International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

Parallel Execution of Programs as a Support for Mutation Testing: a Replication Study

Stevão A. Andrade, Simone R. S. de Souza, Paulo S. L. de Souza, Márcio E. Delamaro
Departamento de Sistemas de Computação
Universidade de São Paulo
São Carlos, São Paulo, Brasil
{*stevao,srocio,pssouza,delamaro*}*@icmc.usp.br*

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Mutation testing is well known as one of the most effective approaches to create test cases, which can detect software faults. However, its drawback is the low scalability – if no special attention is given to improve efficiency – that directly affects its application in practice. This paper shows a replication study focused on emphasizing evidence in which the use of distributed processing structures can improve mutation testing. For this purpose, an architecture that enables mutation testing concurrent execution was designed. Five load balancing algorithms responsible for controlling the distribution and execution of data while carrying out mutation testing were evaluated. Experiments were conducted in order to evaluate the scalability and performance of the architecture considering homogeneous and heterogeneous setups. A time reduction of 50% was observed when executing mutants in parallel in relation to the conventional sequential application of mutation testing. The performance gain was above 95% when there was a higher number of nodes in the distributed architecture.

Keywords: Software Testing; Mutation Testing; Parallel Computing; Experimental Replication.

1. Introduction

Mutation testing emerged aiming at evaluating the effectiveness/quality of a test case set. For this purpose, researchers use the information on the typical mistakes made by developers during coding software processes.

The idea is to create a test set that can show the anomalous behavior of mutant programs comparing them to the original program. Thus, mutant programs are created aiming to model the most frequent mistakes that developers tend to make during the software development process. The aim is to guide testers to produce test cases, which show that faults modeled in mutants are not present in the evaluated product.

The general process involved in mutation testing consists of four steps: *(i)* creating mutant programs; *(ii)* executing the program being tested; *(iii)* executing the mutants; and *(iv)* analyzing the mutants.

Mutants that present the same (correct) behavior as the original program being tested are called live mutants. The others are dead mutants. Among the live mutants, there are those which cannot be killed by any test case because they always produce the same results as the original program. They are known as equivalent mutants. A mutation score gives a measure of the test adequacy and is computed by dividing the number of dead mutants by the total number of non-equivalent mutants.

Apart from being an efficient technique to identify real faults, Just et al. [1], mutation testing has been widely used aiming at evaluating the efficiency of other validation techniques to detect faults. Therefore, mutation testing is used to generate versions of programs containing faults. Then, experiments are conducted to evaluate how much a technique or testing criterion can detect faults modeled by mutation operators. The advantage of using mutation testing in this context is that this approach presents a well-defined process to seed the faults in the product to be evaluated. According to Andrews et al. [2] mutation testing makes it easier to evaluate the effectiveness of techniques or criteria in other experiments.

One of the problems of applying mutation testing is the high computational cost involved in its application as the number of mutants generated can be large, even for small-sized programs. Budd [3] shows that the number of mutants is usually related to the number of variables in a program. This cost is related to the time required to execute the mutants and identify equivalent mutants. The latter is normally done manually or using highly expensive heuristic techniques, which also require executing a high number of mutants [4].

Approaches, such as the automatic test data generation, allow extensive verification of systems. When applied to mutation testing, the generated mutants have to be executed multiple times with a high number of test cases in order to verify, for instance, the efficacy of a given test set to kill a specific group of mutants. Therefore, approaches to increase efficiency in the application of mutation tests are even more important.

Besides, approaches such as experimental evaluations, where the whole set of mutants need to be executed numerous times with a large data set in order to collect data regarding the process, reinforce the need for an approach to speed up the implementation of mutation testing. Thus, interest in approaches with this purpose has become even more crucial.

The present paper proposes to evaluate the application of parallel mutation testing, replicating the study carried out by Reales and Polo [5] in a different setting. The purpose of this replication is to verify the results of the original experiment, and, in the event of inconsistencies, to identify which factors or parameters explain the differences. This paper evaluates five load balancing algorithms using 40 test subjects written in C language with different characteristics such as the domain, size, and the number of test cases. Moreover, different tools from the original paper were used to generate and run mutants and a distributed architecture was developed to accommodate such tools in a parallel execution environment.

In order to consolidate the knowledge produced with experimentation, the results need to be checked thoroughly. This verification is performed using replications. The replication can help to increase the confidence in which the results were not generated due to a specific configuration used in the original experiment. This approach contributes to mitigating threats to the validity of the results obtained in both experiments [6].

Another key point to increase the confidence of the results is to make the data available used in the experiments. This can encourage other researchers who are interested in replicating the experiment. According to Shull et al. [7], replication is useful because in an experiment researchers can make mistakes while using the methodology or even when extracting/analyzing the results. The results of an experiment can be influenced by several factors, such as the tools used, programming language, network configuration, execution environment settings used to perform the experiment, and others. Thus, sharing such data helps avoid these kinds of mistakes and helps to boost confidence in the results produced.

In summary, the main contributions of this article include:

- an evaluation of the performance to understand how many parallel computation techniques can improve the applicability of mutation testing in C;
- an empirical evaluation concerning the behavior of the load balancing algorithms; and
- – a comparison with the results previously presented in the literature, particularly in the study by Reales and Polo [5].

Our results can also be used to reinforce the importance of parallel computing to make the mutation test viable under the perspective of scalability. Besides, it gives some directions to make mutation testing more efficient in an architecture that is simple and accessible. It does not require special hardware or expensive settings. It uses a flexible structure that can be used with a few or a large number of nodes and is available in most academic or industrial environments.

On the other hand, it is important to note that no single experiment is sufficient to give a definitive answer to such a problem [8]. On the contrary, replication is necessary to corroborate (or not) the results of experimental work.

The remainder of this paper is organized as follows: Section 2 introduces a review of fundamental concepts related to this study; the next Section 3 discusses related work. Section 4 presents the architecture and the load balancing algorithms studied in this paper. The experimental setup used to evaluate the architecture and the algorithms are described in Section 5. Section 6 presents the results and discussions of the experiment. Section 7 discusses the threats to validity of the experiment. Section 8 draws the conclusions and the future work to be carried out based on this study.

2. Theoretical Foundations

In this section, we present a review of fundamental concepts related to this study. The main concepts related to mutation testing and concurrent programming are explained. Finally, using tools to support mutation testing is discussed.

2.1. *Mutation Testing*

According to DeMillo et al. [9, 10], mutation testing aims to measure the level of suitability of a test set. To achieve this purpose, mutation testing uses information on typical mistakes that may happen during software development processes.

Considering a program under testing P, faults are purposely inserted using mutation operators [11, 12, 13, 14], generating alternative versions of the program called mutants $(M_1, M_2, ..M_n)$. These syntactic changes represent faults that should be detected while running the tests. Therefore, test cases are designed to prove that the faults modeled in the mutants are not present in the original program.

Offutt [15] states that the activity of creating test cases to distinguish mutants is highly effective in revealing real faults due to the coupling effect. Thus, test cases that reveal small faults are more likely to detect other types of faults [1].

After ensuring that program P complies with the expected outputs for a given test set T, the same test set T is executed against the set of generated mutants. If a mutant M shows a different output from the original program P with some test case in T, it is said to be *dead*. Since it has been distinguished from the original one, it no longer needs to be considered. If, on the other hand, a mutant M produces the same output, for all test cases, as the original program P, then it is said to be *alive*. In cases where a test case cannot be identified to distinguish M_i from P, then M_i is said to be *equivalent* to P.

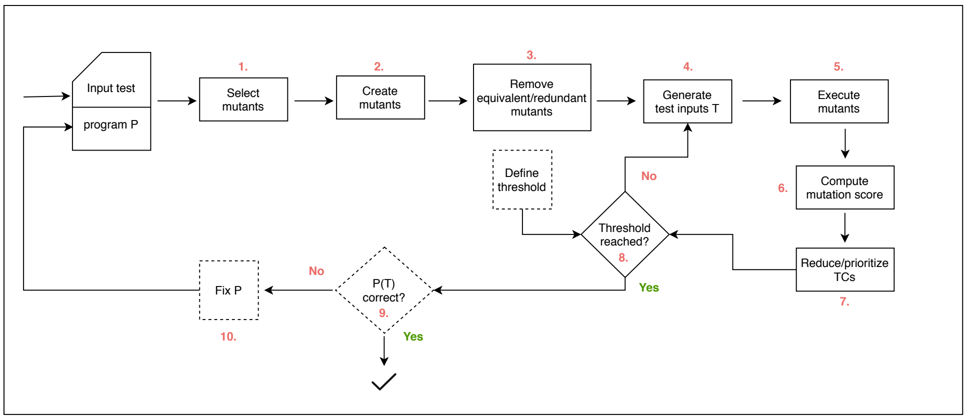


Fig. 1: Modern mutation testing process.

Over the years, the mutation testing process has evolved to consider other aspects, such as selecting a subgroup of mutants, removing redundant mutants, prioritizing test cases, etc. Figure 1 presents an updated version, proposed by Papadakis et al. [16], of the mutation test process, that was originally proposed by Offutt and Untch [17]. The new process takes into account the new research that has been developed in this field.

One of the major problems of applying mutation testing is the high computational cost involved in executing it as the number of mutants generated can be large, even for small-sized programs. This cost is related to the time required to execute the mutants, which is the main reason for developing this research.

2.2. *Mutation Testing Tools*

The use of the mutation test can be a difficult task without the support of a tool since it involves a wide range of complex steps during the process. Delamaro et al. [18] point out that various tools were developed as this technique started to be used until recent days, period in which the interest in mutation testing has grown. Papadakis et al. [16] summarize an extensive list with 87 tools that implement the mutation test criteria for a wide variety of programming languages, ranging from legacy languages like Fortran to more modern languages, like Python.

However, it is still possible to observe that *C* and *Java* languages received more attention from researchers with a larger number of tools to support the application of the criterion (according to the classification described in [16], respectively 13 and 17 tools). Among them, this paper highlights the *Program Testing Using Mutants* (*Proteum*) that support the application of mutation testing for programs developed in *C*.

The main characteristics of *Proteum* are highlighted as follows. It should be mentioned that the following features made us choose this tool to carry out our research. Moreover, it is also widely used in academia:

- it uses a compilation approach instead of an interpreted one;
- it uses control flow information to avoid unnecessary execution of certain mutants;
- it is easy to import test sets to the tool and it allows operations to enable/disable test cases and mutants in a flexible way; and
- there are two modes to execute mutants: testing and research. The first presents the conventional behavior of the mutation testing, interrupting the execution of a mutant when a test case distinguishes the mutant from the original program. In the "*research mode*", all the test cases are applied against all the mutants.

Using the "*research mode*" to run a test session in *Proteum* has proven to be very useful for carrying out experimental studies. On one hand, more accurate data can be collected regarding test sets and mutant sets but, on the other hand, it increases the cost of mutation testing. Therefore, this paper is also one way to support complex studies that require a large number of mutant executions.

2.3. *Parallel Programming*

Over the years, the result of technological advances in the hardware area has caused distributed computing to increasingly form part of the current systems due to the high market demand for high-performance computers.

One of the major challenges was to enable computational solutions capable of reducing processing time and capable of providing even more accurate answers. One of the major areas that are dedicated to proposing such improvements is parallel and high-performance computing, which allows the proposition of techniques that optimize the processing time, making it possible to perform tasks previously considered unfeasible or even that would take a long time to complete.

Grama [19] shows that this development paradigm is essential for building applications capable of reducing the execution time and it has been successfully applied in various fields such as weather forecasting, fluid dynamics, image processing, quantum chemistry, etc.

The basic idea behind parallel programming is, that certain parts of computing are sub-divided into small units so that each unit is responsible for solving one part of a bigger problem. According to Pacheco [20], this approach makes it possible to solve problems in which better use of the available hardware resources is required, for instance, when there is a need to process a large amount of data.

