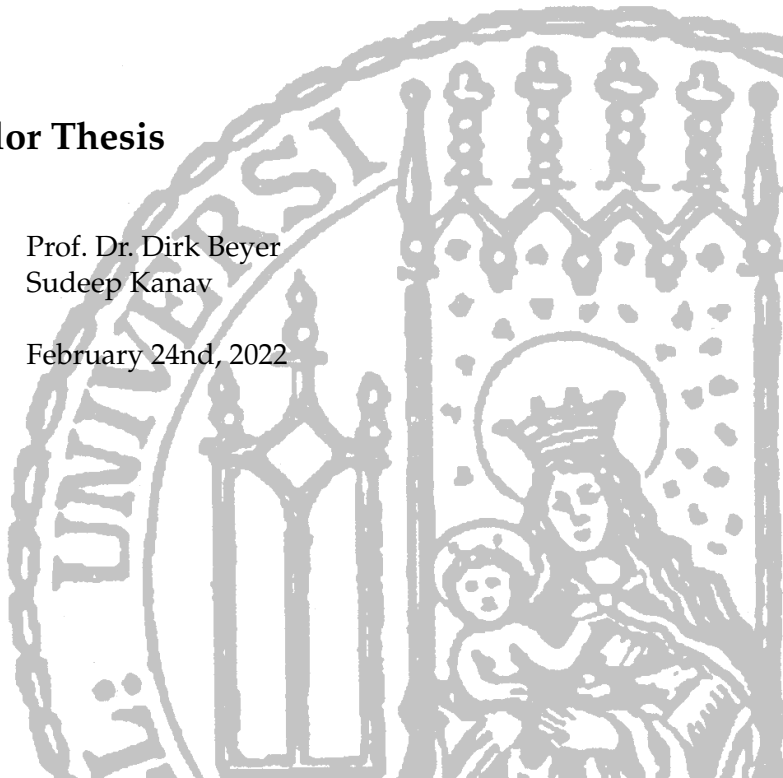
INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

# PARALLEL PORTFOLIO VERIFIER

## Developing a Verifier Based on Parallel Portfolio with CoVeriTeam

Tobias Kleinert



**Bachelor Thesis**

| | |
|---|---|
| **Supervisor** | Prof. Dr. Dirk Beyer |
| **Mentor** | Sudeep Kanav |
| **Submission Date** | February 24nd, 2022 |

# Statement of Originality

***English:***

## Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments.

***Deutsch:***

## Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, February 24nd, 2022 Tobias Kleinert

# Abstract

Because Automatic Software Verification uses a lot of resources, especially time, we tried to minimize the used wall-time for a specific verification task by composing multiple software verifiers into a PARALLEL PORTFOLIO composition. We implemented this composition in the framework COVERITEAM. This composition runs all given verifiers in parallel and uses the first produced result as its own result. We evaluated the composition with a large benchmark set of 8883 verification tasks and the software verifiers of the SV-COMP 21. With this approach we achieved up to 3 times faster wall-times and 30-60 % reduced energy consumption per score point compared with each single verifier in the composition. In order to achieve this results we used four times the number of CPU cores for each verification task. For this benchmark set we used 8 CPU cores per verification task. In conclusion we suggest the PARALLEL PORTFOLIO composition whenever a high amount of CPU cores and at least two different verifiers are available.

# Contents

# List of Figures

CHAPTER 1

# Introduction

Automatic Software Verification is a highly complex branch of computer science. Its computation requires a lot of resources and clever algorithms. That is why there are many different tools to solve such verification tasks. These tools have different strengths in different situations. This leads to situations where one tool is able to verify a specific software in under one second and another one needs hours for the same verification. With a different verification problem the positions may change. It is therefore highly beneficial to try to combine multiple tools to use each of their strengths.

For this we contribute a composition, the PARALLEL PORTFOLIO, to always use the fastest verifier. This composition is created and executed in the program COVERITEAM [3]. COVERITEAM provides its own language for composing multiple tools. These tools are downloaded and executed by COVERITEAM. The PARALLEL PORTFOLIO takes some tools and executes them in parallel until one tool produces a results, which satisfies a given condition. Thereupon the other tools will be stopped. We provide two execution methods for the execution of the PARALLEL PORTFOLIO. One uses MPI and the other the multiprocessing module of python.

For our evaluation we chose a large benchmark set of 8883 verification tasks and use different resource limitations. We compared the outcomes to determine which resource limitation impacts the performance at most. We have also investigated the performance on a computer cluster.

The goal of this work was to develop a composition, which provides a simple way to combine verification tools into a PARALLEL PORTFOLIO and compare the performance with the state of the art software verification tool CPACHECKER.

**Example 1** PARALLEL PORTFOLIO

**Input:** Program p, Specification s
**Output:** Verdict

```
1: verifier_1 := Verifier("CPAChecker")
2: verifier_2 := Verifier("ESBMC")
3: success_condition := verdict ∈ {TRUE, FALSE}
4: parallel_portfolio := ParallelPortfolio(verifier_1, verifier_2, success_condition)
5: result := parallel_portfolio.verify(p,s)
6: return (result.verdict, result.witness)
```

**Contributions** We make the following contributions:

1. We provide a new PARALLEL PORTFOLIO composition in the program COVERITEAM.
2. We provide two execution methods for this PARALLEL PORTFOLIO, the MPI execution and the fallback execution.
3. We implemented a shutdown procedure for the all actors in COVERITEAM.
4. We provide a detailed evaluation of the PARALLEL PORTFOLIO composition.

**Impact** We participated with a PARALLEL PORTFOLIO based verifier in the Software Verification Competition SV-COMP 22 [2]. Our verifier achieved very good results and a high amount of score points. However, because it is a combination of already existing tools, it was not included in the ranking of the competition. Nonetheless, it would have taken 2nd place in many categories in terms of score points.

Additionally our PARALLEL PORTFOLIO was used in the paper "*Construction of Verifier Combinations Based on Off-the-Shelf Verifiers*" [4]. In this paper, Beyer, Kanav, and Richter investigated multiple different compositions of verifiers with varying verifier amounts. The PARALLEL PORTFOLIO achieved the lowest wall time in their evaluation.

## Running example

One of the main ideas behind the PARALLEL PORTFOLIO is the combination of strengths of multiple verifiers. The two programs `while_infinite_loop_3.c` [1] and `sanfoundry_10_ground.c` [2] located in the `SV-Benchmark` repository [3] can be checked against the unreach-call specification. At the SV-COMP 22 [2] both programs were used. During the competition the verifier CPACHECKER [5] was able solve the first program while using 5 seconds of CPU time, but timeouted during the validation of the second program. The verifier ESBMC [12] on the other hand was able to solve the

[1] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/loops/while_infinite_loop_3.c`
[2] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/array-examples/sanfoundry_10_ground.c`
[3] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks`

second program in less than 1 second of CPU time and timeouted during the validation of the first one (see `sanfoundry_10_ground` and `while_infinite_loop` at SV-COMP 22). Our PARALLEL PORTFOLIO shown in Example 1 should combine these two verifiers and be able to verify both programs.

CHAPTER 2

# Prerequisites

This chapter summarizes the technological and conceptual prerequisites to create a parallel portfolio based composition in the program CoVeriTeam [3].

## 2.1 Parallel Portfolio

The term portfolio in general describes a collection of multiple things. For example in the economy a portfolio "*refers to any combination of financial assets such as stocks, bonds and cash*" [23] . In case of arts, portfolio describes a collection of artworks of an specific artist. With a portfolio of stocks, you want to minimize the risk of loosing a lot of money, in case one stock is decreasing, therefore you distribute your resources to a lot of stocks. This approach of risk distribution is also usable in computer science.

Huberman et al. [17] investigated the advantages of portfolios in context of hard computational problems. They differentiated between the sequential and the parallel portfolio. Whereas the sequential portfolio has two or more algorithms, which are executed independently and concurrently in sequence, the algorithms of the parallel one where executed in parallel. Each of those algorithms is trying to solve the same problem. They found mainly two advantages with this approach. First, the probability of a result was higher than using only one algorithm. Second, they achieved a potential performance increase. It was especially high in case of the Parallel Portfolio.

Software Verification also includes expensive computations. A software verifier calculates several program states and checks them against a certain specification. As a result we decided to use a Parallel Portfolio to combine multiple software verifiers to achieve better execution time and to increase the probability of a produced result.

To use a Parallel Portfolio, we implemented a new composition in CoVeriTeam.

This composition takes a set of verifying tools (see Sect. 2.2.1), which receive the same input and a success condition (see Sect. 3.1). Also they must produce the same type of output. These tools are executed in parallel and the first produced output is checked against this condition. If it matches the condition, all other tools are stopped, and the result is taken as a result of the whole PARALLEL PORTFOLIO composition. If no tool produces a valid result, an empty result is returned.

## 2.2 COVERITEAM

This section gives a brief overview of COVERITEAM. For more detailed information please look at the paper "*CoVeriTeam: On-Demand Composition of Cooperative Verification Systems*" [3].

### Artifact

An artifact in COVERITEAM is a file or some result. Artifacts used in this work are:

- **Program**: A program, which will be checked against a given specification
- **Specification**: A specification, which maybe the program fulfills. For example, is there any line in the program not reached
- **Verdict**: True, in case the specification is fulfilled, false otherwise
- **Witness**: Detailed result, how the verdict was achieved.
- **TestSuite**: Tests, which cover multiple test cases

These Artifacts cover the inputs and outputs for Verifiers and Validators in COVERITEAM.

### 2.2.1 Actor

Actors act on artifacts. They transform them or create new ones. In other words, an actor is a tool, which will be executed in COVERITEAM with artifacts as input and new or changed artifacts as output. COVERITEAM supports three types of actors, the utility actor, atomic actor, and composition.

#### 2.2.1.1 Atomic actor

Atomic actors are external binaries, which are executed by RUNEXEC which is part of the benchmarking framework BENCHEXEC [6]. The command line arguments, download location, allowed resource allocation and version of these programs are provided by YAML-files. COVERITEAM saves these tools after downloading them in a cache directory, whose location may be changed by the user. When it comes to the execution of an atomic actor, RUNEXEC constructs and launches a new process with the help of C-GROUPS[1]. This allows RUNEXEC to limit the resources like CPU time,

[1] `https://man7.org/linux/man-pages/man7/cgroups.7.html`

core amount and memory allocation of this process and measure the actually used resources. After the execution is finished, RUNEXEC collects the produced results and measurements and returns it to COVERITEAM. The produced results are Artifacts (see Sect. 2.2).

