94	26.1	15.06	4.06	10	27.06	2.06	25.4	27.7	25.1
Scaling 32 - 1	3	20	26	26	26	25	6	27	6.81	24	27	3
Scaling 32 - 2	4	19	23.97	24.9	24.97	27	7	27	29	25.1	25.9	0.03
Scaling 64 - 1	1	2	1.97	2.42	2.45	1.97	1.97	2.55	1.97	1.97	1.88	1.97
Scaling 64 - 2	4	19	25	27	25	10	27	27	26	24	26	6
Scaling 128 - 1	4	19	25	13	25	26.94	25	0.09	25	27	25	9
Scaling 128 - 2	3	20	25	25	12	25	13	23	19	22	25	4
Scaling 256 - 1	4	18	18	15	23	9	17	8	21	13	6	18
Scaling 256 - 2	4	18	18	26	44	20	11	15	8	12	16	5

The sub-caption of Table 6 mentions that on a larger delay the jobs start to begin solving new problem instances asynchronously. This is something that the code does account for, and each job will have an equal chance to outperform the current fastest solver on the instance. The only drawback to this is that the logging system used to find the starting time of an instance cannot be used anymore. To further elaborate on this, if a solver in a portfolio solves the instance in fifteen seconds, another solver whose start was delayed by two seconds will also get fifteen seconds to try and solve the instance. This causes each delay on a job to further delay the start of the next job and that is what causes the asynchronicity.

Table 7: Solver overhead when scaled shown as the average duration of overhead on all instances

*Scaling 1 is average of the results from experiment 5.1. The shown time is in wallclock time(seconds)

Experiment	Average solver overhead	Single step increase	Total increase
Scaling 1*	50.8447	-	-
Scaling 2	19.6675	-61.3%	-61.3%
Scaling 4	18.4589	-6.1%	-63.7%
Scaling 8	14.7227	-20.2%	-71.0%
Scaling 16	19.705	33.8%	-61.2%
Scaling 32	18.315	-7.1%	-64.0%
Scaling 64	6.2838	-65.7%	-87.6%
Scaling 128	115.8506	1743.6%	127.9%
Scaling 256	613.1013	429.2%	1105.8%
Scaling 512	3510.83	472.6%	6805.0%

The results of the solver overhead, as shown in Table 7 (complete table in Appendix 12), show that the solver overhead reduces when the portfolio size increases and the solving time decreases. However, this trend is broken at Scaling 128, where the solver overhead increases. This indicates that some jobs continue running even after a solver has already solved that instance. In the discussion, several possible causes for this are discussed. Furthermore, possible solutions to combat the problem will also be discussed.

Command used for the experiment:

Scaling size **SIZE**:

```
./Commands/construct_sparkle_parallel_portfolio --portfolio-name scalingSIZE
--solver Solvers/Pb0-CCSAT-Generic/,SIZE
./Commands/run_sparkle_parallel_portfolio.py
--instances Instances/pap_experiments_instances --portfolio-name scalingSIZE
```

The **SIZE** should be varied in the commands above to get all the commands used for the experiment.

6 Discussion

In Chapter 5 the results of the conducted experiments are shown. It is within these results that the performance of the parallel algorithm portfolio is shown. In comparison, without a portfolio, the overhead on a single instance averages around 1.7 seconds, and with a portfolio, the overhead on a single instance is around 3.4 seconds. This seems like a large increase, nevertheless, as seen by attempt 2, with portfolio, in Table 10, the overhead can vary greatly. This is due to the delay which occurs after the **SBATCH** script is executed, the jobs are created, and when empty cores are allocated to the jobs. In Section 5.2 the delay is separated from the overall overhead as the job overhead. By looking at Table 6, in well-performing sections, the delay is around 0.5 to 2 seconds. In the other table sections of Table 6, the delay seems to average around 20 to 25 seconds. This delay could be caused by the Grace cluster which only allocates new solvers on a 30-second interval, due to high activity on the cluster. However, later attempts to recreate the issue in this manner were unsuccessful. This suggests that the issue might be caused by something else. One possible explanation could be that one week after conducting the experiments the **SCRATCH** directory of the Grace cluster had to be cleaned since it almost ran out of space. This could have affected the working memory of the nodes and resulted in the problem. Therefore, when experiments are taking longer than expected be sure to check the accounting logs of the cluster to see if a similar issue occurs.