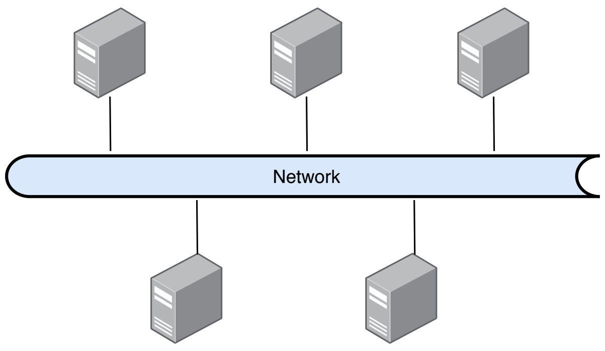


Fig. 2: Parallel architecture - distributed memory paradigm

The approach adopted in this work uses a distributed memory paradigm [21], in which each processing unit has its own memories and processors, which are connected through a network bus, responsible for enabling the exchange of information. Figure 2 presents the communication model adopted, among the main advantages of this approach is the ability to expand or reduce as many processing units as necessary, in order to scale your computer resources according to the problem to be handled.

The models that explore computational distributed memory environments adopt the message-passing paradigm. Such models are composed of communication and task synchronization routines and depend on the technology used to guarantee such characteristics.

3. Related Work

This section presents a list of related studies that support and guide the development of this paper.

Harrold [22] says that although mutation testing can be considered a good criterion to detect faults, there is still no consensus regarding an approach to solving the problem related to the cost of applying mutation testing. Thus, some approaches have been proposed, each one exploring a specific characteristic of the problem.

Offutt and Untch [23] proposed a classification in three types of approaches developed to reduce the cost of the criterion: *(i) do fewer* - try to find ways to reduce the mutants that need to be generated without compromising the applicability of the criterion, *(ii) do faster* - search for ways to speed up the execution of mutants and *(iii) do smarter* - divide the computational cost, preventing the full execution of mutants or store information about the state of the mutants to enable intelligent executions. After that, Jia and Harman [24] proposed a classification that contains only two categories: (a) reduction of the generated mutants (which is equivalent to the “do fewer” strategy) and (b) reduction of the execution cost (which combines two strategies, “do starter” and “do faster”).

The use of advanced platforms to support the mutation testing application is classified by Jia and Harman [24] as a reduction of the execution cost approach. There are various studies in the literature that deal with parallel mutant execution. Two main lines of investigation can be highlighted regarding parallel mutant execution: mutant execution using single instruction multiple data machines (SIMD) and multiple instruction multiple data machines (MIMD) [25].

Some papers that use concurrent execution of mutants were published a few decades ago. Although interesting in theory, they have proved useless in practice, mainly because they require dedicated high-cost architectures. Only recently, with the availability of relatively low-cost networks and the possibility to distribute workload to the nodes of such architecture, the use of parallel computing has become attractive to mutation testing.

In the following, we describe the main works related to the practice of parallelizing the execution of mutation testing. The resume is based on the discussion presented in the systematic literature review conducted by Pizzoleto et al. [26].

Mathur and Krauser [27] proposed to execute various mutants in a vector machine. Specifically, the algorithms described in the study were designed to execute mutants obtained from the Scalar Variable Replacement (SVR) mutation operator since this operator generates a large number of mutants. Afterward, they extended the idea to use the application for other mutation operators [28] and the experimental results obtained showed that, at that moment, the biggest drawback to applying the technique would be the problems related to the time spent on compiling the mutants.

The second line of research was based on MIMD machines. The first study on mutation applied to MIMD machines was proposed by Choi and Mathur [29]. The authors presented an adaptation of a tool called *Mothra*, dividing it into modules, to make it possible to apply mutation testing in parallel. In that study, it was observed that the time of communication between the parallel modules of the tool had a great impact on the applicability of the technique.

Another important study carried out by Offutt et al. [30], presented another variation of the *Mothra* tool, but this time making it possible to execute mutants in *Hypercube* machines. In this new approach, the authors analyzed three different algorithms that distribute static data, which separated the set of mutants: in their original order of generation, distributed randomly, and finally, the uniform distribution of mutants, taking into consideration the mutant operators.

The key point that can justify the low use of these approaches is the fact of the application being limited to the use of complex and expensive architectures, which are not easy to access. However, due to the cheapening of personal computers and the development of local networks, distributed systems became more popular and opened new possibilities to solve problems that require high processing power. Then, recently, new works started to investigate ways to improve the applicability of the mutation test.

Noteworthy are the works proposed by Saleh and Nagi [31], Cañizares [32] and Reales and Polo [5]. In the first one, the authors proposed an approach that uses *MapReduce* programming model to accelerate the generation and execution of mutants. In order to allow the execution using *MapReduce* model, the authors proposed a tool called *HadoopMutator*, which uses a mutation testing tool for the generation of mutants, while their execution is totally built on top of *Hadoop*. The results show that the performance can be enhanced 10 folds, on average, in the subjects analyzed. This approach tends to follow a static schema in which the inclusion of other dynamic distribution algorithms is not considered. Besides the framework does not allow the use of heterogeneous and dynamic environments.

Cañizares and Mercedes [32] proposed *EMINENT*, a dynamic distributed algorithm that uses a *test-level grain*, in which the execution blocks are composed by a single test case, in contrast to *mutant-level grain*, where the execution blocks are formed by a complete set of test cases. The paper discusses that using an approach that defines the granularity based on test cases tends to be more effective regarding approaches that use mutants. The experimental evaluation showed that the proposed algorithm provides high efficiency when compared to the sequential execution. However, the experiments do not make clear how the mutation testing is performed. The algorithm presented in the paper only discusses how the distribution of the data among the machines is performed. Details such as how the approach manages the communication of workers with the mutation testing tool to prevent multiple kills, or if the approach also handles these steps, and which mutation operators were used in the experiment are unclear.

Cañizares and Mercedes also present a comparative study between their proposed algorithm and the results presented by Reales and Polo [5]. Despite the claim that their approach achieves better results than the approach proposed by Reales and Polo, the scenarios used to make such comparison are not described, which tends to hinder the adoption of the approach for comparison purposes.

Reales and Polo [5] employed parallel execution as a way of improving the applicability of mutation testing. In that study, five load balancing algorithms are addressed and the main goal is to split the execution of activities carried out in mutation testing to improve efficiency when applying the technique, reducing the execution time and maximizing the use of computational resources.

That paper presents the evaluation of different load balancing algorithms to enable the application of mutation testing in parallel. The authors evaluated two static approaches (DMBO, DTC), two dynamic approaches (GMOD, GTCOD), and a hybrid approach (PEDRO) that combines both static and dynamic characteristics. The algorithms were evaluated in the context of the mutation testing applied to programs developed in Java. To perform the evaluation, a set of four programs was used. They were analyzed in different environments, changing configurations, like the number of machines used in the parallel test session.

The results showed PEDRO algorithm as the best approach to be used, justified by the fact that the algorithm merges the main features of static and dynamic data distribution approaches. In this way, the algorithm tries to avoid idle machines in the execution and at the same time tries to ensure that there is no communication overhead. In addition, the results showed that after PEDRO, the best algorithms were GMOD, DMBO, DTC, and GTCOD, in this order. The paper does not compare the results of the parallel execution with a sequential version, leaving doubts about how significant the performance gain is.

Thus, the empirical evaluation presented in the current paper works as a complementary study to the work developed by Reales and Polo [5]. The focus of this paper is to compare one more time the algorithms proposed in the work of Reales and Polo, applied in a different scenario. First, the C language was the target of the study, second, a different mutation testing tool was used and third, a large number of subject programs were explored. We also tried to fill the gap existing in the first evaluation by comparing the performance of the sequential version of the mutation testing to the parallel version. The results obtained are compared with the results obtained by Reales and Polo to foster discussions on how the approaches analyzed behaved in different contexts.

4. Load balance algorithms and Architecture

This section presents the algorithms evaluated in this paper, as well as the architecture proposed to provide support for executing the algorithms with the mutation testing tool used in the experiments.

The approach explored in this section benefited from the great progress of today's computer processing capability. This enables us to investigate ways of improving the execution of mutation testing, using parallel computing techniques together with load balancing algorithms to improve its efficiency.

To address this idea, we developed an architecture that allows the application of mutation testing in parallel using the *Proteum* tool as support. Five load balancing algorithms were used to control the data distribution and the programs during the experiment. Among them, two algorithms use a static data distribution approach. The other two adopt a dynamic distribution approach and one uses a mixed approach.

4.1. *Load Balance Algorithms*

According to Grama [19], a load balancing algorithm divides the processing of a task in P parts, where P is the number of machines available to process the existing demand. The idea behind a parallel algorithm is to divide the execution of a task between processors or machines to reduce the processing idleness and increase its execution efficiency.

This sub-section addresses five load balancing algorithms that divide the execution of activities carried out in mutation testing among various machines to improve the efficiency of applying the technique, reducing the execution time, and maximizing the use of computational resources. These algorithms were initially proposed by various authors in the literature and then adapted to the context of mutation testing by Reales and Polo [5].

The five algorithms evaluated in this paper are:

- **Distribute Mutants Between Operators (DMBO)**

  First proposed by Offutt et al. [30], DMBO is a static algorithm that distributes the set of mutants in various parts. It uses the mutation operators adopted as a criterion to divide the mutants. This algorithm consists of a static approach to make the division of the tasks. Given a set of machines M and a set of mutants M_{op} generated by the mutation operator *op*, each part contains a total of $|M_{op}|/|M|$ mutants.

  Each processor machine receives its workload only during the process of generating mutants. Due to the nature of each mutant and test case set used, there is no guarantee that the workers will stop their activities at the same time, which means that some workers may stop their activities before others and the total execution time is defined by the machine with the worst performance;

Algorithm 1 Distribute Mutants Between Operators

```
Input: MutantsDictionary: $ms$, TestSet: $ts$, WorkersList: $workers$
$p \leftarrow$ size($workers$)          // returns the size of the list of workers
$index \leftarrow 0$
$chunks \leftarrow$ empty list with size $p$
$listaOp \leftarrow$ list with mutation operators

for $op$ in $listaOp$ do
    $list_{mop} \leftarrow ms[op]$          // list with mutants generated by $op$
    $k \leftarrow$ size($list_{mop}$)       // returns the size of the list $list_{mop}$
    for $i = 0$ until $k$ do
        $chunks[index] \leftarrow chunks[index] \cup \{list_{mop}[i], ts\}$
        $index \leftarrow (index + 1)\%p$
    end
end
for ($chunk$, $worker$) in ($chunks$, $workers$) do
    Send $chunk$ for $worker$
end
```

- **Distribute Test Cases (DTC)**
  This is another static data-sharing approach similar to DMBO, proposed by Kapfhammer [33]. However, this approach uses the test case distribution between machines instead of using mutants. Similar to DMBO, the number of parts to be processed is defined in terms of the number of machines available for processing, i.e., the number of parts is calculated by $|T|/|M|$, where T is the set of test cases and M is the set of machines available for processing.

  In this approach, the test cases are divided among the number of available computers. Each processing machine receives all the generated mutants and a portion of test cases.

  This algorithm tends to present a problem; as there is no communication between the executing machines, different machines tend to repetitively kill the same mutant, thereby causing unnecessary processing. An advantage of this approach, however, can be observed when the tester is interested in measuring the efficiency of each test case individually, thus requiring that all test cases are performed against all mutants, regardless of the state of the mutant with other test cases;

Algorithm 2 Distribute Test Cases

```
Input: MutantsDictionary: $ms$, TestSet: $ts$, WorkersList: $workers$
$p \leftarrow$ size($workers$)            // Returns the size of the list of workers
$index \leftarrow 0$
$chunks \leftarrow$ empty list with size $p$

for $tc$ in $ts$ do
    $chunks[index] \leftarrow chunks[index] \cup \{tc, ms\}$
    $index \leftarrow (index + 1)\%p$
end
for ($chunk$,$worker$) in ($chunks$,$workers$) do
    Send $chunk$ for $worker$
end
```

- **Given Mutants on demand (GMOD)**
  This algorithm uses a dynamic approach to distribute the work and was proposed by Choi and Mathur [29]. It divides the execution into parts, where each part comprises a single mutant and the entire test set. The algorithm executes a loop that verifies whether there is an available machine or not. If there is, it sends a mutant and the whole test case set to the processing machine and waits for the next available machine to repeat the operation. The algorithm stops when all the mutants are distributed and processed.