#### 2.2.1.2 Compositions

A composition is a new actor, which consists of one or (so far) two actors. Since a composition is a new actor, it is possible create compositions of compositions. At the moment four different compositions are available:

**Sequence** A sequential composition does exactly what the title suggests, it executes two actors one after another. Thereby it is important, that the outputs of the first actor are a superset of the inputs of the second one.

**Parallel** This composition executes two actors in parallel. It is important, that the inputs do not have a *name clash*. „A name clash between two sets $A$ and $B$ exists if there is a name in $A$ that is mapped to a different artifact type in $B$, more formally: $\exists(a, t_1) \in A; (a; t_2) \in B : t_1 \neq t_2$." [3]. The outputs must be disjoint.

**If-then-else (ITE)** If-then-else requires two actors and a condition. The condition is evaluated at the start of the composition and if it is true, the first actor will be executed. Otherwise the second actor will be executed.

**Repeat** This composition only takes one actor and again a condition. The actor is executed repeatedly, until the condition evaluates to true.

#### 2.2.1.3 Utility actor

Utility actors manipulate artifacts. For example, they are able to rename or filter keys in Artifacts.

## 2.3 Message Passing Interface (MPI)

"*MPI is a standardized and portable message-passing system. Message-passing systems are used especially on distributed machines with separate memory for executing parallel applications. With this system, each executing process will communicate and share its data with others by sending and receiving messages.*" [21]

MPI [14] defines an interface with methods to share data between processes. To use MPI in a program, first one needs to include/import the interface and then compile it with the MPI-compiler, a modified version of GCC[2]. To run the compiled program, the developer needs an MPI implementation on his system and to launch the program with the executable of this implementation. Because of the specification of MPI, there

[2] `https://gcc.gnu.org/`

must be an executable called MPIEXEC available. To launch the program, he needs to call `mpiexec your_executable`.

**Types of inter process communication** In MPI several methods for inter process communication are available, however they can be grouped in the following two categories:

- 1 to 1 communication: One process sends data to one another process
- collective communication: All processes in a group share data with each other

To communicate with each other, processes use communicators. A communicator has a unique ID and a group of processes. Every process has a unique ID in each communicator, therefore one process can have multiple IDs depending which communicator it is using to transfer data. To illustrate this, let us consider the following example:

```
#include <mpi.h>
#include <iostream>

int main()
{
    int my_rank;
    int world_size;

    int test = 0;

    MPI_Init(NULL, NULL);

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) {
        // Only executed in process 0
        int number = 100;
        MPI_Send(&number, 1, MPI_INT, 1, 13, MPI_COMM_WORLD);
    } else if (my_rank == 1) {
        // Only executed in process 1
        MPI_Recv(&test, 1, MPI_INT, 0, 13, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    printf("Process %d has stored number %d in variable test", my_rank, test);

    MPI_Finalize();
    return 0;
}
```

Listing 1: `MPI_Send` example

After compilation of this code with MPICC, it is executable with the command "`mpiexec -n 4 program_name`". This is an example of the expected output:

```
Process 1 has stored number 100 in variable test
Process 3 has stored number 0 in variable test
Process 0 has stored number 0 in variable test
Process 2 has stored number 0 in variable test
```

Listing 2: Output of Listing 1

Every process is running the same code, the only difference is the value of the `my_rank` variable. This means, only process 0 sends the number 100 and only process 1 receives it, because process 0 is calling `MPI_Send` and process 1 `MPI_Recv`. All other processes neither send nor receive a number, so they all output the default value of test, that is 0. The `-n` option is used to set the amount of spawned processes. However, one can also spawn processes during runtime:

```
#include <mpi.h>
#include <iostream>

int main() {
    int my_rank;
    MPI_Comm interCommunicator;

    MPI_Init(NULL, NULL);

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    printf("Spawner process with rank %d", my_rank);
    MPI_Comm_spawn(
        "mpi_hello_world_win.exe", // The program above
        MPI_ARGV_NULL,
        3, // Number of spawned processes
        MPI_INFO_NULL,
        0,
        MPI_COMM_SELF,
        &interCommunicator,
        MPI_ERRCODES_IGNORE
    );

    MPI_Finalize();
}
```

Listing 3: `MPI_Comm_Spawn` example

The output after executing this program with "`mpiexec -n 1 program_name`" looks like this:

```
Spawner process with rank 0
Process 1 has stored number 100 in variable test
Process 2 has stored number 0 in variable test
Process 0 has stored number 0 in variable test
```

Listing 4: Output of Listing 3

There are two points to note. First, the process ID 0 appears two times. Hence the processes spawned with `MPI_Comm_spawn` have their own world communicator. It is possible to address the parent by using `MPI_Comm_get_parent`. This method returns a communicator which includes the spawning process. Second, it is possible to spawn new processes at runtime, which will come in handy later.

CHAPTER 3

# Implementation

With all the requirements settled, we are now able to have a look at the implementation of the PARALLEL PORTFOLIO. First, we look at the basic structure of the PARALLEL PORTFOLIO, then how it is declared in the language. Afterwards we look at the generation of the actual executed python code. The final sections describes the two execution modes.

## 3.1 Basic parallel portfolio actor

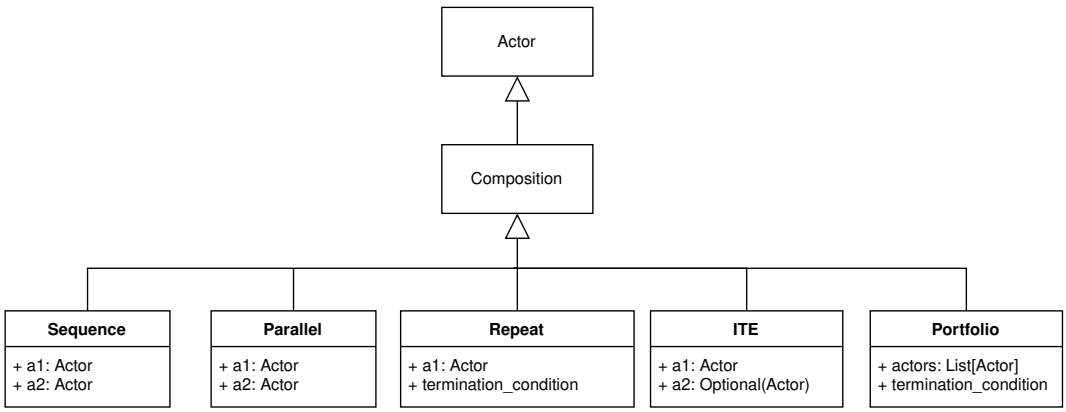



Figure 3.1: Variables in composition are init parameters

A PARALLEL PORTFOLIO is a subclass of composition like the four other compositions. As shown in Fig. 3.1, the PARALLEL PORTFOLIO takes a list of actors and a success condition.

**Type check** Every other compositions has a type check defined, for example in Sequence (see Sect. 2.2.1.2) the outputs of the first actor must be a superset of the

inputs of the second one. In the PARALLEL PORTFOLIO, such requirements are simple. All actors must require the same inputs and produce the same outputs. If this is not the case, the PARALLEL PORTFOLIO creation would fail because of a language exception.

**Success condition** As a reminder, the PARALLEL PORTFOLIO runs until one of its actors has produced a valid result or all actors have finished. The success condition decides whether a result is valid or not. The standard condition for verifiers, which we also used in Example 1, requires the result to have a key named verdict and if its value is true or false, the condition evaluates to true. To use other actors than verifiers, the user has to specify a custom success condition. This is a string with an expression and the evaluation is done by pythons own `eval`-method [1]. Due to the produced result being the only input, the user has to be aware of its content.

## 3.2 Language extension

### 3.2.1 The language itself

Compositions are written in a lightweight language provided by COVERITEAM. The parser for this language is generated by ANTLR [19]. This parser generates executable python code from the given composition descriptions. The generated python code is then executed by COVERITEAM. We use a grammar file with the grammar of ANTLR in version four to define this language. For this work, we added the PARALLEL PORTFOLIO to the language and implemented its python code generation. Now we explain this with the help of the following running examples.

```
1  actor : 'ActorFactory.create(' actor_type ','
2            name=(ID | STRING)
3            (',' version=(STRING | ID))? ')' # Atomic
4        | ID '(' id_list? ')'          # FunCall
5        | utility_actor                # Utility
6        | 'SEQUENCE' '('actor ',' actor ')'
7        | 'ITE' '(' exp ',' actor (',' actor)? ')'
8        | 'REPEAT' '(' quoted_ID ',' actor ')'
9        | 'PARALLEL' '(' actor ',' actor ')'
10       | 'PARALLEL_PORTFOLIO' '('  actor (',' actor )* (',' exp)')
           '
11       | ID               # ActorAlias
12       | '(' actor ')'          # Parenthesis
13       ;
```

Listing 5: Part of grammar rules

**Language definition** Listing 5 shows the grammar rules for actors. The new PARALLEL PORTFOLIO composition is located in line 10. In its parameters, at least one actor is required, following actors are comma separated and at the end a success condition needs to be specified (see Sect. 3.1). The actual creation of a PARALLEL PORTFOLIO may look like it's shown in Listing 6.

[1] See `https://docs.python.org/3/library/functions.html#eval`

```
cpachecker = ActorFactory.create(
        ProgramVerifier, "../actors/cpa-seq.yml", "default"
    );
esbmc = ActorFactory.create(
        ProgramVerifier, "../actors/esbmc-kind.yml", "default"
    );

success_condition = ELEMENTOF(verdict, {TRUE,FALSE});

portfolio = PARALLEL_PORTFOLIO(
    cpachecker,
    esbmc,
    success_condition
);
```

Listing 6: Portfolio creation in CoVeriTeams language

The example in Listing 6 creates the PARALLEL PORTFOLIO shown in Example 1. The success condition in this case is true, if there is a key called verdict in the produced results in CPACHECKER and ESBMC and its value is true or false.

```
def visitParallelPortfolio(self,
        ctx: CoVeriLangParser.ParallelPortfolioContext):

    self.pyp += "ParallelPortfolio(["
    for child in ctx.children:
        # IsInstance needed to support compositions and atomic actors
        if isinstance(child, CoVeriLangParser.ActorAliasContext):
            self.visit(child)
            self.pyp += ","
    self.pyp = self.pyp[:-1]
    self.pyp += "]"

    if ctx.exp():
        self.pyp += ","
        self.visit(ctx.exp())

    self.pyp += ")"
```

Listing 7: Parser code of PARALLEL PORTFOLIO

**Code generation** As mentioned above, the language generator will create python code from those statements. Listing 7 shows the generator implementation for the PARALLEL PORTFOLIO. Notice in line 2 the variable `self.pyp`, which holds the current generated python code. At the top, the call to the PARALLEL PORTFOLIO constructor is added with opening square bracket for the start of the actor list, then the actors need to be parsed. For that purpose we will loop trough every child in this `PortfolioContext` and check if this child is an actor. A child in this case is every type defined in the language. This includes every separating comma, the paranthesis, actors and the success condition. For every found child the visit method gets called, recursively parses and appends it to the source code. After that a comma is added. At last, the comma at the end has to be deleted, which is done in line 9. Now the list is finished and can be closed. Line 12-14 parses the success condition and appends it after the actor list. During this parsing, `True` and `False` are converted to classes of BENCHEXEC. Line 16 then closes the call to the constructor.