When looking at Table 2, the results of experiment 1 seem to suggest that the level of difficulty of the instance that is being solved does not affect the size of the overhead. Furthermore, there is also a possibility that the type of solver will not influence the amount of overhead that occurs. This is because the implementation only considers the job, and will cancel or stop a job at the expected moment without taking the solver into account. The difference in overhead that could occur is the time it takes for a solver to respond to a command given by the cluster.

Up until Scaling 64 of the scaling experiment, the solver overhead itself only improved in comparison to Scaling 1, as shown in Table 7. However, this includes the Experiment 10 which was severely slower. When excluding this from the average, the solver overhead of scaling 1 would be 20.67 and the improvement would still hold, all be it much less significant. After scaling 64 the solver overhead suddenly starts to increase drastically, a cause for this might be related to the issue of the

`SCRATCH` directory. Nonetheless, it is clear that when the portfolio is scaled to a large enough size it causes a strain upon the monitoring process and the cluster on which it is executed.

Another part of the delay is caused by the limitation of the Grace cluster, the running time of a job is only reported up to full seconds. To allow all solvers to be able to compute the full duration of their allotted time, the moment at which a job is cancelled is always rounded up to the next full second. Additionally, this limits the usefulness to try and further examine subsections of the solver overhead since each job has a delay of less than one second. Consequently, it would always return a one-second delay. Besides discussing conditions required for the solver overhead to increase rapidly, the resulting issue should also be considered. When looking at Table 3, the most logical conclusion is that there are jobs, which should have been cancelled, continue to run even after they have received the `SCANCEL` command. The reason for this is that the only other possible cause for an increase in solver overhead would be a delay in the response to the `SCANCEL` command either by SLURM or the job. However, if the delay were the reason there would be a steady increase in overhead, as shown in Table 7, this is not the case.

When reflecting on the methodology of this thesis it states that within the experiments the practical capabilities of parallel algorithm portfolios will be illustrated. However, the findings in Table 3 indicated some unexpected overhead, and the rest of the experiment chapter was used to gain more insight into the overhead and the reasons why it occurred. This led to the exclusion of two experiments. These experiments would show that not only different solver variations, but also different solvers could be run within the same portfolio. Furthermore, the experiments were supposed to showcase the ability of Sparkle to run deterministic, as well as non-deterministic solvers. This limitation could be refuted by saying that the nature of the paper drifted towards presenting the different causes of overhead, and demonstrating a somewhat trivial capability of being able to execute the portfolio with different solvers would not fit within the scope of this study. Consequently, a recommendation for future work would be to study these capabilities.

In the research aim, in Chapter 1, the goal is formulated that this thesis will show the potency of parallel algorithm portfolios and its ability to improve the wallclock execution time. As observed in Table 3 the implementation of parallel algorithm portfolios is able to reduce the wallclock time of 16228 seconds to 4355 seconds with just a portfolio of four solvers. This reduces the time by almost 75%. Furthermore, the extension of parallel algorithm portfolios into Sparkle enables solver developers to support them in their development process from constructing a portfolio to running the portfolio and reporting its findings.

When processing the results of the experiments it became clear that there were several forms of overhead causing delays during the running of the portfolio. For future work, we will discuss the two most significant causes of overhead and possible implementations which would resolve or at least alleviate the issue. First, there is the Job overhead, which caused severely increased delays when executing a lot of workload. A way to possibly help reduce the frequency of this occurring is to remove the temporary files of cancelled jobs directly after they are cancelled. Instead of doing this after all jobs have finished running. This will reduce the running space of the portfolio, hopefully, making enough space to prevent the issue from occurring. Second, the issue which would be beneficial to solve is the issue that occurs at portfolios at a size larger than 64 solvers. The most probable cause of the increase in solver overhead is that some solvers did not receive or properly respond to cancel commands. A possible solution for this would be to implement an additional check in the monitoring function of `run_sparkle_parallel_portfolio` to cancel jobs that have

either failed to receive their cancel command or have failed to follow the cancel command.