  Unlike DMBO and DTC algorithms, GMOD greatly increases network communication because of the high number of messages exchanged between the processing machines and the scheduler. Thus, the network characteristics can be a determining factor in the execution time. On the other hand, due to the dynamic nature of the algorithm, all machines tend to terminate their workload at a similar time interval;

Algorithm 3 Given Mutants on demand

```
Input: MutantsDictionary: $ms$, TestSet: $ts$
$index \leftarrow 0$

for $m$ in $ms$ do
    $chunks[index] \leftarrow chunks[index] \cup \{m, ts\}$
    $index \leftarrow index + 1$
end
$indexProcessed \leftarrow 0$

while $indexProcessed < index$ do
    $executor \leftarrow$ executorAvailable()    // Receive available executor signal
    Send $chunks[indexProcessed]$ for $worker$
    $indexProcessed \leftarrow indexProcessed + 1$
end
```

- **Given Test Cases on Demand (GTCD)**

  GTCOD was also inspired by ideas proposed by Kapfhammer [33]. Similar to GMOD, the GTCOD algorithm also has a dynamic approach, however, it uses the division of the test set as a strategy to create the parts to be processed, instead of mutants.

  The algorithm distributes the execution of the test cases to be processed. Each part consists of only one test case and the whole set of mutants. As in the previous approach, the execution stops when all the test cases are distributed among the processors. Unlike static algorithms, but similar to GMOD, there is an intense flow in the network due to the vast communication between the scheduler and the worker computers. Therefore, the network structure directly influences the algorithm's performance.

  For both DTC and GTCOD, when the number of test cases is lower than the number of available machines for processing, the test cases are divided among the workers, based on the order in which they connect to the parallel test session. Therefore, those workers that do not receive any data for execution finish executing and wait for the next session; and

Algorithm 4 Given Test Cases on Demand

```
Input: MutantsDictionary:$ms$, TestSet:$ts$
$index \leftarrow 0$

for $tc$ in $ts$ do
    $chunks[index] \leftarrow chunks[index] \cup \{ms, tc\}$
    $index \leftarrow index + 1$
end
$indexProcessed \leftarrow 0$

while $indexProcessed < index$ do
    $executor \leftarrow$ executorAvailable()    // Receive available executor signal
    Send $chunks[indexProcessed]$ for $worker$
    $indexProcessed \leftarrow indexProcessed + 1$
end
```

- **Parallel Execution with Dynamic Ranking and Ordering (PEDRO)**

  PEDRO is a hybrid algorithm with static and dynamic characteristics proposed by Reales and Polo [5]. This algorithm was inspired by ideas suggested by Hummel et al. [34], but was designed to be applied in the context of mutation testing.

  This consists of two phases: the first is a ranking phase, which resembles the DMBO algorithm and the second is an ordering phase that resembles the GMDO algorithm. The first phase aims to reduce traffic on the network compared to what happens with the GMOD algorithm. The second phase prevents machines from being idle during the process, preventing some machines from ending their processing before the others.

  In the ranking phase, the execution of the algorithm orders the processors that perform better, distributing an initial configurable workload. Having the results of the first execution in the second ordering phase, the processors that obtained a better result receive a larger workload. The algorithm stops when all the mutants are distributed among the workers.

  The first phase sends a fixed number of mutants for each machine. This phase is useful for ranking the machines for the next phase, ensuring that the machines that performed better in the first phase run a larger amount of data in the second phase. In the first phase, data is sent only once, thereby reducing the network traffic.

  When a machine finishes, the execution of the first phase, the next step is initiated. The algorithm divides the execution into parts of decreasing size. When a machine finishes its run, it gets a new part to be executed. The size of this part depends on the number of mutants running. Thus, the more the execution of the mutants is approaching its end, the smaller the size of the pieces created. This behavior indicates that the second phase of the PEDRO algorithm tends to resemble the GMOD algorithm at the end of the process.

Algorithm 5 Parallel Execution with Dynamic Ranking and Ordering

```
Input: MutantsDictionary: $ms$, TestSet: $ts$, WorkersList: $workers$, int:$\alpha$
// First phase
$p \leftarrow$ size($workers$)          // Returns the size of the list of workers
$k \leftarrow$ size($ms$)               // Returns the size of the mutant list
$index \leftarrow 0$
$chunks1 \leftarrow$ empty list with size number of machines used in the test session
$sizePartes \leftarrow$ max($k/(p * \alpha)$,1)
for $i = 0$ until $p$ do
    for $j = 0$ until $sizePartes$ do
        $chunks1[i] \leftarrow chunks1[i] \cup \{ms[index], ts\}$
        $index \leftarrow (index + 1)\%p$
    end
end
for (chunk,worker) in (chunks,workers) do
    Send $chunk$ for $worker$
end
// Second phase
while $index < k$ do
    $executor \leftarrow$ executorAvailable()    // Receive available executor signal
    $sizePartes \leftarrow$ max($k - index/(p * \alpha)$,1)
    $parte \leftarrow \emptyset$

    for $i = 0$ until $sizePartes$ do
        $parte \leftarrow chunk \cup \{ms[index], ts\}$
        $index \leftarrow (index + 1)$
    end
    Send $chunk$ for $worker$
end
```

4.2. *Architecture*

In order to provide support to the algorithms, the *Proteum* tool was extended so as to evaluate the behavior of each algorithm when applied to the context of programs developed in C.

Each piece of *Proteum* – create a test session, create mutants, run mutants, for instance – is an independent program that can be run in a separate shell process. Due to this independence, an architecture was created that executes mutation testing in parallel. The focus of this approach is to make available an infrastructure that supports mutation testing execution, clearly meeting the demand of execution using load balancing algorithms.

An interface was developed to facilitate communication with the sequential version of the tool. Then, an architecture that controls data distribution and module execution was defined so that communication can take place between various instances of the sequential version of *Proteum*. Figure 3 shows in a simplified way of how communication is accomplished. The interface is responsible for handling the data to run in *Proteum* and then receiving the data produced, treating them appropriately to be passed along to the architecture.

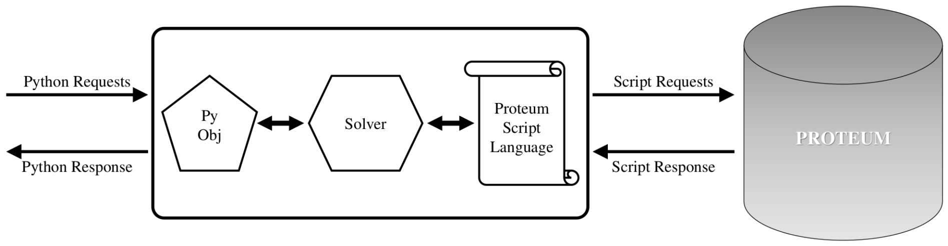


Fig. 3: Interface to allow communication between the architecture and *Proteum*

Figure 4 shows how the architecture was designed to use *Proteum* in parallel[a]. The interface was constructed using *Python* programming language[b] and consists of a set of routines responsible for making calls to the tool modules, as well as analyzing data received and presenting it suitably to each module. Communication between the machines was also implemented using *Python*, with the help of *ZMQ* library[c] used to abstract the communication layer between the machines connected in the architecture.

[a]`https://github.com/stevao-andrade/parallel-proteum`
[b]`www.python.org`
[c]`www.zeromq.org`

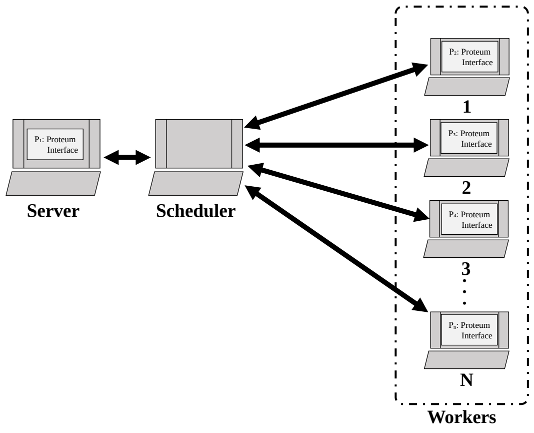


Fig. 4: Parallel Mutation Testing - Architecture.

The architecture was designed using three different types of machines (server, scheduler, and worker) and each type of machine carries out a certain set of tasks. Below we present the main functionalities of each type of machine and present a state machine diagram that describes its behavior:

- **Server**:

  The server is the machine responsible for controlling mutation testing execution. It is responsible for defining the group of mutants and the test set to be used in the test session.

  The process begins identically to the sequential mutation testing execution. The mutants are generated and the test set that will be used during the execution is defined. After that, this information, together with a copy of the program, is sent to the scheduler. The server then waits for the scheduler to distribute the data and receives the results in order to process them and report back to the server.

  After receiving the data, the server summarizes the results and calculates the mutation score. At this point, a new iteration starts with new data.

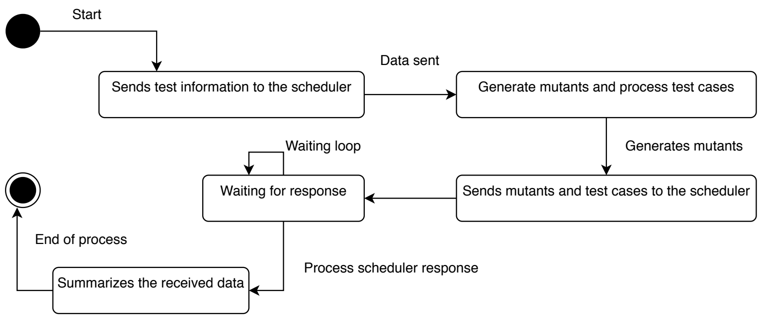

Start
Sends test information to the scheduler
Data sent
Generate mutants and process test cases
Generates mutants
Sends mutants and test cases to the scheduler
Waiting loop
Waiting for response
Process scheduler response
Summarizes the received data
End of process

Fig. 5: State Machine Diagram - Server Behavior

- **Scheduler**:

  This is the single machine that has a connection with all other machines that participate in the test session. It is responsible for receiving and controlling the data distribution during the tests.

  When the process starts, the scheduler defines the number of processing machines to be used in the test session and the load balancing algorithm that will be used. Then, it waits for connections from both the server and workers until reaching the number of connections defined.

  The scheduler then starts the data distribution process. It can divide workloads statically or dynamically, depending on the load balancing algorithm used. For all algorithms, the scheduler always receives all information about mutants and test sets from the server. However, the way that information is distributed to the workers depends on the load balancing approach.

  When the workers end their processing, the scheduler receives the results and reports them to the server. At this point, the scheduler waits to receive new test data for distribution or for a message that determines the end of the execution.

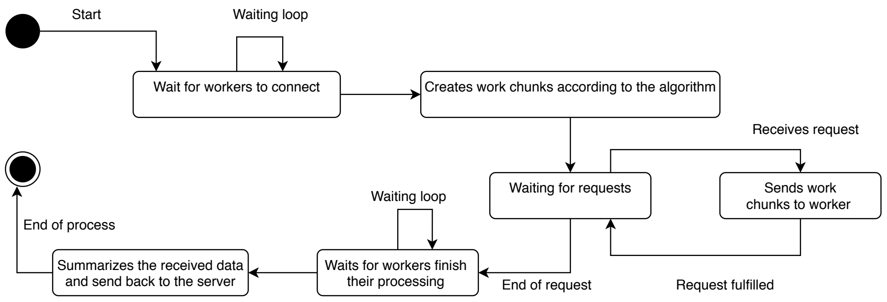

Start
Waiting loop
Wait for workers to connect
Creates work chunks according to the algorithm
Receives request
Waiting for requests
Sends work chunks to worker
Request fulfilled
End of request
Waiting loop
Waits for workers finish their processing
Summarizes the received data and send back to the server
End of process

Fig. 6: State Machine Diagram - Scheduler Behavior

- **Worker**:

  Workers are the machines responsible for executing the mutants with the set of test cases provided. In general, each worker behaves as a sequential version of *Proteum*. However, the way data is processed differs from a conventional execution because the worker runs its mutants based on load balancing algorithms, processing only a portion of the total set of mutants or test cases. In the beginning, all workers connect to the scheduler and receive a copy of the program under test and some test cases. Based on this information, each worker acts as a sequential version of *Proteum* and creates its own test session. After creating the test session, workers make requests to the scheduler indicating that they are ready to process data. In response to such requests, workers receive a set from the scheduler comprising identifiers in order to know the mutants and test cases that need to be processed.