With this translation the generated python code for Listing 6 looks as follows:

```
1  cpachecker = ProgramVerifier(
2      "/.../coveriteam/actors/cpa-seq.yml",
3      "default"
4  )
5  esmbc = ProgramVerifier(
6      "/.../coveriteam/actors/esbmc-kind.yml",
7      "default"
8  )
9  portfolio = Portfolio(
10     [cpachecker,esbmc],
11     "verdict in [RESULT_CLASS_TRUE,RESULT_CLASS_FALSE]"
12 )
```

Listing 8: Translated cvt-code

The first two lines of Listing 6 were translated more or less directly, depending on the given type in the ACTORFACTORY (in this case ProgramVerifier). To execute this PARALLEL PORTFOLIO the execute method needs to be called in the cvt-file, which looks like this:

```
1  result = execute(portfolio, input_parameters);
2  print(result);
```

The "`input_parameters`" are Artifacts (see Sect. 2.2) required for the actors inside the PARALLEL PORTFOLIO. To see the result, the print statement is needed.

With this implementation we are able to create the PARALLEL PORTFOLIO from Example 1. The next section explains the execution of it.

## 3.3 MPI parallel portfolio execution

There are currently two execution modes for the PARALLEL PORTFOLIO. One is using MPI (see Sect. 2.3) and the other one the multiprocessing module of python to create processes. Figure 3.2 shows the spawned processes. Each execution is spawning a process per actor in the PARALLEL PORTFOLIO. For a PARALLEL PORTFOLIO of $n$ actors (2 in Example 1), the MPI execution spawns $n$ MPI Worker processes and the Fallback Execution spawns $n$ PYTHON processes. However the MPI execution has two processes more: The MPI Scheduler and the QueueManager. The next sections describe their functionality.

### 3.3.1 Entry in MPI environment

As mentioned above in Sect. 2.3, to use MPI it needs to be executed with an MPI executable. There must be a call to `mpiexec` (or `mpirun`) and because COVERITEAM is written in python it will look like "`mpiexec python-executable python-file.py`". There are two ways to achieve this, one is to launch the whole COVERITEAM with `mpiexec`, or to spawn a new process, which is using MPI. The first approach is not very practical. First, if one wants to use COVERITEAM, one must have some MPI implementation installed on the system. Otherwise COVERITEAM would not be able to start, because `mpiexec` would be missing. To prevent this we could use a

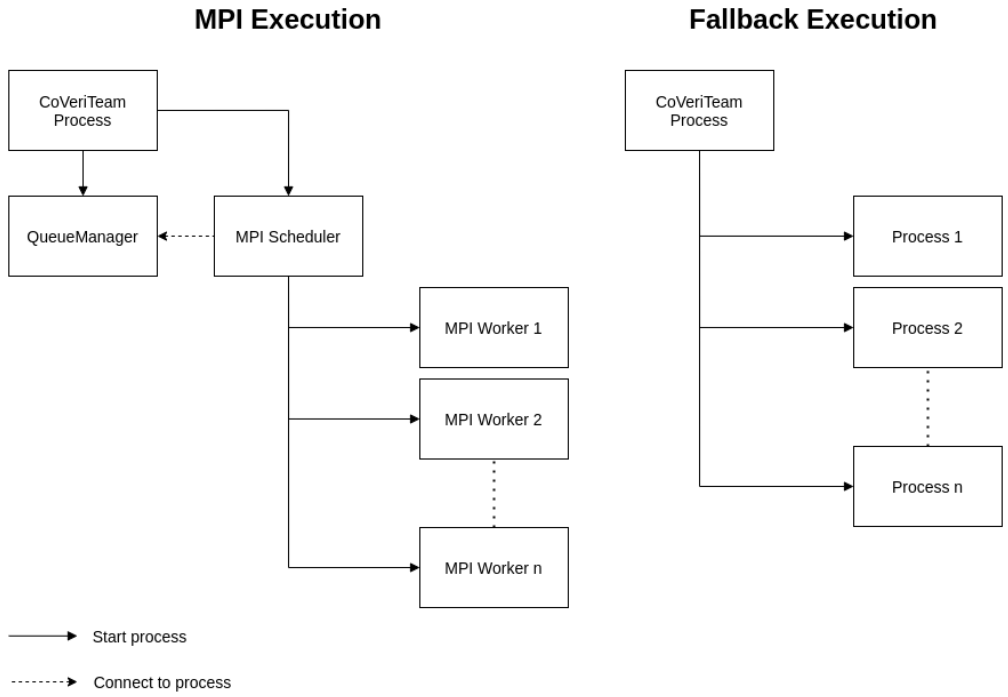



Figure 3.2: Process spawning at PARALLEL PORTFOLIO execution time

check at the beginning and if there is a PARALLEL PORTFOLIO inside the given cvt-file we execute it with MPI. This check could be done automatically or by the user. To rely on the user was no option for us and when done automatically a python process is started, which would check the need of MPI and then start the real COVERITEAM process, which is also not ideal. What remains is to spawn a new subprocess with `mpiexec`. This subprocess then executes the actors of the PARALLEL PORTFOLIO and terminates them after a valid result was produced. This is the purpose of the MPI Scheduler from Fig. 3.2. However because it is a completely new process, it does not have any information about the current state of the PARALLEL PORTFOLIO, which brings us to the next problem and the function of the QueueManager.

### 3.3.2 Process synchronization

The QueueManager is a subclass of python's `BaseManager`-class[2], which is located in the `multiprocessing.managers` package. It creates a TCP server socket and is able to share python objects between multiple processes. To receive an object from a BaseManager, a python process has to create its own instance of BaseManager with the IP-address of the machine, which has created the server at first, and the authentication key, which was used. After the connection it is possible to use the already created shared objects or build new ones. To share data between the COVERITEAM process and the MPI scheduler, only one queue from the `multiprocessing`-package is used. Both processes use this queue. That is a simple way to synchronize them,

[2] `https://docs.python.org/3/library/multiprocessing.html`

because the CoVeriTeam process sends the Parallel Portfolio and the required arguments into the queue and then waits, until a result is put into it. The MPI Scheduler on the other hand waits after start up on this queue until the Parallel Portfolio and its arguments are available. After it has finished, the produced result is put into the queue.

### 3.3.3 Computation

Once the MPI Scheduler has received the Parallel Portfolio and its execution arguments, it creates new instances of MPI Worker for each actor inside the Parallel Portfolio. Then each Worker receives one of those actors with `MPI_Send`. In case of Example 1, one is receiving CPAchecker and the other Esbmc. Subsequently, the arguments are broadcasted to each MPI Worker with `MPI_Broadcast`. Now the worker has everything needed to call the `act`-method of the actor. But first it creates a new thread, which listens to a stop signal from the Scheduler, when a valid result (e.g. it satisfies the success condition) was produced. To receive this stop signal, we used `MPI_Recv` with the MPI Scheduler process ID as source and 99 as its tag. After the thread has started, the actual execution of the actor begins. After the termination of the actor, the result is sent to the Scheduler, which checks if the received result satisfies the success condition. In this case, the stop signal is sent to all running workers and the result is put into the queue. Otherwise, nothing will happen. After every actor has terminated and no result matches the condition, an empty artifact is put into the queue.

**Preventing busy waiting** MPI Implementations are often using busy waiting, when waiting for a message. Thus, the thread, which called `MPI_Recv`, would use the CPU at 100%, checking as fast as possible, if there is a new message with tag 99 available. That is why we switched to a non-blocking receive and manually pausing the thread for half a second. It is a minor detail, to stop as fast a possible, but with the manual pause, the thread is way more efficient in terms of CPU usage and thus energy consumption.

### 3.3.4 Process termination

When a worker receives the `stop` message from the scheduler, it calls the stop-**method** of the actor. This performs a clean shutdown, where all reserved resources are released and the processes are stopped. Before the implementation of the Parallel Portfolio, a shutdown was never required. This feature is introduced with this thesis. There are two categories of shutdowns:

- Composition Shutdown: Each composition must be able to shutdown each of its actors
- Atomic Actor Shutdown: Atomic actors, as an external program, must be able to stop

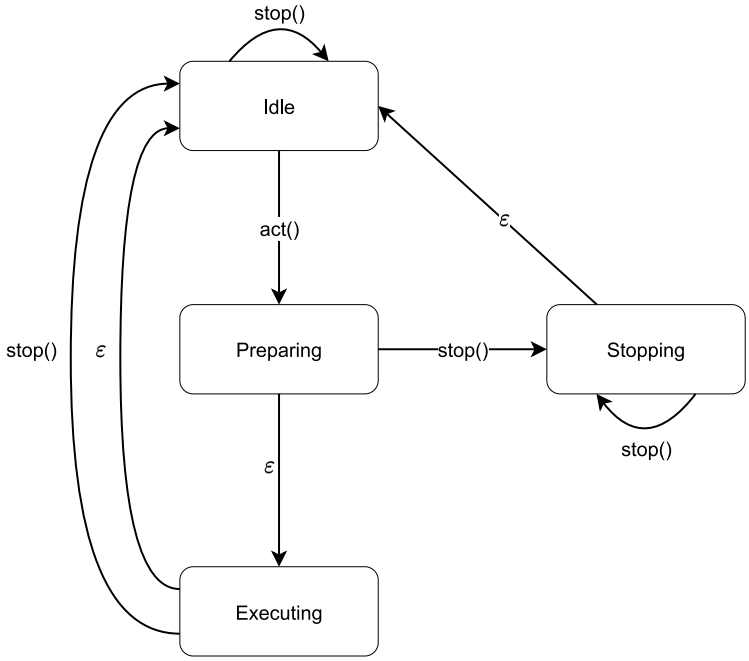



Figure 3.3: States of the atomic actor shutdown

### Compositions

Each of the compositions has its own shutdown algorithm, because each composition treats its actors differently.