To conclude, this thesis has introduced parallel algorithm portfolios to Sparkle. Within this thesis, the design of the system is shown, and the implementation of the design is outlined by looking at the commands and their options, as well as by conducting several experiments. The results show the implementation has a clear reduction of the overall wallclock time with the limitation of creating a portfolio with a size of 64 solvers. When scaling a portfolio above 64 solvers an unexpected increase of overhead occurs. The recommendations for future work are the introduction of an additional check during the running of the portfolio to prevent the increase in overhead. Secondly, an effort could be made to reduce the memory usage during the running of the portfolio by removing temporary files whilst the portfolio is running instead of after. Although the study has various limitations, it shows that the use of parallel algorithm portfolios in Sparkle can have a beneficial impact on its users.

References

- [1] H. Hoos, “Sparkle: A PbO-based Multi-agent Problem-solving Platform,” tech. rep., Department of Computer Science, University of British Columbia, 2015.
- [2] K. van der Blom, C. Luo, and H. H. Hoos, “Sparkle: Towards automated algorithm configuration for everyone,” 2019.
- [3] C. P. Gomes and B. Selman, “Algorithm portfolios,” *Artificial Intelligence*, vol. 126, no. 1, pp. 43 – 62, 2001. Tradeoffs under Bounded Resources.
- [4] B. A. Huberman, R. M. Lukose, and T. Hogg, “An economics approach to hard computational problems,” *Science*, vol. 275, no. 5296, pp. 51–54, 1997.
- [5] C. P. Gomes and B. Selman, “Algorithm portfolio design: Theory vs. practice,” in *UAI*, 2013.
- [6] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, Y. Shoham, *et al.*, “A portfolio approach to algorithm selection,” in *IJCAI*, vol. 3, pp. 1542–1543, 2003.
- [7] M. Lindauer, H. Hoos, and F. Hutter, “From sequential algorithm selection to parallel portfolio selection,” in *International Conference on Learning and Intelligent Optimization*, pp. 1–16, Springer, 2015.
- [8] M. Lindauer, H. H. Hoos, F. Hutter, and T. Schaub, “Autofolio: An automatically configured algorithm selector,” *Journal of Artificial Intelligence Research*, vol. 53, pp. 745–778, 2015.
- [9] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, “SATzilla: Portfolio-based Algorithm Selection for SAT,” *Journal of Artificial Intelligence Research*, vol. 32, pp. 565–606, 06 2008.
- [10] M. Heule, M. Jarvisalo, and T. Balyo, “SAT competition 2016 - Benchmarks.” <https://baldur.iti.kit.edu/sat-competition-2016/index.php?cat=benchmarks>, 2016. Accessed: 2020-07-12.
- [11] C. Luo, H. Hoos, and S. Cai, “PbO-CCSAT: Boosting local search for satisfiability using programming by optimisation,” in *International Conference on Parallel Problem Solving from Nature*, pp. 373–389, Springer, 2020.
- [12] M. J. Heule, “Avoiding Monochromatic Solutions of $a + b = c$ and $a^2 + b^2 = c^2$,” *SAT COMPETITION 2016*, p. 63, 2016.
- [13] “Grace User Guide,” 2019. Unpublished internal document.