  The information is processed by the worker based on the load balancing algorithm adopted by the scheduler. Therefore, the worker needs to know the approach used in order to make the proper requests. After processing all information, the worker returns a summary of the processed data to the scheduler with information about the execution of the mutants.

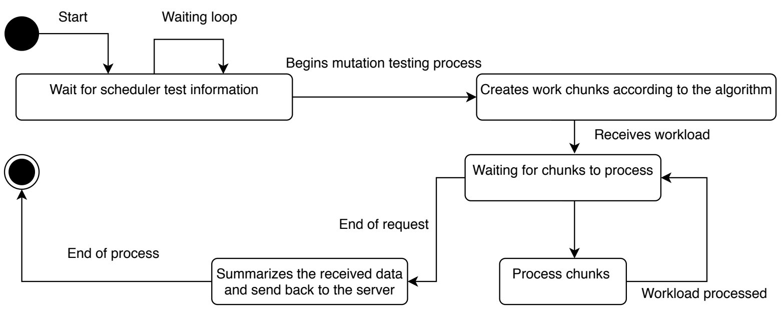

Start
Waiting loop
Wait for scheduler test information
Begins mutation testing process
Creates work chunks according to the algorithm
Receives workload
Waiting for chunks to process
End of request
Process chunks
Workload processed
Summarizes the received data and send back to the server
End of process

Fig. 7: State Machine Diagram - Worker Behavior

5. Experimental Setup

The aim of this experiment is to analyze how the load balancing algorithms behave when inserted during a mutation testing application in parallel, as well as to contribute to improving the efficiency and applicability of the criterion. The experiment used the described parallel architecture, and the results should point the most efficient algorithms. In particular, the following research questions (RQ) were investigated:

RQ_1 : Is it possible to improve computational efficiency of mutation testing applicability using parallel computation techniques?

RQ_2 : Is there a significant difference between the response time of the load balancing algorithms implemented in the proposed architecture?

$\mathbf{RQ_3}$: What is the relationship between the results of this work compared to similar studies in the literature?

To evaluate $\mathbf{RQ_1}$, the implementation of the architecture presented in Section 4 is compared with the sequential version of the mutation testing tool, *Proteum*.

To evaluate $\mathbf{RQ_2}$, the results of the experiments conducted with each load balancing algorithm are compared with each other, aiming at determining how each load balancing algorithm behaves during the experiments and which of them perform better.

To evaluate $\mathbf{RQ_3}$, the results of the experiments are compared with the results presented by Reales and Polo [5], aiming at determining how the load balancing algorithms behave when applied in different contexts. The original results are presented in the context of mutation testing applied to Java programs and the experiment conducted in this work, focusing on programs developed in C language. Furthermore, the tools used in each case are not the same and have different structures, which can lead the load balancing algorithms to different behaviors.

5.1. *Goal Definition*

For the goal definition, the Goal/Question/Metric (GQM) model adapted from Wohlin et al. [35] was used. The GQM model presents the objectives of the experiment divided into five parts:

- **Object of study:** The objects of study are the five load balancing algorithms presented and the *Proteum* tool.
- **Purpose:** The purpose of this experiment is to evaluate the mutation testing applicability using parallel execution techniques. Specifically, it investigates the behavior of five load balancing algorithms applied to mutation testing. The experiment offers a perspective concerning to what extent the computational performance of the mutation testing can be improved using this approach.
- **Perspective:** This experiment is conducted from the researcher's point of view.
- **Quality focus:** The primary effect under investigation is the computational cost measured by the execution time. The execution time of a program being tested is defined as the time spent by the tool to execute the program and all the mutants against a test set.
- **Context:** This experiment was conducted using the *Proteum* mutation testing tool in a set of machines executing the *Ubuntu Linux 14.04 LTS* operational system.

  As the implementation is an academic proof of concept, the results presented here are intended for academic environments.

5.2. *Hypothesis Formulation*

The research questions ($\mathbf{RQ_1}$,$\mathbf{RQ_2}$) were converted into hypotheses so that the statistical tests could be carried out.

For research question $\mathbf{RQ_1}$, the following hypotheses were defined:

Null Hypothesis, H_0**:** There is no significant difference in the execution time spent on the mutation testing application using a sequential execution and parallel execution.

$$H_0\textbf{:}\ \mu_{Proteum} = \mu_{Proteum^p}$$

Alternative Hypothesis, H_1**:** There is a significant difference in the execution time spent on the mutation testing application in parallel with relation to a sequential execution.

$$H_1\textbf{:}\ \mu_{Proteum} \neq \mu_{Proteum^p}$$

Where $\mu_{Proteum}$ is the average time to run a test session using the sequential version of *Proteum* and $\mu_{Proteum^p}$ is the average time to run a test session using its parallel version.

The same procedure was adopted for research question $\mathbf{RQ_2}$. Thus, the following hypotheses were defined:

Null Hypothesis, H_0**:** There is no significant difference between the execution time spent by each analyzed algorithm in parallel execution of the mutation testing.

$$H_0\textbf{:}\ \mu_{DMBO} = \mu_{DTC} = \mu_{GMOD} = \mu_{GTCOD} = \mu_{PEDRO}$$

Alternative Hypothesis, H_1**:** There is a significant difference between the execution time spent by each analyzed algorithm in parallel execution of the mutation testing.

$$H_1\textbf{:}\ \mu_{DMBO} \neq \mu_{DTC} \neq \mu_{GMOD} \neq \mu_{GTCOD} \neq \mu_{PEDRO}$$

Where μ_{alg} is the average time to run a test session using load balancing algorithm *alg* in the parallel version of *Proteum.*

5.3. *Environment description*

The configuration of the machines used to conduct the experiment can be seen in Table 1. The algorithms were tested on ten environments with different characteristics, five of them homogeneous (all computers have the same configuration) and five of them heterogeneous. The machines were connected under an IP network and the speed in all the links was 100 Mbits/s.

The results used as the baseline in a sequential environment were obtained by performing the experiment using a machine with the configuration *ID 1*.

The homogeneous environments consist of a server machine, another scheduler machine and 4, 8, 12, 16 and 32 worker machines. All configurations in this environment were assembled using machines described by *ID 1* in Table 1.

Table 1: Characteristics of the computers used in the experiment

ID	Processor	Clock (GHz)	Memory	Cores
1	Core i7 (3770)	3.4 GHz	8GB	4
2	Core 2 Duo (E8400)	3 GHz	4GB	2
3*	VM	2 GHz	1GB	2
4*	VM	1 GHz	500MB	1

*Virtual machines ID 3 and 4 run, respectively, on top of ID 1 and ID 2 machines.

The heterogeneous environments are formed by a server machine and the scheduler machine running as *ID 1* configuration and workers were assembled as follows:

- **4 workers**: each machine has a different configuration, one machine of each *ID* described in Table 1.
- **8 workers**: this environment was assembled using two machines for each *ID* described in Table 1.
- **12 workers**: this environment was assembled using three machines for each ID described in Table 1.
- **16 workers**: this environment was assembled using four machines for each ID described in Table 1.
- **32 workers**: this environment was assembled using eight machines for each *ID* described in Table 1.

5.4. *Experiment Design*

For all the experiments, two independent variables were controlled. The first is a ratio variable and defines the number of worker machines. The second is nominal and defines which load balancing algorithm should be used.

The independent variables altered by treatment are:

- Number of workers: ratio variable which defines the number of worker machines used in the test session;
- Type of algorithm used: nominal variable which is alternated to define the load balancing algorithm (DMBO, DTC, GMOD, GTCOD, PEDRO) used in the test session.

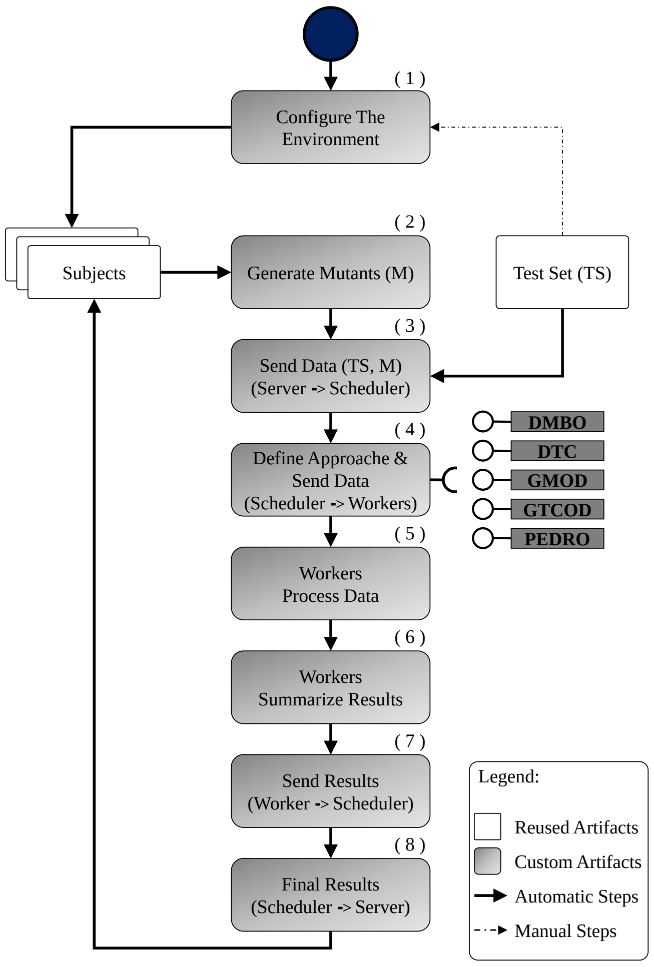
The dependent variable collected by the treatment is the *time* used by all machines involved in the test session from the beginning of the process to the final ending message. This is a continuous variable (parametric) collected in seconds.

The experiment is divided into two steps. The purpose of the first step is to perform a hypothesis test to find out how much parallel execution overperforms the sequential execution. As it is a well-known fact, implementations that use a distributed infrastructure tend to perform better than conventional structures, confirming this fact would increase the reliability of the data obtained, which provides subsidies for the second step of the experiment.

The aim of the second step is to individually evaluate each load balancing algorithm so that the most suitable algorithm can be found and, therefore, it is possible to draw a comparison with the results presented by Reales and Polo [5].

The procedure carried out during the execution of the experiment is described in Figure 8 and consists of the following steps:

(1) Configuration of the environment in the network machines to properly run the proposed approach.
   (a) Configuration of the server: informs the programs that will be used in the test session, inform the set of test cases to be used, enter the Internet Protocol (*IP*) address of the scheduler and start the process.
   (b) Configuration of the scheduler: report the number of workers that will be used and inform the load balancing algorithm to be used during the process.
   (c) Configuration of the workers: informs the *IP* address of the scheduler in each worker machine and start the process.
(2) Generation of the mutants on the server.
(3) Server sends information about the test session to the scheduler.
(4) Scheduler distributes the test data for workers using one of the load balancing algorithms.
(5) Workers execute the mutants against the test set, according to the selected approach.
(6) Summary of the results in each worker.
(7) Each worker informs the results to the scheduler.
(8) Scheduler passes along the final data to the server.



(1)
Configure The Environment
(2)
Subjects
Generate Mutants (M)
Test Set (TS)
(3)
Send Data (TS, M) (Server -> Scheduler)
(4)
Define Approache & Send Data (Scheduler -> Workers)
DMBO
DTC
GMOD
GTCOD
PEDRO
(5)
Workers Process Data
(6)
Workers Summarize Results
(7)
Send Results (Worker -> Scheduler)
(8)
Final Results (Scheduler -> Server)
Legend:
Reused Artifacts
Custom Artifacts
Automatic Steps
Manual Steps

Fig. 8: Steps taken to carry out the experiment.

A complete list of the program names, number of functions, lines of code (LOC), number of mutants and number of test cases used is described in Table 2.