**Parallel** Both actors are executed in parallel, thus it is enough to call their stop methods.

**Sequence** Similar to the parallel composition, it is enough to stop both actors. If the first one is running, it will be stopped, the second starts and is immediately stopped afterwards. If the second one is running, the stop call to the first one has no effect and the second one will be stopped.

**ITE** In this case, we have to check, if there is a second actor available. If so, its like in sequence, otherwise the only actor is stopped.

**Repeat** In the Repeat composition, an actor is executed until a certain success condition is reached. Thus only stopping the actor will not be enough, because it may start again immediately. The solution is to set a flag, that the composition has to stop. Now when this flag is set, the success condition will always be true, hence if the actor is stopped, it would not start again.

### Atomic actors

Atomic actors are executed in a RunExec container, which provides a `RunExecutor` object. This object has a `stop`-method, which shutdowns the container and releases all its reserved resources. Therefore, our first approach was to call this method immediately in the MPI Worker after it received the stop signal from the MPI Scheduler. We encountered two problems. First, it was possible that the stop signal arrived before the `RunExecutor` was created. In this case, CoVeriTeam raised an error, because no `RunExecutor` was present. Second, it was also possible that the stop signal arrived during the start-up phase of the actor. In this case the stop signal was ignored. This caused the tool to finish booting and then not interrupt its execution. We solved this by defining four states for the atomic actor. Figure 3.3 shows these states and the behavior of the atomic actor when its `stop`-method gets called. The `act`-method starts the atomic actor. The epsilon transition ($\varepsilon$) in Figure 3.3 denotes the transition into the state after an unspecified amount of time. For example, the atomic actor changes from the Executing state to the Idle state after some time when it is finished. The states are defined as follows:

- `Idle`: The `act`-method of the actor has not been called yet or the last execution has finished. Stop signals are ignored in this state.

- `Preparing`: The atomic actor is in the start-up phase. A call to the `stop`-method of the `RunExecutor` would be ignored. After receiving a stop signal the actor switches to the `Stopping` state.

- `Executing`: The creation of the `RunExecutor` has finished and the tool is running. After receiving a stop signal the `stop`-method of the `RunExecutor` is called. The tools shuts down and the atomic actor switches to state `Idle`.

- `Stopping`: The atomic actor is currently starting and will shutdown as soon as possible. Additional stop signals are ignored. A shutdown is possible after the tool has finished booting up. After the shutdown the atomic actor switches to state `Idle`.

This approach provided a robust shutdown procedure and was able to fix both problems we had with the first approach.

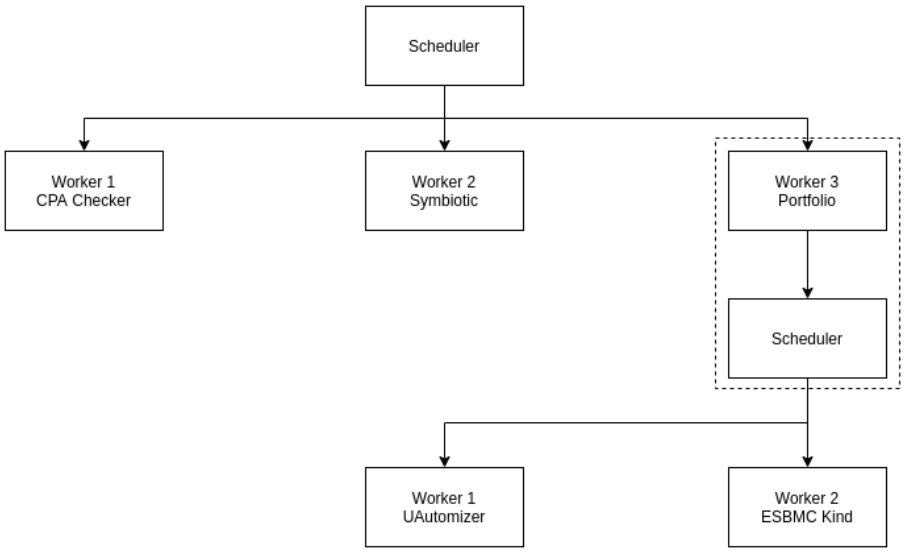



Figure 3.4: Process creation in nested PARALLEL PORTFOLIO

### 3.3.5 Dynamic vs. Static spawning

#### 3.3.5.1 Dynamic spawning

```
1  cpachecker = ActorFactory.create(...);
2  symbiotic = ActorFactory.create(...);
3  uautomizer = ActorFactory.create(...);
4  esbmc_kind = ActorFactory.create(...);
5
6  // Prepare example inputs
7  prog = ArtifactFactory.create(...);
8  spec = ArtifactFactory.create(...);
9
10 inputs = {'program':prog, 'spec':spec};
11
12 result = execute(
13     PORTFOLIO(
14         cpachecker,
15         symbiotic,
16         PORTFOLIO(
17             uautomizer,
18             esbmc_kind
19             )
20         ),
21     inputs
22     );
23
24 print(result);
```

Listing 9: Nested PARALLEL PORTFOLIO in COVERITEAM language

For the process spawning in MPI, we use a method called `MPI_Comm_Spawn`[3]. This method is able to spawn new processes during runtime. We explain it with the help

[3] https://www.open-mpi.org/doc/v4.1/man3/MPI_Comm_spawn.3.php

of the example shown in Listing 9. This example constructs a PARALLEL PORTFOLIO of 2 verifiers and another PARALLEL PORTFOLIO, which also includes 2 verifiers. With the dynamic spawning approach, we get the processes shown in Fig. 3.4 The dashed line indicates one process, which means Worker 3 creates an instance of Scheduler. The Scheduler itself creates two new processes afterwards. With the dynamic spawning, new processes will be created when needed. But there are two downsides, for example, a job scheduling system like SLURM[4] may require some additional configuration, because at schedule time, the total amount of processes is not provided. The second downside we encountered is a problem in some MPI Implementations with `MPI_Comm_Spawn`. When we tested the program on a little PC Cluster, it would not work with OPENMPI version 4.0.3, which was the default released for the OS we used (Ubuntu 20.04). After upgrading to version 4.0.6., it worked perfectly fine.

#### 3.3.5.2 Static spawning

Static spawning requires the computation of all needed processes before the execution of the PARALLEL PORTFOLIO and then spawning them at once. This may be better for job scheduling systems, because the total amount of processes is known before the execution and it is the standard way to spawn processes, thus there may be less errors with MPI Implementations. But it requires advanced configuration of the MPI environment, because now every process has the same world communicator and is able to send messages to other processes, therefore there is a need to create new communicators for each PARALLEL PORTFOLIO in the top one, something which is done automatically with `MPI_Comm_spawn`. But there are more important downsides, described in the following (shortened) cvt-file:

```
actor_1 = ActorFactory.create(...);
actor_2 = ActorFactory.create(...);
actor_3 = ActorFactory.create(...);
actor_4 = ActorFactory.create(...);

portfolio_inputs = {...};

success_condition = ...;

nested_portfolio = ParallelPortfolio(
    actor_3,
    actor_4,
    success_condition
);

sequence_with_first_actor = SEQUENCE(actor_1, nested_portfolio);
sequence_with_second_actor = SEQUENCE(actor_2, nested_portfolio);

portfolio_of_sequences = ParallelPortfolio(
    sequence_with_first_actor,
    sequence_with_second_actor,
    success_condition
);
execute(portfolio_of_sequences, portfolio_inputs);
```

Listing 10: Sequence with PARALLEL PORTFOLIO inside PARALLEL PORTFOLIO

[4] `https://slurm.schedmd.com/documentation.html`

With static spawning, the amount of required processes must be determined. This is only possible by iterating through the actors in the top PARALLEL PORTFOLIO and if it is an AtomicActor (created with ActorFactory) increase the amount of needed processes by one. In case it is a composition, it is important to know, if there is any PARALLEL PORTFOLIO inside this composition, and if so, increase the amount of processes for every AtomicActor and Composition in it. Thus it is like iterating through a tree. This requires complex code changes in the source code of COVERITEAM, because either every composition is able to return their actors or it has a recursive method which returns the count of the needed processes for itself. But this is a deficient approach for scalability, because now the compositions need to have code in it, which is specific for the PARALLEL PORTFOLIO, because each compositon has to search their own actors for PARALLEL PORTFOLIO. However, the worst downside coming with static spawning are potentially never used processes. If you look at the above cvt-file, you may notice, there is a possibility that one nested PARALLEL PORTFOLIO is never used. Given the situation, that `actor_1` is way faster than `actor_2` (which, in COVERITEAM, happens very often). It is very likely to happen, that `actor_1` and the nested PARALLEL PORTFOLIO together are terminating faster than `actor_2`. In this case, the spawned processes for the second nested PARALLEL PORTFOLIO will not be used and therefore are wasting system resources.

### 3.3.6 Conclusion

We think in the context of COVERITEAM, which allows very complex, but also flexible, compositions, dynamic process spawning is preferred. The first problem with the need of new communicators in static spawning is not difficult to solve. There are `graph communicators`[5] in MPI, which would satisfy the needs, but the calculation of needed processes is against the concept of COVERITEAM. In addition, there is the potential waste of system resources. Therefore dynamic spawning is the right solution.

## 3.4 Fallback parallel portfolio execution

In case there is no possibility to install an MPI Implementation or, for some reasons, MPI4PY is not available, we have provided a fallback execution, which is using pythons own `multiprocessing` module. As shown above in Figure 3.2, the structure is less complicated than the MPI one. The fallback execution is used, if no `mpiexec`-executable or no MPI4PY is found. But it is also possible to force the use of this mode by adding `--use-python-process` as a parameter to the command line, when starting COVERITEAM. The execution is very basic, first a queue is created, which is shared between the spawned and the main processes. Every spawned process then executes its own actor, and if a result is produced, it puts it into the queue. The main COVERITEAM process in the meantime is waiting on this queue for the produced results. If it receives a new one, the success condition will be evaluated on it, and if it is true, all processes will be terminated. If no proper result was produced

[5] `https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node171.htm#Node171`

by any of the actors, an empty artifact is returned. Because python uses fork to create those new processes, there is no need to share the PARALLEL PORTFOLIO between them, so the QueueManager is not needed. The big downside to the fallback execution is the missing possibility to run the PARALLEL PORTFOLIO on multiple machines at once.

CHAPTER 4

# Evaluation

In this chapter we will have a look at the performance and possibilities of the PARALLEL PORTFOLIO composition. In order to perform an evaluation as deep as possible we have limited ourselves to software verifiers as tools inside the PARALLEL PORTFOLIO. We therefore used only verification tasks as test data.