Appendices

A Listings of the User guide

Listing 1: Decision algorithm example

```
#!/bin/bash

### Example usage of Sparkle for parallel algorithm portfolio
### The example illustrates the use of an decision algorithm that uses runtime
    as measurement

#### Initialise the Sparkle platform

Commands/initialise.py

#### Add instances

# Add instances (in this case for the portfolio) in a given directory, without
    running solvers or feature extractors
# Note that you should use the full path to the folder containing the instance(
    s)

Commands/add_instances.py --run-solver-later --run-extractor-later Examples/
    Resources/Instances/PTN/

#### Add solvers

# Add a solver with a wrapper containing the executable name of the solver and
    a string of command line parameters, without running the solver yet
# The path used should be the full path to the solver directory and should
    contain the solver executable and the 'sparkle_smac_wrapper.py' wrapper

# If needed solvers can also include additional files or scripts in their
    directory, but try to keep additional files to a minimum as it speeds up
    copying.
# Use the --solver-variations option to set the default number of solver
    variations of a solver which will be used when a portfolio is constructed.

Commands/add_solver.py --run-solver-later --deterministic 0 Examples/Resources/
    Solvers/CSCCSat/
Commands/add_solver.py --run-solver-later --deterministic 0 Examples/Resources/
    Solvers/MiniSAT/
Commands/add_solver.py --run-solver-later --deterministic 0 Examples/Resources/
    Solvers/Pb0-CCSAT-Generic/

#### Construct the portfolio

# The construction of the portfolio uses all the added solvers in the Solver/
    directory and keeps in mind the default overwrite setting
```

```

# Overwrite is only used if a portfolio is constructed holding the same name as
  an already existing portfolio.
# This will happen by default unless the nickname option is used then this will
  throw an error

# The --nickname option can be used to name your portfolio, the option must be
  followed by a nickname to name your portfolio.
# For example '--nickname runtime_experiment', if this option is not used then
  the default nickname is used
# This is sparkle_parallel_portfolio
# Without using the --solver option all solvers will be added, if you want, for
  example, only a subset of solvers from the Solver/ directory
# you can use a space seperated list, like --solver Solvers/CSCCSat Solvers/Pb0
  -CCSAT-Generic or --solver Solvers/TCA Solvers/MiniSAT

# In order to add multiple variations of a single solver you have to add ',
  number_of_solver_variations' within the space seperated solver list.
# For example --solver Solvers/Pb0-CCSAT-Generic,4 which will create a
  portfolio containing four variations of Pb0-CCSAT-Generic

Commands/construct_sparkle_parallel_portfolio.py --nickname runtime_experiment

#### Run the portfolio

# By running the portfolio a list of jobs will be created which will be
  executed by the cluster.
# Use the --cutoff-time to specify the allotted time of which the portfolio is
  allowed to run.
# add --portfolio-name to specify a portfolio otherwise it will select the
  latest constructed portfolio

# The --instance-paths option must be followed by a space seperated list of
  paths to an instance or an instance set.
# For example --instance-paths Instances/Instance_Set_Name/Single_Instance
  Instances/Other_Instance_Set_Name

Commands/run_sparkle_parallel_portfolio.py --instance-paths Instances/PTN/ --
  portfolio-name runtime_experiment

#### Generate the report

# The report details the experimental procedure and performance information.
# This will be located at Components/Sparkle-latex-generator-for-parallel-
  portfolio/Sparkle_Report.pdf

Commands/generate_report.py

```

Listing 2: Optimisation algorithm example

```

#!/bin/bash

### Example usage of Sparkle for parallel algorithm portfolio
### The example illustrates the use of an optimisation algorithm that uses
  quality as measurement