The subjects used in this experiment consist of a list of forty programs written in C. Some of the programs were taken from books about Software Engineering, the other part is available in the Software-artifact Infrastructure Repository[d], and most of them have already been used in other experiments, for instance: Delamaro et al. [36, 37] and Ammann et al. [38]. The test sets used in this experiment were also used in previous studies.

[d]`http://sir.unl.edu/`

As the primary focus of this experiment is to measure the capability to execute a large number of mutants, information related to the equivalence of mutants was ignored.

Table 2: Subjects used in the experiments.

Program	**Func**	**LOC**	**Mut**	**Test Cases**
boundedQueue	6	49	1121	13
cal	71	18	891	8
Calculation	7	46	1118	13
checkIt	1	9	104	9
CheckPalindrome	1	10	166	8
countPositive	1	9	151	5
date-plus	3	132	2421	44
DigitReverser	1	17	496	5
findLast	1	10	198	6
findVal	1	7	190	7
Gaussian	6	23	1086	21
Heap	7	41	1079	8
InversePermutation	1	15	576	12
jday-jdate	2	49	2821	27
lastZero	1	9	173	5
LRS	5	51	1132	8
MergeSort	3	32	991	18
numZero	1	10	151	5
oddOrPos	1	9	361	7
pcal	8	204	6419	49
power	1	11	268	9
printtokens	17	349	4322	34
printtokens2	18	275	4734	27
printPrimes	2	35	715	7
Queue	6	64	469	12
quicksort	1	23	1026	13
RecursiveSort	1	17	555	8
replace	20	390	11100	142
schedule	18	213	2108	45
schedule2	16	195	2626	41
Space	135	6218	135875	500
Stack	6	56	460	11
stats	1	19	884	7
sum	1	7	165	6
tcas	8	63	2384	62
testPad	1	24	629	14
totInfo	7	214	6693	49
trashAndTakeOut	2	19	599	12
twoPred	1	10	246	10
UnixCal	4	119	4852	27
Total	324	9075	201393	1313

6. Results and Discussion

This section presents the experimental findings. The analyses are divided into two subsections: *(I)* descriptive statistics and *(II)* hypothesis testing.

The experimental results are presented considering two different configurations (homogeneous and heterogeneous). For both configurations, the execution time for all the evaluated subjects is presented. It is emphasized that due to the size and number of test cases and execution time, the results for the *Space* program are presented individually.

Time values collected in the experiment can be affected by various elements, such as the operating system scheduler, interruptions, etc. To minimize this risk, the experiment was carried out several times and time measurements reported here correspond to the average of these multiple executions.

In the homogeneous environments, all programs (except *Space*) were executed 15 times. Due to time constraints to access the university laboratory, in heterogeneous environments, the programs were executed five times. The same happened with the *Space* program, which was executed five times in homogeneous environments and five times in heterogeneous environments.

Since we were unable to collect the same amount of data for all observed experiment scenarios, to make sure this would not bias the experiment, we estimated the probability that the true population means lies within a range around our sample mean. In order to ensure that the results presented in the sections below truly reflect behavior that tends to repeat independently of the number of observations to be considered, we calculated the confidence interval using a significance level of 95%.

A confidence interval gives an estimated range of values which is likely to include an unknown population parameter, the estimated range being calculated from a given set of sample data [39]. The results presented in the tables enable us to have a better idea of how the results of the overall population tend to behave, based on the results of our specific study.

Thus, we present a summary of the collected data, presenting minimum and maximum values for 95% confidence interval, the mean of the experiment executions, and their respective standard deviations and standard error. Due to a large amount of data, all the data related to this evaluation (worksheets 9, 10, and 11) and a complete set of the data of the experiment are available externally as a lab-package[e].

6.1. *Descriptive Statistics*

This sub-section provides descriptive statistics of the experimental data. The first analysis to be made is related to the speedup reached while using a parallel mutant execution approach when compared to the sequential version of the *Proteum* tool

[e]`http://bit.ly/2M6Nuio`

and to determine the most efficient approach to be used.

Initially, we present an analysis of the results considering all 40 programs. This analysis is referred to as a "global experiment". In this scenario, the collected data concerns the total time spent on executing the whole experiment.

The results are presented for both configurations, in homogeneous and heterogeneous environments, and show the execution time for the experiment. The experiment consists of running all mutants with all test sets listed in Table 2. The execution time collected consists of the time spent to complete the execution, in other words, the time spent to execute all the mutants from all the programs.

Due to the size and number of mutants and the number of test cases, the results for the *Space* program are assessed individually to avoid any bias in the experiment results.

After this analysis, data on the performances of each subject are detailed individually. This discussion is relevant to describe how each of the approaches behaves for each subject program.

6.1.1. *Sequential x Parallel*

As the purpose of this experiment is to measure to what extent the efficiency of the mutation testing execution can be improved, the test session of the *Proteum* tool in *research mode* was adopted. This approach obliges all existing test cases in the session to be executed with the whole set of mutants, regardless of the mutant state when the test case is being executed. In a conventional test session, a dead mutant would not be further executed against other test cases.

The left side of the Figure 9 shows that regardless of the load balancing algorithm, even for the simplest scenario, where four workers were used to process the mutants, the mutation testing execution in parallel shows great speedup. The scenario that presented the lowest speedup was the one that used the GTCOD algorithm, although it still improved the efficiency of the experiment in 70.51% at the homogeneous environment and in 45.28% in the heterogeneous environment in comparison with the sequential execution.

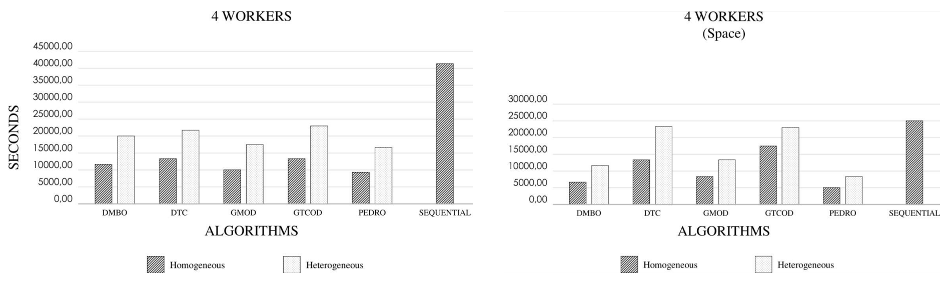


Fig. 9: Average of the execution time for 4 workers compared to the sequential execution of the mutation testing.

The right side of the Figure 9 shows the same comparison for the *Space* program. The results show that the algorithm GTCOD had the worst performance and was able to achieve a speedup of 31.11% compared to the sequential execution in the homogeneous environment, while in the heterogeneous environment the algorithm that had the lowest speedup compared to the sequential execution was the DTC algorithm, with a speedup of 12.08%.

Table 3 shows the comparison of the percentage gain (% Dif) in efficiency for each algorithm is presented. The data presented compares the total time spent on executing the experiment using each parallel algorithm and 4 workers to the time spent on executing the experiment in a sequential approach. The (% Dif (HO)) refers to the results of the experiment conducted in the homogeneous environment, whereas the (Dif % (HE)) column refers to the results of the experiment conducted in the heterogeneous configuration.

Table 3: Average execution time using 4 Workers (39 Programs).

Algorithm	**4 Workers (HO)**	**4 Workers (HE)**	**Sequential**	**% Dif (HO)**	**% Dif (HE)**
DMBO	10774.08s	19903.10s		73.53%	51.09%
DTC	11889.48s	20388.76s		70.78%	49.90%
GMOD	9617.33s	16217.48s	40695.49s	76.37%	60.15%
GTCOD	11999.66s	22268.36s		70.51%	45.28%
PEDRO	8865.40s	15706.86s		78.22%	61.40%

In both scenarios, the PEDRO algorithm presented the best result, ensuring efficiency of 78.22% compared to the sequential mutation testing execution in the homogeneous configuration and 61.40% in the heterogeneous configuration, followed by DMBO, GMOD, and, finally, the algorithms with the worst results are the approaches that use the DTC and GTCOD distribution of test cases.

The same analysis was performed for the *Space* program. Table 4 shows the results for both heterogeneous and homogeneous environments. The PEDRO algorithm achieved the best results, reaching a speedup of 79.47% in the homogeneous configuration and 70.01% in the heterogeneous configuration.

Table 4: Average execution time using 4 Workers (*Space*).

Algorithm	**4 Workers (HO)**	**4 Workers (HE)**	**Sequential**	**% Dif (HO)**	**% Dif (HE)**
DMBO	63737.858s	109704.8s		74.60%	56.29%
DTC	25667.24 6s	220661.9s		49.93%	12.08%
GMOD	73664.964s	122074.9s	250971.072s	70.65%	51.36%
GTCOD	172883.118s	215658.6s		31.11%	14.07%
PEDRO	51523.636s	75274.39s		79.47%	70.01%

Regarding the execution of the experiment using 8 workers, Figure 10 shows the execution results for the set of 39 programs and the *Space* program. The results show that in a setup using 8 workers, the PEDRO algorithm achieved the best results.

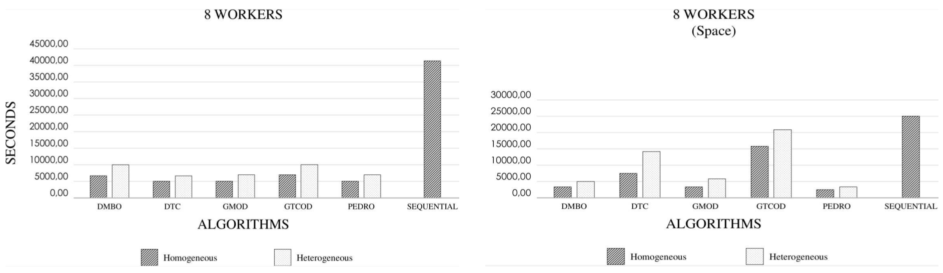


Fig. 10: Average execution time for 8 workers compared to the sequential execution of the mutation testing.

Table 5 shows the percentage of reduction in the execution time for the set of 39 programs. For the homogeneous environment, the PEDRO algorithm provided a reduction of 88.42% and in the heterogeneous environment, it achieved a reduction percentage of 83%. Table 6 shows the same analysis for the *Space* program.

Table 5: Average execution time using 8 Workers (39 Programs).

Algorithm	**8 Workers (HO)**	**8 Workers (HE)**	**Sequential**	**% Dif (HO)**	**% Dif (HE)**
DMBO	6759.5s	9656.16s		83.39%	76.27%
DTC	4957.37s	6956.53s		87.82%	82.91%
GMOD	5144.82s	7480.02s	40695.49s	87.36%	81.62%
GTCOD	7356.43s	10054.52s		81.92%	75.29%
PEDRO	4712.47s	6613.17s		88.42%	83.75%

Table 6: Average execution time using 8 Workers (*Space*).

Algorithm	**8 Workers (HO)**	**8 Workers (HE)**	**Sequential**	**% Dif (HO)**	**% Dif (HE)**
DMBO	33650.692s	49364.9s		86.59%	80.33%
DTC	5485.912s	95626.36s		73.91%	61.90%
GMOD	37227.718s	54846.3s	250971.072s	85.17%	78.15%
GTCOD	157377.75s	205921.7s		37.29%	17.95%
PEDRO	26180.888s	38603.92s		89.57%	84.62%

Following the same trend of the execution with 4 workers, the algorithm with worst the performance using 8 workers was the GTCOD, with 81.92% reduction of the execution time in the homogeneous environment and 75.29% in the heterogeneous environment. Regarding the results of the GTCOD for the *Space* program, the percentage of reduction for the homogeneous environments and heterogeneous were respectively 37.29% and 17.95%, showing a drastic loss of performance compared to the other algorithms.

It can be observed in Figures 11, 12, 13 and Tables 7, 8, 9, 10, 11 and 12 that in all scenarios, the parallel mutation testing application shows a significant speedup compared to the sequential execution.