## 4.1 Experiment setup

### 4.1.1 Verifier selection

Beyer et al. [4] investigated parts of the PARALLEL PORTFOLIO. They tested different selections of verifiers from the SV-COMP 2021 [1]. The best performing selection for a PARALLEL PORTFOLIO of three verifiers was the selection of CPACHECKER [5], ESBMC [12], and SYMBIOTIC [7]. For the PARALLEL PORTFOLIO of four verifiers, they added ULTIMATE AUTOMIZER [15]. We used the same verifiers for our evaluation, therefore every PARALLEL PORTFOLIO of three verifiers in the following experiments used CPACHECKER, ESBMC, and SYMBIOTIC. With four verifiers we added ULTIMATE AUTOMIZER.

### 4.1.2 Benchmark selection

For the benchmarks we chose the ReachSafety[1] benchmark set. It includes 8883 different programs and their results, when checked against the **unreach call** specification. This specification is a safety property describing that the error location is never reached [4].

[1] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks`

### 4.1.3 Execution environment

The benchmarks were executed on the VERIFIER CLOUD[2]. It uses the following specifications:

- CPU: 3.4 GHz (Intel Xeon E3-1230 v5) with 8 processing units (4 physical + hyper-threading)
- RAM: 33 GB
- OS: Ubuntu 20.04
- MPI-Implementation: OPENMPI[3] 4.0.3

This cloud executes programs inside a BENCHEXEC [6] container. Therefore, we had the possibility to specify the available resources and measure the actual used resources, like CPU time, wall-time and memory usage. We used 15 GB as available memory to get a good comparison of the runs of each single verifier at the SV-COMP 2021 [1]. All PARALLEL PORTFOLIO runs which used the MPI execution were executed with OPENMPI 4.0.3. The only exception is the cluster experiment (see Sect. 4.6).

### 4.1.4 Evaluation criteria

We evaluated the performance of the PARALLEL PORTFOLIO in 5 categories:

- achieved total score
- used CPU time
- used wall-time
- used memory
- used energy

Because we use the first valid result we expected very good results in terms of wall-time. As for CPU time and memory usage higher results were expected. In each row of the following tables, the best value is marked in bold. However, for marking the best value the real numbers are used.

**Score** The score is computed with the same schema as used in SV-COMP [1]. Correct proofs are rewarded with 2 score points, correct alarms with 1 score point, wrong proofs with $-32$ score points, and wrong alarms with $-16$ score points.

[2] https://vcloud.sosy-lab.org/cpachecker/webclient/help/
[3] https://www.open-mpi.org//

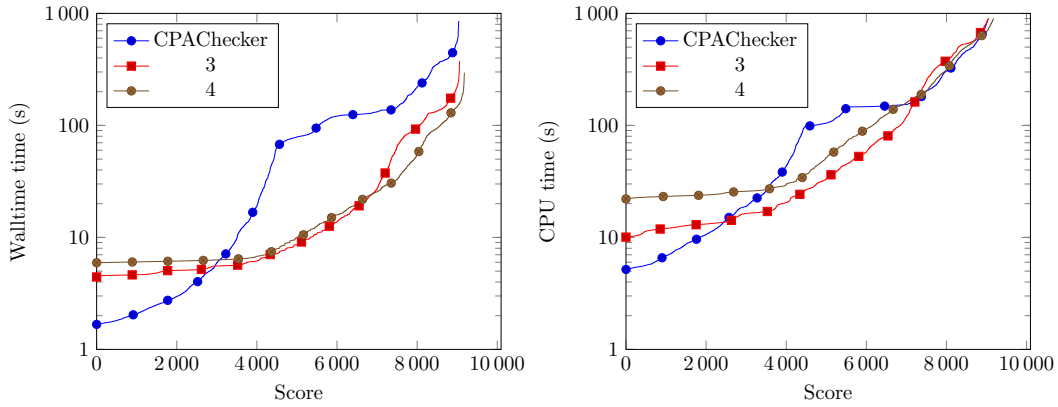



Figure 4.1: Wall-time and CPU time per score point

Table 4.1: Comparison of parallel portfolio of different size with CPAchecker

| Verifier | CPAchecker | Parallel Portfolio of 3 | Parallel Portfolio of 4 |
|---|---|---|---|
| Score | 9 040 | 9 057 | **9 178** |
| Correct results | 5 652 | 6 129 | **6 167** |
| Correct proofs | 3 516 | 3 824 | **3 827** |
| Correct alarms | 2 136 | 2 305 | **2 340** |
| Wrong results | **8** | 35 | 31 |
| Wrong proofs | **0** | 21 | 20 |
| Wrong alarms | **8** | 14 | 11 |
| Total resource consumption for correct results | | | |
| CPU time(h) | 190 | **180** | 220 |
| Wall time (h) | 140 | 48 | **42** |
| Memory (GB) | **7 000** | 11 000 | 12 000 |
| Energy (KJ) | 7 700 | 5 400 | **5 200** |
| Median resource consumption for correct results | | | |
| CPU time(s) | 61 | **26** | 38 |
| Wall time (s) | 32 | **7.1** | 8.0 |
| Memory (MB) | 600 | **530** | 740 |
| Energy (J) | 590 | **200** | 250 |
| Total resource consumption per score point | | | |
| CPU time (s/sp) | 380 | 340 | **340** |
| Wall time (s/sp) | 270 | 100 | **79** |
| Memory (MB/sp) | **3 000** | 3 200 | 3 200 |
| Energy (J/sp) | 4 100 | 2 900 | **2 500** |

## 4.2 General performance

For the general performance we used a PARALLEL PORTFOLIO with 3 and 4 verifiers (For verifier selection see Sect. 4.1.1).

### Resource limitations

- CPU-Time limit: 15 minutes
- Memory limit: 15 GB

The same limits were used for the run with CPACHECKER.

**Interpretation** With the score one disadvantage of the PARALLEL PORTFOLIO reveals. We produce many more correct results than CPACHECKER, but also more wrong results. This was expected, because we use the fastest produced result and we have no way to check, if this result is correct with the standard PARALLEL PORTFOLIO. We tried to minimize the wrong results with a PARALLEL PORTFOLIO, which uses a combination of a verifier and a validator (see Sect. 4.5). But because of the use of the fastest result more verifiers in the PARALLEL PORTFOLIO should increase the score, if we have unlimited resources at our disposal. With limited resources one may think that there is an optimal ratio of verifier amount and resources. Beyer et al. [4] have found a good selection of verifiers for these resource limitations, which we used here also.

As mentioned in Sect. 2.1, the main purpose of the PARALLEL PORTFOLIO is to be fast in terms of wall-time, which is proven by the difference in the wall-time for correct results compared with CPACHECKER. In terms of CPU time we expected the outcome to be much higher, because we use more cores at the same time. The relatively low results can be explained with the low wall-time. The program does not run long enough per experiment to create high CPU times. However the memory usage of the PARALLEL PORTFOLIO is higher than of single verifiers.

The most interesting figure is the energy consumption. As mentioned energy consumption does not scale linearly with other used resources. This is shown by the PARALLEL PORTFOLIO of four actors. Despite the CPU time and the memory usage is the highest for correct results the energy consumption is the lowest.

**Results** The PARALLEL PORTFOLIO is fast in terms of wall-time and surprisingly uses less energy than single verifiers. But it also produces more wrong results. The CPU time is also lower than expected. However the total score is not as high as intended, because the wrong results decrease it by a large amount.

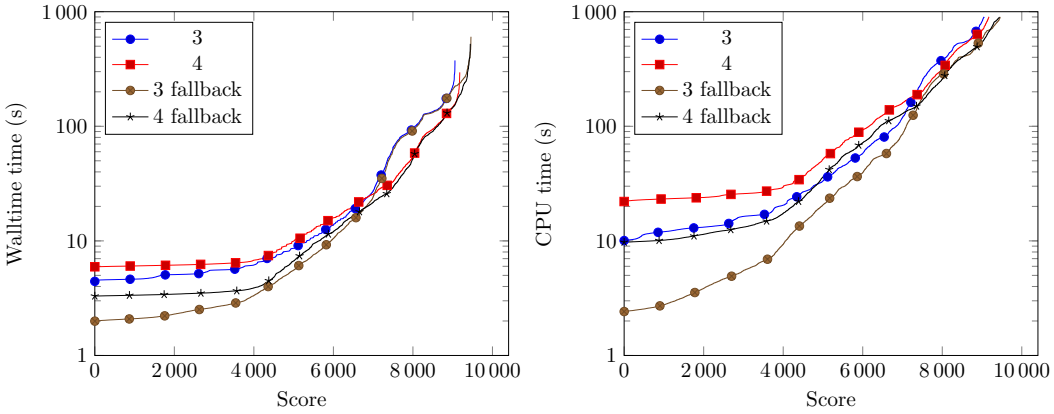



Figure 4.2: Wall-time and CPU time per score point

Table 4.2: Comparison standard execution vs fallback execution

| Verifier | Parallel Portfolio | | | |
|---|---|---|---|---|
| | 3 | 3 fallback | 4 | 4 fallback |
| Score | 9 057 | **9 460** | 9 178 | 9 443 |
| Correct results | 6 129 | **6 364** | 6 167 | 6 363 |
| Correct proofs | 3 824 | **3 992** | 3 827 | 3 944 |
| Correct alarms | 2 305 | 2 372 | 2 340 | **2 419** |
| Wrong results | 35 | 35 | **31** | 33 |
| Wrong proofs | 21 | 21 | **20** | 21 |
| Wrong alarms | 14 | 14 | **11** | 12 |
| Total resource consumption for correct results | | | | |
| CPU time(h) | 180 | **180** | 220 | 200 |
| Wall time (h) | 48 | 61 | **42** | 49 |
| Memory (GB) | **11 000** | **11 000** | 12 000 | 12 000 |
| Energy (KJ) | 5 400 | 5 300 | 5 200 | **5 000** |
| Median resource consumption for correct results | | | | |
| CPU time(s) | 26 | **16** | 38 | 27 |
| Wall time (s) | 7.1 | **4.7** | 8.0 | 5.4 |
| Memory (MB) | 530 | **430** | 740 | 620 |
| Energy (J) | 200 | **120** | 250 | 170 |
| Total resource consumption per score point | | | | |
| CPU time (s/sp) | 340 | **290** | 340 | 290 |
| Wall time (s/sp) | 100 | 130 | **79** | 98 |
| Memory (MB/sp) | 3 200 | 3 000 | 3 200 | **2 900** |
| Energy (J/sp) | 2 900 | 2 600 | 2 500 | **2 200** |