```

```
#### Initialise the Sparkle platform

Commands/initialise.py

#### Add instances

# Add instances (in this case for the portfolio) in a given directory, without
# running solvers or feature extractors
# Note that you should use the full path to the folder containing the instance(
# s)

Commands/add_instances.py --run-solver-later --run-extractor-later Examples/
Resources/CCAG/Instances/CCAG/

#### Add solvers

# Add a solver with a wrapper containing the executable name of the solver and
# a string of command line parameters, without running the solver yet
# The path used should be the full path to the solver directory and should
# contain the solver executable and the 'sparkle_smac_wrapper.py' wrapper

# If needed solvers can also include additional files or scripts in their
# directory, but try to keep additional files to a minimum as it speeds up
# copying.
# Use the --solver-variations option to set the default number of solver
# variations of a solver which will be used when a portfolio is constructed.

Commands/add_solver.py --run-solver-later --deterministic 0 Examples/Resources/
CCAG/Solvers/FastCA/
Commands/add_solver.py --run-solver-later --deterministic 0 Examples/Resources/
CCAG/Solvers/TCA/

#### Construct the portfolio

# The construction of the portfolio uses all the added solvers in the Solver/
# directory and keeps in mind the default overwrite setting
# Overwrite is only used if a portfolio is constructed holding the same name as
# an already existing portfolio.
# This will happen by default unless the nickname option is used then this will
# throw an error

# The --nickname option can be used to name your portfolio, the option must be
# followed by a nickname to name your portfolio.
# For example '--nickname quality_experiment', if this option is not used then
# the default nickname is used
# This is sparkle_parallel_portfolio
# Without using the --solver option all solvers will be added, if you want, for
# example, only a subset of solvers from the Solver/ directory
# you can use a space separated list, like --solver Solvers/FastCA or --solver
# Solvers/TCA

# In order to add multiple variations of a single solver you have to add ',
# number_of_solver_variations' within the space separated solver list.
```

```
# For example --solver Solvers/FastCA,4 wich will create a portfolio containing
  four variations of FastCA

Commands/construct_sparkle_parallel_portfolio.py --nickname quality_experiment

#### Run the portfolio

# By running the portfolio a list of jobs will be created which will be
  executed by the cluster.
# Use the --cutoff-time to specify the allotted time of which the portfolio is
  allowed to run.
# add --portfolio-name to specify a portfolio otherwise it will select the
  latest constructed portfolio

# The --instance-paths option must be followed by a space seperated list of
  paths to an instance or an instance set.
# For example --instance-paths Instances/Instance_Set_Name/Single_Instance
  Instances/Other_Instance_Set_Name

Commands/run_sparkle_parallel_portfolio.py --instance-paths Instances/CCAG/ --
  performance-measure QUALITY_ABSOLUTE --portfolio-name quality_experiment

#### Generate the report

# The report details the experimental procedure and performance information.
# This will be located at Components/Sparkle-latex-generator-for-parallel-
  portfolio/Sparkle_Report.pdf

Commands/generate_report.py
```

B Results of experiment 5.1

Table 8: 5.1 Without portfolio 2

Instance	Total duration	Solving duration	Overhead
Ptn-7824-b20	2 seconds	1.32669 seconds	0.67331 seconds
Ptn-7824-b18	3 seconds	2.60099 seconds	0.39901 seconds
Ptn-7824-b13	23 seconds	22.3015 seconds	0.6985 seconds
Ptn-7824-b09	40 seconds	39.5757 seconds	0.4243 seconds
Ptn-7824-b11	67 seconds	65.7377 seconds	1.2623 seconds
Ptn-7824-b15	194 seconds	193.351 seconds	0.649 seconds
Ptn-7824-b06	1444 seconds	1442.56 seconds	1.44 seconds
Ptn-7824-b01	2478 seconds	2477.07 seconds	0.93 seconds
Ptn-7824-b07	3001 seconds	3000.05 seconds	0.95 seconds
Ptn-7824-b05	3001 seconds	3000.06 seconds	0.94 seconds
Ptn-7824-b04	3003 seconds	3000.05 seconds	2.95 seconds
Ptn-7824-b03	3006 seconds	3000.07 seconds	5.93 seconds

Table 9: 5.1 Without portfolio 3

Instance	Total duration	Solving duration	Overhead
Ptn-7824-b20	3 seconds	2.44982 seconds	0.55018 seconds
Ptn-7824-b18	4 seconds	2.62 seconds	1.38 seconds
Ptn-7824-b13	24 seconds	22.442 seconds	1.558 seconds
Ptn-7824-b09	41 seconds	39.7472 seconds	1.2528 seconds
Ptn-7824-b11	67 seconds	65.6625 seconds	1.3375 seconds
Ptn-7824-b15	196 seconds	194.977 seconds	1.023 seconds
Ptn-7824-b06	1399 seconds	1398.25 seconds	0.75 seconds
Ptn-7824-b01	2382 seconds	2380.79 seconds	1.21 seconds
Ptn-7824-b07	3001 seconds	3000.02 seconds	0.98 seconds
Ptn-7824-b05	3003 seconds	3000.05 seconds	2.95 seconds
Ptn-7824-b04	3003 seconds	3000.05 seconds	2.95 seconds
Ptn-7824-b03	3003 seconds	3000.05 seconds	2.95 seconds