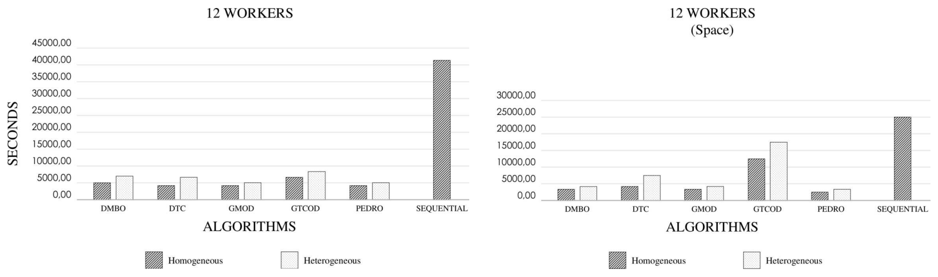


Fig. 11: Average execution time for 12 workers compared to the sequential execution of the mutation testing.

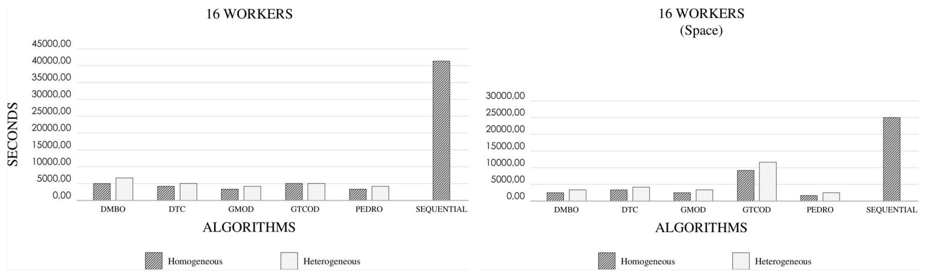


Fig. 12: Average execution time for 16 workers compared to the sequential execution of the mutation testing.

Table 7: Average execution time using 12 Workers (39 Programs).

Algorithm	12 Workers (HO)	12 Workers (HE)	Sequential	% Dif (HO)	% Dif (HE)
DMBO	4898.29s	6583.68s		87.96%	83.82%
DTC	4101.37s	5533.00s		89.92%	86.40%
GMOD	3680.07s	5063.30s	40695.49s	90.96%	87.56%
GTCOD	6174.15s	8008.15s		84.83%	80.32%
PEDRO	3444.75s	4601.23s		91.54%	88.69%

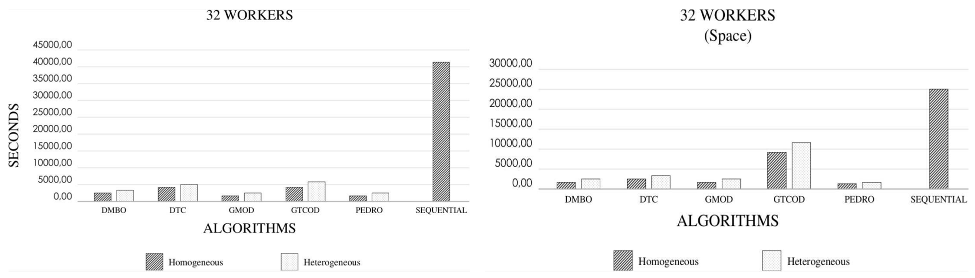


Fig. 13: Average execution time for 32 workers compared to the sequential execution of the mutation testing.

Table 8: Average execution time using 12 Workers (*Space*).

Algorithm	**12 Workers (HO)**	**12 Workers (HE)**	**Sequential**	**% Dif (HO)**	**% Dif (HE)**
DMBO	22489.288s	31357.34s		91.04%	87.51%
DTC	4243.244s	60827.1s		82.37%	75.76%
GMOD	25064.174s	34929.88s	250971.072s	90.01%	86.08%
GTCOD	125615.406s	159815.8s		49.95%	36.32%
PEDRO	17135.916s	23577.24s		93.17%	90.61%

Table 9: Average execution time using 16 Workers (39 Programs).

Algorithm	**16 Workers (HO)**	**16 Workers (HE)**	**Sequential**	**% Dif (HO)**	**% Dif (HE)**
DMBO	4395.83s	5374.80s		89.20%	86.79%
DTC	4219.26s	5223.06s		89.63%	87.17%
GMOD	2822.87s	3634.78s	40695.49s	93.06%	91.07%
GTCOD	5413.47s	5413.47s		86.70%	86.70%
PEDRO	2711.74s	3493.24s		93.34%	91.42%

The scenario that achieved the best results was the configuration using 32 workers. In the set of 39 programs, the PEDRO algorithm was able to reduce the execution time in the homogeneous environment experiment by 95.66%. For the heterogeneous environment, the results were very similar, reaching 94.56% of reduction.

The GMOD algorithm presented results close to PEDRO and it was able to reduce the execution time by 95.26% and 93.99%, respectively for homogeneous and heterogeneous environments, followed by the DMBO algorithm with 94.07% and 92.87%. DTC and GTCOD algorithms showed poorer results, but there was still a great speedup, with savings of about 90% and 88%, respectively, for homogeneous and heterogeneous environments.

The results of running the *Space* program follow a similar pattern. In the homogeneous environment, the PEDRO algorithm achieved a reduction percentage of 97.14% compared to the sequential execution. In the heterogeneous environment, the reduction percentage reached 96.32%. DMBO and GMOD algorithms achieved almost the same results with 96.09% and 95.81%, respectively, in the homogeneous

Table 10: Average execution time using 16 Workers (*Space*).

Algorithm	16 Workers (HO)	16 Workers (HE)	Sequential	% Dif (HO)	% Dif (HE)
DMBO	17353.97s	22828.46s		93.09%	90.90%
DTC	4618.22s	46033.35s		86.21%	81.66%
GMOD	19017.12s	25059.88s	250971.072s	92.42%	90.01%
GTCOD	95232.92s	118569.7s		62.05%	52.76%
PEDRO	13026.15s	17278.91s		94.81%	93.12%

Table 11: Average execution time using 32 Workers (39 Programs).

Algorithm	32 Workers (HO)	32 Workers (HE)	Sequential	% Dif (HO)	% Dif (HE)
DMBO	2412.5s	2901.53s		94.07%	92.87%
DTC	3739.07s	4371.14s		90.81%	89.26%
GMOD	1930.95s	2446.40s	40695.49s	95.26%	93.99%
GTCOD	4561.32s	5381.21s		88.79%	86.78%
PEDRO	1764.90s	2215.04s		95.66%	94.56%

Table 12: Average execution time using 32 Workers (*Space*).

Algorithm	32 Workers (HO)	32 Workers (HE)	Sequential	% Dif (HO)	% Dif (HE)
DMBO	9819.408s	13135.35s		96.09%	94.77%
DTC	8740.4s	25380.72s		92.53%	89.89%
GMOD	10511.58s	15366.04s	250971.072s	95.81%	93.88%
GTCOD	49823.37s	68304.64s		80.15%	72.78%
PEDRO	7168.21s	9223.97s		97.14%	96.32%

environment and 94.77% and 93.88% in the heterogeneous environment. The list of the results with DTC and GTCOD algorithms can be found in Table 12.

Figures 14 and 15 show the behavior of the algorithms as a function of the execution time by increasing the number of machines for execution. The sections on the left of the figures refer to the execution in a homogeneous machines' configuration, while the information on the right refers to the heterogeneous configuration.

As expected, the performance improved as the number of machines increased, however, it should be observed that in one scenario, there was a slight increase in the total time of the experiment execution when the number of machines was increased. This behavior can be observed when the DTC algorithm performs better using 12 workers compared to running with 16 workers. Yet, when scaling to 32 machines, the algorithm regained performance and achieved its best result.

It is well known that there is always a maximum point to scale up to a parallel infrastructure. From that point, increasing the number of machines does not ensure that the performance continues to improve. On the contrary, increasing the number of machines causes the efficiency to stabilize or even decrease.

To explain the behavior presented by the DTC algorithm, we should analyze the individual results of each subject program. If we look at the worksheet 2 in the

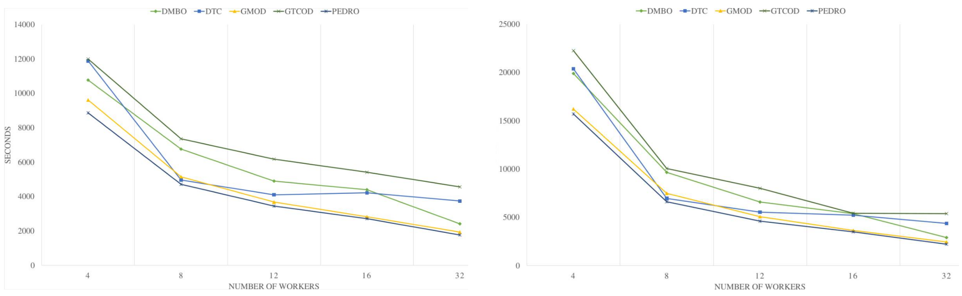


Fig. 14: Execution time X number of workers (39 programs).

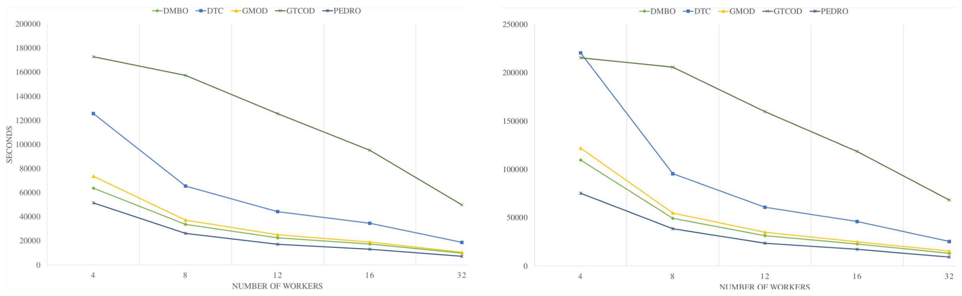


Fig. 15: Execution time X number of workers (*Space*).

labpackage[f], it can be observed that many subjects (*boundedQueue*, *countPositive*, *date-plus*, *DigitReverser*, *FindLast*, etc.) had their best point of scalability in the experiment configuration with 12 workers. This contributes directly to the DTC algorithm presenting this behavior change compared to other algorithms.

It is important to remember that the behavior demonstrated in this sub-section was already foreseen, as it is expected that the use of complex architectures consisting of various processors can enable extra gains in processing capacity in exchange for more expensive infrastructure. However, the analysis presented so far is a starting point to validate the architecture and the algorithms implemented and it gives a precise view of how much performance improvement can be achieved.

6.1.2. *Evaluating the algorithms*

This subsection focuses on performing an analysis of each load balancing algorithm on each subject program. The average execution time for each algorithm

[f]`http://bit.ly/2M6Nuio`

(DMBO, DTC, GMOD, GTCOD, PEDRO) for each subject is available in detail in worksheets 1, 2, 3, and 4 in the labpackage[g]. Figure 16 shows how each algorithm behaves with respect to the complete set of subject programs. The information is divided into graphs, showing how many subjects each algorithm resulted in, sorted by the best, the second, the third, fourth, and fifth best alternative. There are 39 programs times 5 different homogeneous configurations, making a total of 195 combinations. The top leftmost graph shows that PEDRO was the best choice for more than half (101) of these combinations. DMBO was the best for 21; DTC for 48; GMOD for 16 and GTCOD for only 9.

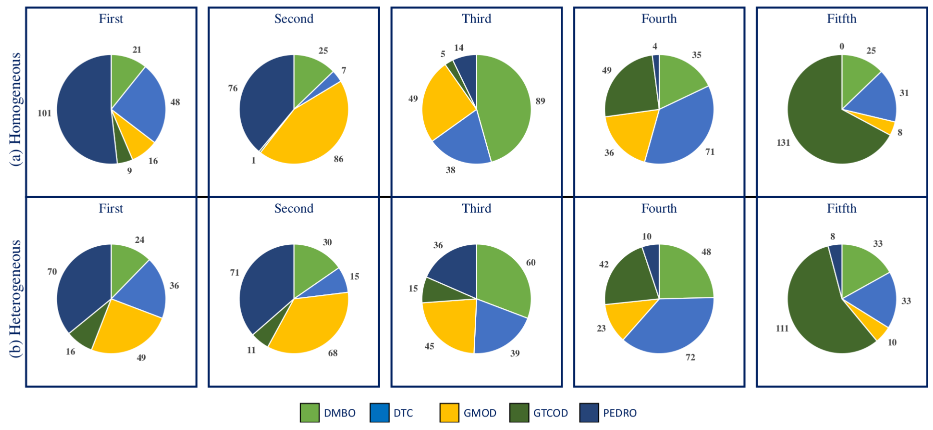


Fig. 16: Graphs demonstrating overall results for each algorithm.