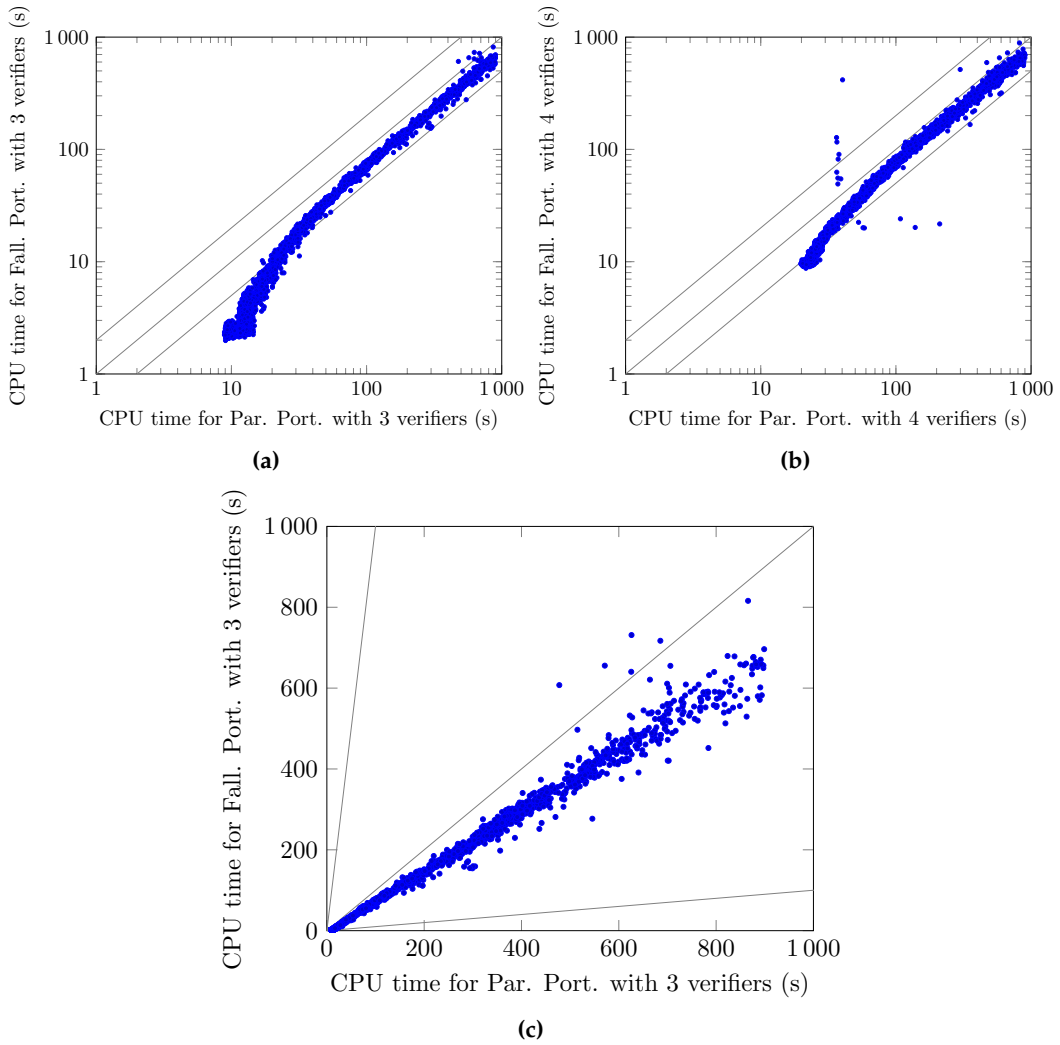



Figure 4.3: Fallback Portfolio vs MPI Portfolio (CPU time)

## 4.3 Fallback vs MPI execution

To measure the performance differences of the fallback and the MPI execution, we ran the same benchmarks as in Sect. 4.2 with the fallback execution.

### Resource limitations

- CPU time limit: 15 minutes
- Memory limit: 15 GB

**Interpretation** The most obvious difference is the higher score of the fallback execution. This is due to the higher amount of correct results, because the wrong result amount is almost the same.

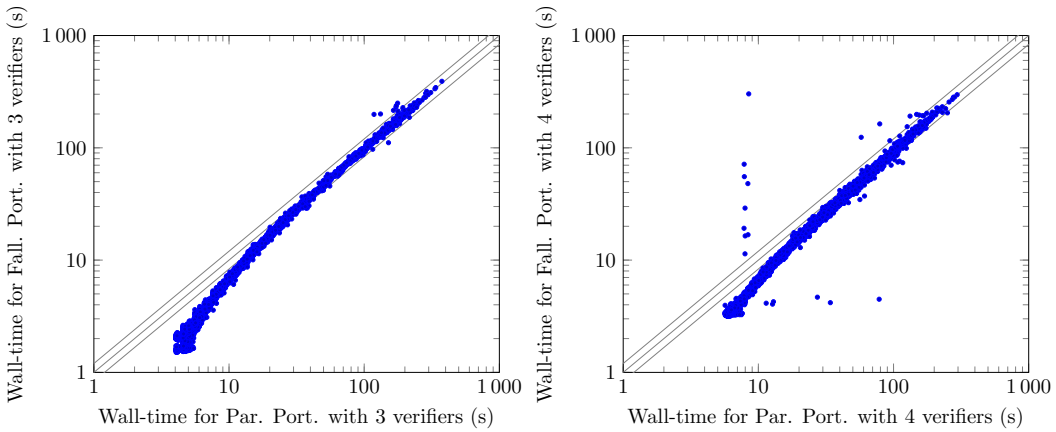



Figure 4.4: Fallback Portfolio vs MPI Portfolio (Wall-time)

Another interesting aspect is, while the total CPU time stays about the same, the increased consumption of total wall time. The explanation for this difference is the in Sect. 3.3.1 described MPI-Scheduler. This process uses the busy waiting strategy [22] for waiting at the next result. Therefore it uses a large amount of CPU time and because we are limiting the CPU time of the benchmarks, the program was stopped earlier. This effect is shown in Figure 4.3 (c). The difference between the CPU time of the MPI execution and the fallback execution is constantly increasing, because the MPI-Scheduler uses CPU time while waiting for new results. Therefore the fallback execution has more total CPU time available for the verifiers. This also explains the higher amount of solved problems.

Also we noticed the lower median wall- and CPU time with the fallback execution. We noticed this behavior earlier during the measuring of wall-time with one single program. Figure 4.2 and Figure 4.4 indicate that the creation of an MPI environment and the spawning of MPI processes take more time than spawning python processes. Both fallback executions need less wall-time for the fast tasks when compared with the MPI execution. For the heavier tasks the difference becomes less significantly.

Another advantage of the fallback execution is the slightly lower energy consumption. However, this mainly results of the more efficient CPU time usage (see 4.3) compared to the MPI execution. The fallback PARALLEL PORTFOLIO uses the same or less amount of CPU time for almost any verification task.

**Results** The fallback execution is faster, more energy efficient and achieves a better score. It only lacks the possibility of using multiple machines to run the PARALLEL PORTFOLIO. But used on a single machine, it is the more efficient execution method.

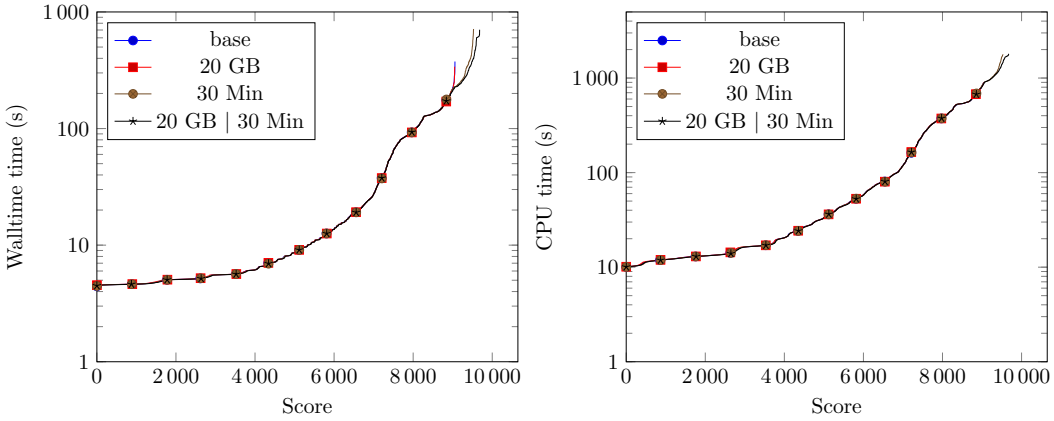



Figure 4.5: Wall-time and CPU time per score point

Table 4.3: Comparison of PARALLEL PORTFOLIO with 3 actors

| Verifier | PARALLEL PORTFOLIO of 3 actors | | | |
|---|---|---|---|---|
| | base | 20 GB | 30 Min | 20 GB \| 30 Min |
| Score | 9 057 | 9 050 | 9 529 | **9 681** |
| Correct results | 6 129 | 6 128 | 6 404 | **6 490** |
| Correct proofs | 3 824 | 3 818 | 4 021 | **4 087** |
| Correct alarms | 2 305 | 2 310 | 2 383 | **2 403** |
| Wrong results | 35 | 35 | 35 | 35 |
| Wrong proofs | 21 | 21 | 21 | 21 |
| Wrong alarms | 14 | 14 | 14 | 14 |
| Total resource consumption for correct results | | | | |
| CPU time(h) | **180** | **180** | 270 | 300 |
| Wall time (h) | 48 | **47** | 73 | 82 |
| Memory (GB) | **11 000** | **11 000** | 12 000 | 14 000 |
| Energy (KJ) | **5 400** | **5 400** | 8 100 | 9 000 |
| Median resource consumption for correct results | | | | |
| CPU time(s) | **26** | **26** | 29 | 30 |
| Wall time (s) | **7.1** | **7.1** | 7.6 | 7.8 |
| Memory (MB) | **530** | **530** | 560 | 580 |
| Energy (J) | **200** | **200** | 220 | 230 |
| Resource consumption per score point | | | | |
| CPU time (s/sp) | **73** | 73 | 100 | 110 |
| Wall time (s/sp) | 19 | **19** | 27 | 30 |
| Memory (MB/sp) | **1 200** | 1 200 | 1 300 | 1 500 |
| Energy (J/sp) | **2 900** | 2 900 | 4 300 | 4 300 |

## 4.4 Variation of memory and time limit

We wanted to know, which of the two limited resources are more important for the PARALLEL PORTFOLIO. In order to investigate this, we have run several different benchmarks. We decided to use the PARALLEL PORTFOLIO with three verifiers for those benchmarks, because with fewer verifiers each of them benefits more from the increase of memory. Thus we expected more significant differences.