Table 10: 5.1 With portfolio 2

Instance	Total duration	Solving duration	Overhead
Ptn-7824-b20	8 seconds	1.32852 seconds	6.67148 seconds
Ptn-7824-b18	9 seconds	2.62897 seconds	6.37103 seconds
Ptn-7824-b13	29 seconds	22.1711 seconds	6.8289 seconds
Ptn-7824-b09	46 seconds	40.0358 seconds	5.9642 seconds
Ptn-7824-b11	72 seconds	65.9823 seconds	6.0177 seconds
Ptn-7824-b15	200 seconds	193.796 seconds	6.204 seconds
Ptn-7824-b06	1421 seconds	1414.38 seconds	6.62 seconds
Ptn-7824-b01	2497 seconds	2490.21 seconds	6.79 seconds
Ptn-7824-b07	3006 seconds	3000 seconds	6 seconds
Ptn-7824-b05	3006 seconds	3000 seconds	6 seconds
Ptn-7824-b04	3008 seconds	3000 seconds	8 seconds
Ptn-7824-b03	3011 seconds	3000 seconds	11 seconds

Table 11: 5.1 With portfolio 3

Instance	Total duration	Solving duration	Overhead
Ptn-7824-b20	2 seconds	1.33808 seconds	0.66192 seconds
Ptn-7824-b18	3 seconds	2.58902 seconds	0.41098 seconds
Ptn-7824-b13	23 seconds	22.2955 seconds	0.7045 seconds
Ptn-7824-b09	41 seconds	39.8896 seconds	1.1104 seconds
Ptn-7824-b11	67 seconds	66.0713 seconds	0.9287 seconds
Ptn-7824-b15	195 seconds	194.308 seconds	0.692 seconds
Ptn-7824-b06	1473 seconds	1471.65 seconds	1.35 seconds
Ptn-7824-b01	2496 seconds	2492.74 seconds	3.26 seconds
Ptn-7824-b07	3001 seconds	3000 seconds	1 seconds
Ptn-7824-b05	3003 seconds	3000 seconds	3 seconds
Ptn-7824-b04	3003 seconds	3000 seconds	3 seconds
Ptn-7824-b03	3006 seconds	3000 seconds	6 seconds

C Results of experiment 5.2

Table 12: Solver overhead when scaled shown as the average duration of overhead on each instance

* Scaling 1 are the results from experiment 5.1

Experiment	Solver overhead	Single step increase	Total increase
Scaling 1 - 1*	19.22199	-	-
Scaling 1 - 2*	82.46731	-	-
Scaling 2 - 1	19.86024	-60.9%	-60.9%
Scaling 2 - 2	19.47483	-61.7%	-61.7%
Scaling 4 - 1	18.93607	-3.7%	-62.8%
Scaling 4 - 2	17.9818	-8.6%	-64.6%
Scaling 8 - 1	14.84878	-19.6%	-70.8%
Scaling 8 - 2	14.59655	-20.1%	-71.2%
Scaling 16 - 1	20.52	39.4%	-59.6%
Scaling 16 - 2	18.89	28.3%	-62.8%
Scaling 32 - 1	28.80125	46.2%	-43.4%
Scaling 32 - 2	7.8287	-60.3%	-84.6%
Scaling 64 - 1	4.82875	-73.6%	-90.5%
Scaling 64 - 2	7.73875	-57.7%	-84.8%
Scaling 128 - 1	61.77625	883.1%	21.5%
Scaling 128 - 2	169.925	2604.2%	234.2%
Scaling 256 - 1	553.586875	377.8%	988.8%
Scaling 256 - 2	672.615625	408.6%	1222.9%
Scaling 512 - 1	3378.782	451.1%	6545.3%
Scaling 512 - 2	3642.878	492.2%	7064.7%

D Grace specification

D.1 Hardware

Note: The specifications are duplicated from Graces' internal user guide[\[13\]](#)

At the time of this writing (January 2019) Grace is constituted from a total of 34 nodes. 26 of them are so-called CPU nodes, 10 of them are GPU nodes with 2 GPUS (8 nodes) or 4 GPUs (2 nodes). They are connected with Infiniband FDR (~56 Gb/sec/link FD) The following section describes the hardware of the nodes. Grace is *homogeneous*, meaning that each CPU is the same and each GPU is the same. And it will stay homogeneous. Each node has 2 CPUs (processor) of 16 cores.