In short, the PEDRO algorithm has the largest number of results as the best alternative to be used. When analyzing the part of the figure that describes how many times each algorithm behaves like the second best alternative, it can be observed that the PEDRO algorithm also has a large number of results, which shows that in cases where the PEDRO algorithm is not the best alternative, it eventually becomes the second best choice.

The GMOD algorithm also has an interesting behavior, although not appearing as the best alternative for a large number of subjects, it ends up becoming an intermediate alternative, focusing its performance mainly as a second or third best approach to be used. GTCOD is the worst alternative to be used for most of the subjects evaluated.

Regarding the results for the *Space* program, for both homogeneous and heterogeneous environments, the PEDRO algorithm appears as the best solution to be used, followed by DMBO and GMOD algorithms, which have almost equal results.

[g]`http://bit.ly/2M6Nuio`

Then, with the worst performance the two algorithms, DTC and GTCOD, which use test case distribution as an approach.

Table 13 shows a comprehensive result regarding the performance of each algorithm. It is clear that as the number of machines increases, the PEDRO algorithm is more and more the best approach in a larger number of subjects. In general, as was shown in Subsection 5.1.1, as the number of machines increases, DTC and GTCOD algorithms tend to have the worst results, even though DTC shows good results for a few subjects.

Table 13: Number of subjects in which each algorithm proved to be the best approach to be used.

Environment	**DMBO**	**DTC**	**GMOD**	**GTCOD**	**PEDRO**
4 Workers (HO)	5	10	5	7	12
4 Workers (HE)	5	4	11	9	10
8 Workers (HO)	2	15	4	0	18
8 Workers (HE)	1	15	9	3	11
12 Workers (HO)	1	15	3	0	20
12 Workers (HE)	4	10	10	1	14
16 Workers (HO)	6	6	4	0	23
16 Workers (HE)	8	4	9	1	17
32 Workers (HO)	7	2	0	2	28
32 Workers (HE)	6	3	10	2	18

For both environments analyzed (homogeneous and heterogeneous), the results regarding the performance of the algorithms seem to follow the same pattern, however, it can be observed that in the homogeneous configuration, the PEDRO algorithm achieved significantly better results.

The good results for the DTC algorithm tend to be justified because of the test session (research mode) used in the experiment and, for some subjects, due to the low number of test cases used in the execution. The research mode causes all existing test cases in the test session to be executed with the whole set of mutants, regardless of the mutant state. In some cases, this approach ends up penalizing the algorithms that use the strategy of the mutant division because they execute mutants redundantly. Another point that can be highlighted is the fact that the low number of test cases used in some of the subjects reduces the network traffic during the application of the algorithms that use the division of test cases, which is also good for their performance.

6.2. *Hypothesis Testing*

To perform the hypothesis testing, only data related to the total time spent on the execution of the experiment is considered, i.e., the time to execute all the mutants from all the 39 subjects against their respective test sets. For each load balancing algorithm and each configuration (4, 8, 12, 16, and 32 workers), in the homogeneous environment, the total time was measured 15 times. For instance, Table 14 shows the results obtained for the configuration of 4 workers. Each column represents the total execution time on each of the 15 runs with the same algorithm. Thus, we can compare, for each configuration, whether there is a significant difference between their average.

The same analysis is applied to the experiment conducted in the heterogeneous environment and for the *Space* program, however, due to time constraints, the execution times were averaged over 5 executions. As in the descriptive analysis section, for the next set of tables, results labeled with *(HO)* refer to the results in the homogeneous environment and those labeled with *(HE)* correspond to the results of the experiment executed in the heterogeneous environment.

Before choosing the statistical test to be applied, we should examine if the collected data follows a linearity and can be considered as a normal distribution or not. The normality of data was measured using the *Shapiro-Wilk* test. The results are presented in Tables 15 and 16 and suggest that the data collected (15 runs for each load balancing algorithm) follow a normal distribution (p-value > 0.05) and, therefore, ensure the application of the parametric tests for statistical evaluation.

Table 14: Sample of data used in hypothesis testing.

Run	**Algorithms**				
	DMBO	**DTC**	**GMOD**	**GTCOD**	**PEDRO**
1	10846.16	12656.36	9609.58	12148.66	8693.68
2	10672.09	11289.13	9613.96	11672.29	8677.45
3	10781.92	11421.04	9592.91	11758.92	8682.54
4	10768.4	11266.76	9595.44	11715.74	9179.95
5	10769.27	11254.82	9638.00	12168.2	8676.19
6	10766.51	11769.21	9652.02	12905.08	8869.45
7	10845.18	12428.3	9658.92	12069.08	8932.94
8	10738.06	12283.68	9660.32	12536.42	8872.71
9	10812.15	11777.03	9615.17	11526.53	9073.54
10	10763.18	11612.68	9586.93	11584.94	8917.41
11	10819.21	12557.56	9579.78	12163.1	8803.9
12	10839.75	11903.49	9578.19	12018.73	8857.24
13	10759.52	11820.97	9649.03	11491.05	8980.4
14	10705.65	12435.25	9659.08	12068.9	8818.76
15	10724.19	11865.96	9570.66	12167.24	8944.79
Std Dev	51.70	482.37	32.89	389.46	148.61
Average	10774.08	11889.48	9617.33	11999.65	8865.4

In order to verify the second research question (RQ_2), a comparison was made between the five treatments referring to each parallel algorithm implemented, for each configuration and environment. To perform this analysis, the One-Way Analysis of Variance ($ANOVA$-Test) was used.

The null hypothesis (H_0) for the $ANOVA$-Test states that the averages for the samples analyzed are equal. Thus, if the results of the one-way $ANOVA$ produce

Table 15: Significance of *p-value* in *Shapiro-Wilk* test (39 programs).

Shapiro-Wilk **test of normality (p-value)**					
Configuration	**Treatment**				
	DMBO	**DTC**	**GMOD**	**GTCOD**	**PEDRO**
4 workers (HO)	0.695027	0.157805	0.069689	0.212271	0.088735
4 workers (HE)	0.41095	0.181453	0.63591	0.406776	0.504033
8 workers (HO)	0.746409	0.969443	0.605938	0.634636	0.646489
8 workers (HE)	0.17294	0.253456	0.39414	0.856089	0.680986
12 workers (HO)	0.965744	0.904120	0.435913	0.656024	0.509029
12 workers (HE)	0.09258	0.340129	0.17428	0.918728	0.240557
16 workers (HO)	0.257434	0.161261	0.141310	0.251307	0.909989
16 workers (HE)	0.467116	0.900244	0.35063	0.657279	0.055276
32 workers (HO)	0.121804	0.152438	0.363226	0.344847	0.711486
32 workers (HE)	0.52935	0.415962	0.8577	0.445106	0.347069

Table 16: Significance of *p-value* in *Shapiro-Wilk* test (*Space*).

Shapiro-Wilk **test of normality (p-value)**					
Configuration	**Treatment**				
	DMBO	**DTC**	**GMOD**	**GTCOD**	**PEDRO**
4 Workers (HO)	0.17168	0.416771	0.54393	0.425571	0.45135
4 Workers (HE)	0.24169	0.542006	0.60836	0.10689	0.910115
8 Workers (HO)	0.848922	0.102131	0.62151	0.596352	0.408771
8 Workers (HE)	0.588744	0.792022	0.35412	0.30832	0.629173
12 Workers (HO)	0.278524	0.765247	0.63432	0.449432	0.543663
12 Workers (HE)	0.32913	0.065441	0.72234	0.13099	0.945646
16 Workers (HO)	0.591304	0.162359	0.51915	0.426584	0.178641
16 Workers (HE)	0.41505	0.869797	0.29184	0.790475	0.85808
32 Workers (HO)	0.89734	0.296197	0.19276	0.439088	0.39184
32 Workers (HE)	0.880547	0.243626	0.49835	0.611388	0.294476

significant results, i.e. rejecting H_0, then the alternative hypothesis (H_1) which considers that the samples are significantly different can be assumed. Therefore, in order to determine if there is a statistical difference between the time execution averages of each algorithm for each one of the scenarios explored, the *ANOVA*-test was applied and the results are presented in Table 17 for the set of 39 programs, and in Table 18 for the analysis performed using the *Space* program. Once again, it is worth noting that we consider *Space* individually because it is larger and takes longer to execute. If we consider it with the small programs, its time will prevail over the total time of all the other programs.

According to the analysis in Tables 17 and 18, as the absolute value for all *p-values* is less than the margin error (0.01%), which corresponds to the complement of the significance level established at 99.99%, statistically, the null hypothesis (H_0) can be rejected for the second research question (RQ_2), which states that the execution time spent by each algorithm is quite similar, justifying the use of anyone of the algorithms. Therefore, there is evidence to corroborate the alternative hypoth-

Table 17: *ANOVA*-Test results (39 programs).

***ANOVA*-Test**		
Configuration	***f-value***	***p-value***
4 Workers (HO)	348.402628352	2.10941538406e-45
4 Workers (HE)	1573.33465977	1.11647566373e-24
8 Workers (HO)	11164.5412797	2.31065291928e-97
8 Workers (HE)	393.60966776	1.04865738524e-18
12 Workers (HO)	2018.40153744	1.78855633482e-71
12 Workers (HE)	534.82641389	5.0684957148e-20
16 Workers (HO)	12450.0944095	5.12445691704e-99
16 Workers (HE)	2650.78504499	6.14355249725e-27
32 Workers (HO)	812.700820959	7.6248427166e-58
32 Workers (HE)	5482.54275139	4.33464641148e-30

Table 18: *ANOVA*-Test results (*Space*).

***ANOVA*-Test**		
Configuration	***f-value***	***p-value***
4 Workers (HO)	22685.24134	2.96902987306e-36
4 Workers (HE)	15638.76242	1.22336035763e-34
8 Workers (HO)	84896.59097	5.5196455664e-42
8 Workers (HE)	63198.43076	1.05597822785e-40
12 Workers (HO)	64223.42194	8.99061653384e-41
12 Workers (HE)	135653.6268	5.08866674732e-44
16 Workers (HO)	66524.59561	6.32286950426e-41
16 Workers (HE)	73593.34549	2.3034850165e-41
32 Workers (HO)	24244.71580	1.52737185631e-36
32 Workers (HE)	41575.60308	6.95244257755e-39

esis (H_1) which states that the execution time varies between the algorithms. This result is confirmed for the set of 39 programs and for the *Space* program in all five configurations and both environments (homogeneous, heterogeneous) analyzed.

Based on the results presented in the descriptive analysis, it can be concluded that PEDRO and GMOD algorithms performed better in the global experiment with remarkably close results. However, as the hypothesis test for the second research question (RQ_2) shows, there are differences between the five algorithms.

As the null hypothesis for the second research question was rejected, we should analyze how different the analyzed samples are. The *ANOVA*-test shows if there is a significant overall difference between the groups analyzed, however, it is not able to determine how the groups differ from each other, therefore a *Post Hoc Test* was performed. The *Tukey HSD*-test [40] was used to indicate which pairs of treatment are statistically different.

Tables 19, 20, 21, 22 and 23 show the results for the *Tukey HSD*-Test, adopting a significance level established at 95% for the configurations of the experiment using respectively 4, 8, 12, 16 and 32 workers in homogeneous environment for the set of 39 programs. Conducting a peer analysis on the tables, it is possible to see the difference between the groups can be observed.

Table 19: *Tukey HSD*-Test for 4 Workers (homogeneous environment).