#### Resource limitations

- CPU-Time limit: 15 minutes, if not said otherwise above
- Memory limit: 15 GB, if not said otherwise above
- Executed with MPI

**Interpretation** As expected, the PARALLEL PORTFOLIO achieved the highest score, when given the most resources. However, we did not expect the higher importance of the CPU time limit compared to the memory limit. Also the increase of the memory limit while using the same CPU time limit had no effect at all. The seven points difference in the score is due to the fluctuating results. However, if we increased the CPU time limit, increasing the memory limit began to have a noticeable effect.

**Results** The most limiting factor of the PARALLEL PORTFOLIO is CPU time limit. Given that we run several tools in parallel, we need significantly more CPU time than running one single verifier. We used 15 minutes for most of the benchmarks to get a good comparison of single verifiers, because all of those benchmarks were executed with those limits.

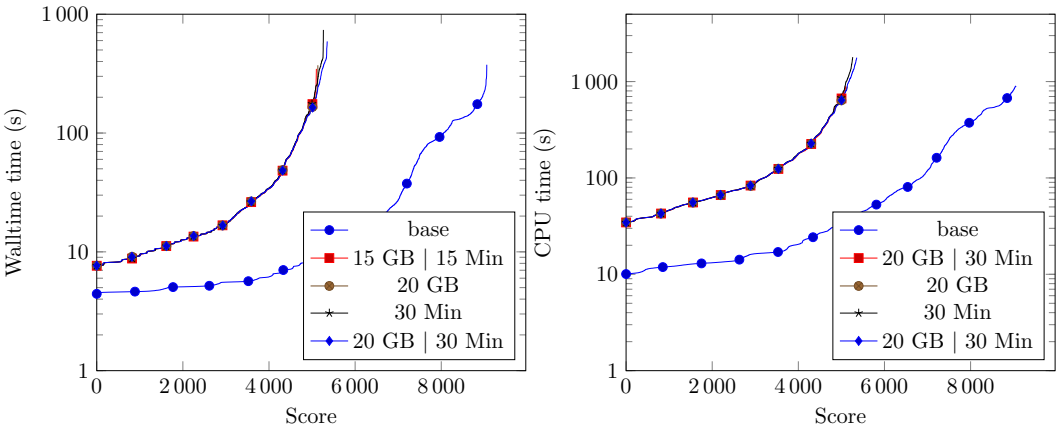



Figure 4.6: Wall-time and CPU time per score point

Table 4.4: Validating PARALLEL PORTFOLIO

| Verifier | Validating PARALLEL PORTFOLIO of 3 actors | | | | |
|---|---|---|---|---|---|
| | base | 15 GB \| 15 min | 20 GB | 30 Min | 20 GB \| 30 Min |
| Score | **9 057** | 5 100 | 5 135 | 5 263 | 5 356 |
| Correct results | **6 129** | 3 612 | 3 641 | 3 747 | 3 807 |
| Correct proofs | **3 824** | 1 552 | 1 558 | 1 580 | 1 613 |
| Correct alarms | **2 305** | 2 060 | 2 083 | 2 167 | 2 194 |
| Wrong results | 35 | **4** | **4** | **4** | **4** |
| Wrong proofs | 21 | **0** | **0** | **0** | **0** |
| Wrong alarms | 14 | **4** | **4** | **4** | **4** |
| Total resource consumption for correct results | | | | | |
| CPU time(h) | 180 | **140** | 140 | 190 | 200 |
| Wall time (h) | 48 | **32** | 33 | 45 | 49 |
| Memory (GB) | 11 000 | **7 900** | 8 500 | 9 200 | 9 900 |
| Energy (KJ) | 5 400 | **3 700** | 3 900 | 5 100 | 5 600 |
| Median resource consumption for correct results | | | | | |
| CPU time(s) | **26** | 75 | 75 | 77 | 78 |
| Wall time (s) | **7.1** | 15 | 15 | 16 | 16 |
| Memory (MB) | **530** | 1 200 | 1 200 | 1 200 | 1 300 |
| Energy (J) | **200** | 500 | 510 | 510 | 520 |
| Resource consumption per score point | | | | | |
| CPU time (s/sp) | **73** | 98 | 100 | 130 | 140 |
| Wall time (s/sp) | **19** | 22 | 23 | 31 | 33 |
| Memory (MB/sp) | **1 200** | 1 500 | 1 700 | 1 700 | 1 800 |
| Energy (J/sp) | **2 900** | 8 300 | 8 300 | 13 000 | 14 000 |

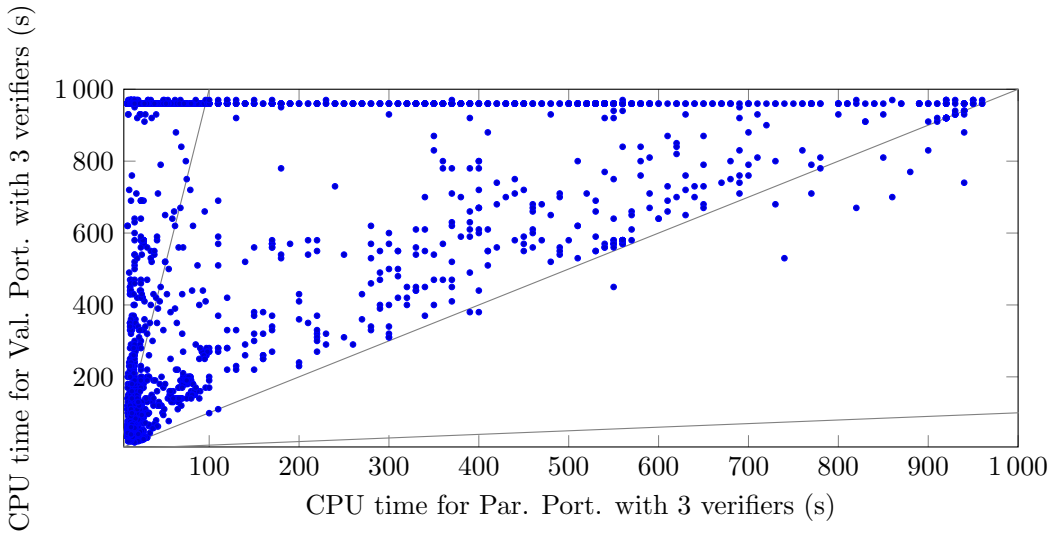



Figure 4.7: Proof tasks solved by the `base` PARALLEL PORTFOLIO

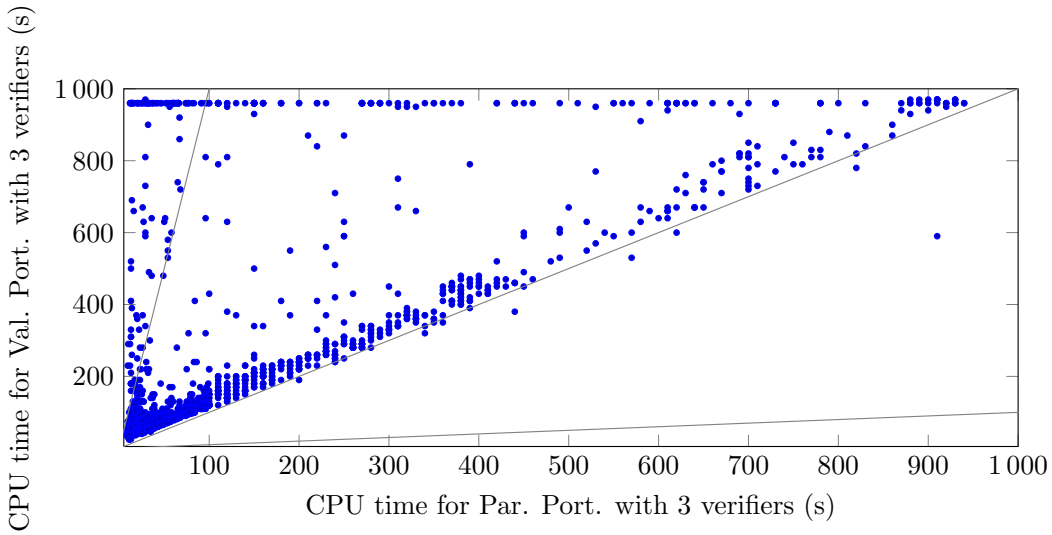



Figure 4.8: Alarms tasks solved by the `base` PARALLEL PORTFOLIO

## 4.5 The validating portfolio

As mentioned in Sect. 4.2, the PARALLEL PORTFOLIO produces a lot of wrong results. We tried to minimize those with this experiment. The validating PARALLEL PORTFOLIO uses a validator after each verifier, which validates the result of the verifier. We expected a significantly lower amount of wrong results and tested different resource limitations. We compared all validating PARALLEL PORTFOLIO experiments with the PARALLEL PORTFOLIO of three verifiers from Sect. 4.2, which is called `base` in Table 4.4.

### Resource limitations

- CPU-Time limit: 15 minutes, if not said otherwise above
- Memory limit: 15 GB, if not said otherwise above
- Executed with MPI

**Interpretation** We have significantly fewer wrong results, but at the expense of performance. The correct results have been almost halved. The remaining 4 wrong alarms are due to the validator. We cross checked each of those 4 tasks with the results of SV-COMP 21 [1] and the validator produced the same wrong alarms for 3 tasks during the competition. No result was produced for the remaining task in this competition, because the witness listing did not succeed.

One indication of those poor outcomes is the median used wall-time for correct results. It is more than doubled compared with the standard PARALLEL PORTFOLIO. It seems, that the validator we used for this benchmarks takes more time on average than the actual verifier. Additionally the used median CPU time is nearly tripled, which can be explained with the still running verifiers during the validation of a result from an already finished verifier. For the same reason the median of the used memory is also higher than normal. All this leads to more than twice the energy consumption. Interestingly, the total resource consumption is lower than that of the standard PARALLEL PORTFOLIO.

Another interesting aspect is the low amount of correct proofs (see Table 4.4). While the amount of correct alarms remains about the same the amount of correct proofs is less than half that of the `base` PARALLEL PORTFOLIO. We created two scatter plots to investigate this. 4.7 shows all correct proof tasks solved by the `base` PARALLEL PORTFOLIO. There are a lot tasks which use very little CPU time but timeout in the validating PARALLEL PORTFOLIO. Figure 4.8 in comparison shows all correct alarm tasks. Here the amount of timeouting tasks is much lower compared with Figure 4.7. Also the used CPU time of the validating PARALLEL PORTFOLIO is much closer to the used CPU time of the `base` PARALLEL PORTFOLIO. The only difference between the `base` and the investigated (15 GB | 15 min) validating PARALLEL PORTFOLIO is the validator after each verifier. It seems that the validator needs significantly more CPU time in case of a proof than in case of an alarm.