File server

```
Vendor ID: GenuineIntel
Model: Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
Advertised max. speed: 2.1 GHz
Max. speed: 3.0 GHz
Cache size: 20 MB
Memory: ~ 125 GB
```

Nodes

```
Vendor ID: GenuineIntel
Model: Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz
Advertised max. speed: 2.1 GHz
Max. speed: 3.0 GHz
Cache size: 40 MB
Memory: ~ 94 GB
```

Note: As the turbo option is disabled for consistency, it shouldn't reach the max. speed.

GPUs

```
Model: NVIDIA Corporation GP102 [GeForce GTX 1080 Ti] (rev a1)
Video Memory: 11GB
```

Power

```
Power when full load: ~ 11 KW
Power when idle: ~ 4 KW
```

D.2 Software

Operating System CentOS Linux

Scheduling System Slurm

E Example sparkle report

Parallel portfolio report

Sparkle

30th July 2021

1 Introduction

Sparkle [2] is a multi-agent problem-solving platform based on Programming by Optimisation (PbO) [1], and would provide a number of effective algorithm optimisation techniques (such as automated algorithm configuration, portfolio-based algorithm selection, etc.) to accelerate the existing solvers.

This experimental report is automatically generated by *Sparkle*. This report presents experimental results of *Sparkle* parallel portfolio containing 3 solver(s).

2 Experimental Preliminaries

This section presents the experimental preliminaries, including the list of solvers in the portfolio, the list of instance sets and information about the experimental setup.

2.1 Solvers

There are 3 solver(s) included in *Sparkle*, as listed below.

1. **CSCCSat**
2. **PbO-CCSAT-Generic**
3. **MiniSAT**

2.2 Instance Set(s)

There are 1 instance set(s) included in *Sparkle*, as listed below.

1. **PTN**, number of instances: 12

2.3 Experimental Setup

The experimental setup is described below.

Performance computation: *Sparkle* runs the portfolio one time on each instance. The cutoff time for the computation run is set to 300 seconds. The outcome of the computation is listed below. The scores of the outcomes are calculated according to PAR10, this means that for each instance the solver which solved the instance is scored its runtime and the remaining solvers are scored the runtime times ten. If however the portfolio reaches the cutofftime, which means that no solvers solved the instance, all solvers are scored the cutofftime times ten.

1. Solver **PbO-CCSAT-Generic**, was the best solver on **3** instance(s)
2. Solver **CSCCSat**, was the best solver on **4** instance(s)

3. 5 instance(s) remained unsolved

In the table below the computed PAR10 scores of all solvers have been listed aswell as the portfolio itself.

Portfolio results

Portfolio nickname	PAR10	#Timeouts	#Cancelled	#Best solver
test3	15156.91	5	0	7

Solver results

Solver	PAR10	#Timeouts	#Cancelled	#Best solver
CSCCSat	24038.83	5	3	4
PbO-CCSAT-Generic	27118.08	5	4	3
MiniSAT	36000.0	5	7	0

2.4 Scatter Plot Analysis

Figure 1 shows the empirical comparison between the actual parallel portfolio in *Sparkle* and the single best solver (*SBS*).

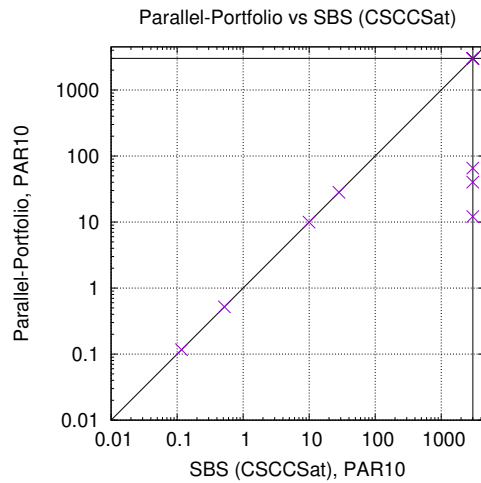


Figure 1: Empirical comparison between the actual parallel portfolio in *Sparkle* and the *SBS*.

References

- [1] Holger H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.
- [2] Holger H. Hoos. Sparkle: A pbo-based multi-agent problem-solving platform. Technical report, Department of Computer Science, University of British Columbia, 2015.