**Results** The combination of verifier and validator in the PARALLEL PORTFOLIO is very expensive, but it also reduces the produced wrong results to a minimum. And again the increase of CPU time is more beneficial than the increase of the memory limit.

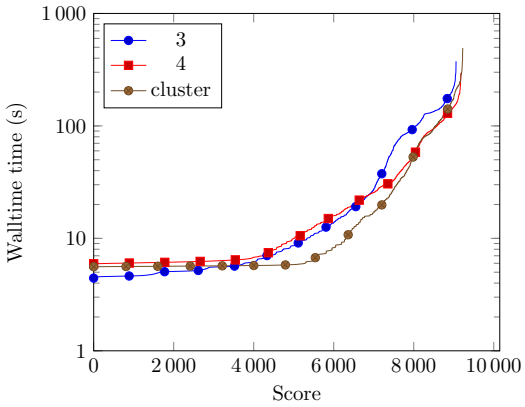



Figure 4.9: Wall-time per score point

Table 4.5: Running PARALLEL PORTFOLIO on a cluster

| Verifier | PARALLEL PORTFOLIO of 3 | 4 | cluster |
|---|---|---|---|
| Score | 9 057 | 9 178 | **9 227** |
| Correct results | 6 129 | 6 167 | **6 681** |
| Correct proofs | 3 824 | 3 827 | **4 114** |
| Correct alarms | 2 305 | 2 340 | **2 567** |
| Wrong results | 35 | **31** | 66 |
| Wrong proofs | 21 | **20** | 32 |
| Wrong alarms | 14 | **11** | 34 |
| Wall time for correct results | | | |
| Wall time (h) | 48 | **42** | **42** |
| Median Wall time (s) | 7.1 | 8.0 | **5.7** |
| Total Wall time (s/sp) | 19 | 16 | **16** |

## 4.6 Cluster run

At last we investigated the performance of the PARALLEL PORTFOLIO when executed on a cluster of multiple PCs. Our test setup consisted of **four** computers with the following specifications:

- CPU: 3.4 GHz (Intel i7-6700) with 8 processing units (4 physical + hyper-threading)
- RAM: 33 GB
- Network: 1 GBit/s Ethernet
- OS: Ubuntu 20.04
- MPI-Implementation: MPICH[4]

[4]https://www.mpich.org/

Each run had limited resources on the host machine. Since we execute the runs with RunExec we were able to limit the resources only on the machine, which starts the run.

#### Resource limitations

- CPU-Time limit: 30 minutes
- Memory limit: 30 GB (10 GB per verifier)
- Executed with MPI

**Verifier selection** As we had significantly more resources at our disposal, we chose all available verifiers from the SV-Comp 2021[5] for the Parallel Portfolio. Each verifier had a total of 10 GB of memory at its disposal and each machine was running 3 of them.

**Interpretation** RunExec is capable of measuring the used resources on one machine only, therefore we dropped the CPU time, memory and energy consumption for this experiment. These values would not have been meaningful.

Since the Parallel Portfolio uses the fastest result, we expected a lower wall-time compared to previous runs. Given a machine with endless resources, the wall-time should always decrease or at least remain the same. Interestingly the total wall-time for correct results was the same compared with the 4 verifier Parallel Portfolio. This can be explained by the lower verifier count on each machine, therefore the comparsion with the 3 verifier Parallel Portfolio is more meaningful here. The median wall-time decreased significantly compared with both Parallel Portfolio, because some of the now included verifiers are much faster in some cases. This is particularly visible in Figure 4.9

Despite being faster, the score of the experiment was disappointing. Although it solved a lot more verification tasks, it also produced almost double amount of wrong results. It even falls behind the Parallel Portfolio executed with the fallback execution (see Sect. 4.3).

**Results** To quadruple the available resources has only led to faster wall-time and a slight increase of score. The experiment solved the most problems, but also produced the most errors. Therefore the results seem to be disappointing. But because we used the best combination of verifiers with the previous experiments, there was little room for improvement. Our selected verifiers (see [4] for more information) were chosen to complement each other. This selection took a lot of time. If this time is not available, a kind of brute force approach is possible. In this case, all verifiers are selected. The Parallel Portfolio produces promising results even with this approach. A remaining disadvantage is the high number of false results. To

[5] https://sv-comp.sosy-lab.org/2021/

minimize this number, the validating PARALLEL PORTFOLIO remains to be tested on a cluster (see Sect. 5.2).

## 4.7 Reflection

**The use of MPI** At the start of this work we chose MPI as framework to spawn processes as it provides an easy way to exchange messages between those processes. These new processes should execute the tools of COVERITEAM and send the results back to one specific process. For this result sending we needed the exchange of messages. Additionally MPI allows the execution on PC Cluster, which was another objective for this work. However to use MPI we needed some sort of bridge between the MPI functions interface, which is only available in C, C++, and Fortran. We chose MPI4PY ( [8] [9] [10]) as bridge. Furthermore, we needed an MPI implementation to execute the program. This implementation and MPI4PY have to be installed on the system and are difficult to ship with COVERITEAM. Therefore, we chose to implement the fallback execution which only uses features integrated in python. We then discovered the increased performance of the fallback execution compared to the MPI execution on a single machine described in Sect. 4.3. One could think that the MPI execution is obsolete now, but in our evaluation we used an already optimized selection of verifiers (see Sect. 4.1.1 and [4]). Without an optimized selection one might want to execute a lot of tools in parallel on multiple machines to get many result with a minimum of preparation time. However, the MPI execution should not be used, when the PARALLEL PORTFOLIO is executed on a single machine.

CHAPTER 5

# Conclusion

## 5.1 Conclusion

We were able to achieve the main goal. The PARALLEL PORTFOLIO provides very short wall-times and outperforms each single standalone tool within the composition. Furthermore, despite using more CPU time and memory it saves a significant amount of energy, when compared with single tools. However, its high amount of wrong results makes its usefulness difficult in some situations. In total the PARALLEL PORTFOLIO produces results up to 3 times faster than single verifiers while its energy consumption is 30-60 % lower. The only downsides are more wrong results and more required CPU cores to work properly. Therefore we think, if a result is required fast, the PARALLEL PORTFOLIO is always the preferred choice. Also the fallback execution, which was designed to simplify the use of the PARALLEL PORTFOLIO due to the omitted need of MPI, was a success. It performed even better than the standard execution, but it obviously lacks the ability to run on a cluster. It should be noted, that the performance of the PARALLEL PORTFOLIO strongly depends on the selected tools. It is up to the user to select the best tools for their tasks. If it is not possible to select the best tools the user has the ability to use an arbitrary amount of tools inside the PARALLEL PORTFOLIO and execute it on a cluster.

We hope our PARALLEL PORTFOLIO will increase the productivity in the field of software verification through faster wall-times, while also helping to protect the planet, because of the lower energy consumption.

## 5.2 Future Work

We tested the performance of the PARALLEL PORTFOLIO on a cluster of multiple machines. The performance was disappointing, mainly due to many wrong results. We also tested a composition called validating PARALLEL PORTFOLIO. In this composition

each verifier result was checked by a validator. This composition also performed poorly due to the limited amount of resources. We will investigate different validating PARALLEL PORTFOLIO and their execution on clusters in the future. We hope to achieve a high amount of correct results with little to no amount of wrong results in reasonable wall-time.

# Bibliography


[1] D. Beyer. Software verification: 10th comparative evaluation (SV-COMP 2021). In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–422. Springer International Publishing, 2021.

[2] D. Beyer. Progress on software verification: SV-COMP 2022. In *Proc. TACAS (2)*, LNCS 13244. Springer, 2022.

[3] D. Beyer and S. Kanav. CoVeriTeam: On-demand composition of cooperative verification systems. In *Proc. TACAS*. Springer, 2022.

[4] D. Beyer, S. Kanav, and C. Richter. Construction of Verifier Combinations Based on Off-the-Shelf Verifiers. In *Proc. FASE*. Springer, 2022.

[5] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.

[6] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, Nov. 2017.

[7] M. Chalupa, T. Jašek, J. Novák, A. Řechtáčková, V. Šoková, and J. Strejček. Symbiotic 8: Beyond symbolic execution (competition contribution). In *Proc. TACAS (2)*, LNCS 12652, pages 453–457. Springer, 2021.

[8] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011. New Computational Methods and Software Tools.

[9] L. Dalcín, R. Paz, and M. Storti. Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[10] L. Dalcín, R. Paz, M. Storti, and J. D'Elía. Mpi for python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655–662, 2008.

[11] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. MIT Press, 1990.

[12] M. Y. Gadelha, H. I. Ismail, and L. C. Cordeiro. Handling loops in bounded model checking of C programs via *k*-induction. *Int. J. Softw. Tools Technol. Transf.*, 19(1):97–114, February 2017.

[13] M. Y. R. Gadelha, F. R. Monteiro, L. C. Cordeiro, and D. A. Nicole. ESBMC v6.0: Verifying C programs using *k*-induction and invariant inference (competition contribution). In *Proc. TACAS (3)*, LNCS 11429, pages 209–213. Springer, 2019.

[14] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2*. The MIT Press, 1999.

[15] M. Heizmann, Y.-F. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski. ULTIMATE AUTOMIZER and the search for perfect interpolants (competition contribution). In *Proc. TACAS (2)*, LNCS 10806, pages 447–451. Springer, 2018.

[16] M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *Proc. CAV*, LNCS 8044, pages 36–52. Springer, 2013.

[17] B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

[18] J. W. Klop. Term Rewriting Systems: From Church-Rosser to Knuth-Bendix and Beyond. In *ICALP*, pages 350–369, 1990.

[19] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[20] D. A. Plaisted and Y. Zhu. Equational Reasoning using AC Constraints. In *In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97*, pages 108–113, 1997.

[21] Wikibooks. Message-passing interface — wikibooks, the free textbook project, 2020. [Online; accessed 30-August-2021].

[22] Wikipedia contributors. Busy waiting — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Busy_waiting&oldid=1043540640`, 2021. [Online; accessed 20-January-2022].

[23] Wikipedia contributors. Portfolio (finance) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Portfolio_(finance)&oldid=1041339985`, 2021. [Online; accessed 25-November-2021].