DMBO subtracted from				
	Difference	Lower	Upper	*P-value*
DTC	1115.4	82253345	14082666	<0.0001
GMOD	-11567493	-14496159	-86388278	<0.0001
GTCOD	1225576	93270945	15184426	<0.0001
PEDRO	-19086827	-22015492	-16158161	<0.0001
DTC subtracted from				
	Difference	Lower	Upper	*P-value*
GMOD	-22721493	-25650159	-19792828	<0.0001
GTCOD	110176	-18269055	40304255	0.8294
PEDRO	-30240827	-33169492	-27312161	<0.0001
GMOD subtracted from				
	Difference	Lower	Upper	*P-value*
GTCOD	23823253	20894588	26751919	<0.0001
PEDRO	-75193333	-10447999	-45906678	<0.0001
GTCOD subtracted from				
	Difference	Lower	Upper	*P-value*
PEDRO	-31342587	-34271252	-28413921	<0.0001

Table 20: *Tukey HSD*-Test for 8 Workers (homogeneous environment).

DMBO subtracted from				
	Difference	Lower	Upper	*P-value*
DTC	-18021227	-18466317	-17576137	<0.0001
GMOD	-16146767	-16591857	-15701677	<0.0001
GTCOD	596932	552423	641441	<0.0001
PEDRO	-2036492	-2081001	-1991983	<0.0001
DTC subtracted from				
	Difference	Lower	Upper	*P-value*
GMOD	187446	142937	231955	<0.0001
GTCOD	23990547	23545457	24435637	<0.0001
PEDRO	-23436933	-27887833	-18986034	<0.0001
GMOD subtracted from				
	Difference	Lower	Upper	*P-value*
GTCOD	22116087	21670997	22561177	<0.0001
PEDRO	-42181533	-46632433	-37730634	<0.0001
GTCOD subtracted from				
	Difference	Lower	Upper	*P-value*
PEDRO	-2633424	-2677933	-2588915	<0.0001

For all p-values lower than the margin of error (0.5%), corresponding to the complement of the significance level established (95%), the difference between the

Table 21: *Tukey HSD*-Test for 12 Workers (homogeneous environment).

DMBO subtracted from				
	Difference	Lower	Upper	*P-value*
DTC	-796918	-89445619	-69937981	<0.0001
GMOD	-1218214	-13157522	-11206758	<0.0001
GTCOD	12758653	11783271	13734035	<0.0001
PEDRO	-14535393	-15510775	-13560011	<0.0001
DTC subtracted from				
	Difference	Lower	Upper	*P-value*
GMOD	-421296	-51883419	-32375781	<0.0001
GTCOD	20727833	19752451	21703215	<0.0001
PEDRO	-65662133	-75415953	-55908314	<0.0001
GMOD subtracted from				
	Difference	Lower	Upper	*P-value*
GTCOD	24940793	23965411	25916175	<0.0001
PEDRO	-23532533	-33286353	-13778714	<0.0001
GTCOD subtracted from				
	Difference	Lower	Upper	*P-value*
PEDRO	-27294047	-28269429	-26318665	<0.0001

Table 22: *Tukey HSD*-Test for 16 Workers (homogeneous environment).

DMBO subtracted from				
	Difference	Lower	Upper	*P-value*
DTC	-176568	-21722694	-13590906	<0.0001
GMOD	-1572954	-16136129	-15322951	<0.0001
GTCOD	1030.98	99032106	10716389	<0.0001
PEDRO	-1684086	-17247449	-16434271	<0.0001
DTC subtracted from				
	Difference	Lower	Upper	*P-value*
GMOD	-1396386	-14370449	-13557271	<0.0001
GTCOD	1207548	11668891	12482069	<0.0001
PEDRO	-1507518	-15481769	-14668591	<0.0001
GMOD subtracted from				
	Difference	Lower	Upper	*P-value*
GTCOD	2603934	25632751	26445929	<0.0001
PEDRO	-111132	-15179094	-70473063	<0.0001
GTCOD subtracted from				
	Difference	Lower	Upper	*P-value*
PEDRO	-2715066	-27557249	-26744071	<0.0001

groups can be seen. Table 19 clearly shows that the PEDRO algorithm achieved better results compared to other algorithms analyzed. If we look at the results presented in Tables 20, 21 and 22, it can be clearly observed that the PEDRO algorithm performed better in all the analyzed configurations, ensuring it is the most appropriate approach used.

The same analysis can be used for the purpose of verifying the performance of

Table 23: *Tukey HSD*-Test for 32 Workers (homogeneous environment).

DMBO subtracted from				
	Difference	Lower	Upper	*P-value*
DTC	13265673	11551991	14979356	<0.0001
GMOD	-563034	-73440227	-39166573	<0.0001
GTCOD	2148818	19774497	23201863	<0.0001
PEDRO	-64759733	-81896561	-47622906	<0.0001
DTC subtracted from				
	Difference	Lower	Upper	*P-value*
GMOD	-18896013	-20609696	-17182331	<0.0001
GTCOD	82225067	65088239	99361894	<0.0001
PEDRO	-19741647	-21455329	-18027964	<0.0001
GMOD subtracted from				
	Difference	Lower	Upper	*P-value*
GTCOD	2711852	25404837	28832203	<0.0001
PEDRO	-84563333	-25593161	86804942	0.6412
GTCOD subtracted from				
	Difference	Lower	Upper	*P-value*
PEDRO	-27964153	-29677836	-26250471	<0.0001

the other algorithms among themselves. For example, when analyzing the DMBO algorithm in Table 19, it can be concluded that it achieves better results than DTC and GTCOD but it had the worst results compared to the GMOD and PEDRO algorithms.

Considering the *Post Hoc*-test, it can be observed that the resulting data is not conclusive in terms of which is the most efficient approach. For instance, the GTCOD algorithm in Table 19, when compared to the DTC algorithm, shows that in this scenario, it cannot be determined which approach is better due to the fact that there is no significant difference between them.

The same analysis can be performed to conclude that using the significance level adopted in the test, it could not be determined which approach was best for the configuration with 32 workers. Table 23 shows that the *p-value* obtained for the comparative analysis between the GMOD and PEDRO algorithms was better than the established margin of error (0.05%), therefore despite the fact that the PEDRO algorithm presented better results compared to the GMOD algorithm, we cannot assume that for the degree of significance adopted in the *Tukey HSD*-test that the same results will always be obtained.

The worksheets 6,7 and 8 in the labpackage[h] reproduce the same analysis for the heterogeneous environment and for the space program. We do not present the results with the same details we have done in this section, but the results follow a similar pattern to those reported for the execution in the homogeneous environment. The *Tukey HSD*-test was able to identify the differences between all pairs of groups,

[h]`http://bit.ly/2M6Nuio`

and the labpackage[i] highlights an example for each scenario in each configuration environment, as well the complete data for this experiment.

6.3. *Discussion of the Results*

The third research question (RQ_3) raises an issue of how the results obtained during the experiments compare to similar studies in the literature. The previously cited work developed by Reales and Polo [5] presents the implementation of the five load balancing algorithms evaluated in this paper, however, it is applied to the context of mutation testing for *Java* programs with different subjects and a different testing tool.

6.3.1. *Summary of the original results*

The original work describes an experimental comparison of five distribution algorithms, one of them (PEDRO) proposed by the authors. Besides presenting this new approach that combines static and dynamic data distribution characteristics, the article provides an extensive evaluation of the characteristics of each distribution algorithm. Subsequently, an experimental evaluation was performed showing that PEDRO presented better results compared to the other algorithms.

The experiment consists of evaluating the five algorithms with a set of four subjects. The authors explore how the results can be influenced by the execution environment, thus the experiment runs over six environments with different characteristics, where three of them are homogeneous and three of them are heterogeneous. The paper also discusses the effect of this kind of configurations and concludes that a heterogeneous configuration tends to present better results than a homogeneous configuration, only with a small number of nodes.

In order to ensure the reproducibility of the results, the paper also conducts statistical analysis to compare all the algorithms and evaluate whether there are significant differences in the results. The established hypotheses were:

$\mathbf{H_0}$: The execution process time is similar to all algorithms, thus any algorithm can be used.

$\mathbf{H_1}$: The execution process time is different from each algorithm, thus the best algorithm should be used.

The statistical results rejected the null hypotheses (H_0), proving that the algorithms do indeed have differences. However, it was observed that in cases where the number of executions and the number of available machines for processing was low, the differences between the PEDRO algorithm and GMOD algorithm tended to be reduced. Thus, a specific study was carried out to verify that the difference between them was, in fact, statistically significant. To do so, an individual analysis with both algorithms and new statistical testing was performed. This time the aim was to check the following hypotheses:

[i]`http://bit.ly/2M6Nuio`

$\mathbf{H_0}$: The total time obtained with the two algorithms are similar.

$\mathbf{H_1}$: the total times obtained with the two algorithms are different, thus the PEDRO algorithm obtains better results.

Once again, the null hypothesis was rejected and the authors were able to state that for the analyzed subjects, in fact, there was a significant difference between the total time obtained by the algorithms, concluding that the PEDRO algorithm achieved the best results.

6.3.2. *Comparing results*

By comparing the results of both experiments we can observe that there was a similarity between the results. Similar to the results of the experiments performed by Reales and Polo [5], the results in this paper show evidence that the PEDRO algorithm presents better results, followed by the GMOD algorithm and the DMBO algorithm.

However, extending the analysis performed by Reales and Polo [5] to individual subjects, we can observe that in the context of *C* programs there was a significant variation between the performance of the five algorithms in each subject.

Table 24: Comparing results presented in Reales and Polo (2013) with the experiments conducted.

Algorithm	***Java* environment**	***C* environment**
DMBO	0	45
DTC	0	84
GMOD	8	65
GTCOD	0	25
PEDRO	16	173
TOTAL	24	392

Table 24 shows this comparison and describes that in the experiments from [5], only PEDRO and GMOD algorithms proved to be the best approach to be used for any of the subjects. In the experiment shown in the present work, DMBO and DTC algorithms also had positive results for some subjects analyzed. The number of total results corresponds to the number of subjects evaluated in each experiment, considering the different configurations in which each subject underwent analysis. The original work presents the results of the execution of four subjects in six different environments (three heterogeneous and three homogeneous) and in this present work, the analysis is performed using forty subjects in ten different environments (five homogeneous and five heterogeneous).

Despite the fact that the PEDRO algorithm proved to be the best alternative for most subjects analyzed in this paper, the DTC, GMOD, and DMBO algorithms have also been shown to be the best approach to be used for some of the subjects analyzed in the experiment. This behavior can possibly be explained due to the type of test session (*research* mode) used while conducting the experiment. Moreover, the size and complexity of the programs and the cardinality of test sets may have influenced the results.

As mentioned earlier, a point that may explain the positive results for algorithms, such as DTC, is the size of some of the subjects in the experiment and the low number of test cases used to execute them. For example, the *trashAndTakeOut* subject, in general, for all tested configurations displays a relatively similar execution time for all balancing algorithms. In scenarios in which the algorithms are assessed together with subjects with a larger number of test cases and mutants, the results obtained in this work and the original results tend to converge.

By analyzing the execution results for the *Space* program, this statement is more easily explained. Figures 17 and 18 shows *boxplots* describing the behavior of the algorithms. It can be observed that the difference between the algorithms tends to be more visible in scenarios where algorithms are forced to process a large number of mutants, as well as test cases and programs with a larger *runtime*.

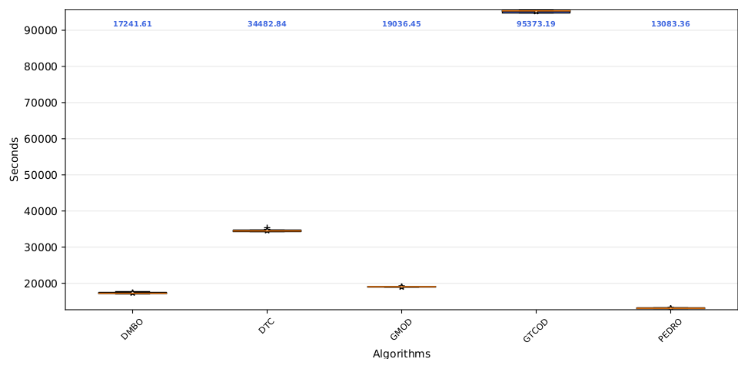


Fig. 17: *Boxplot* comparing the difference between groups (16 Workers - *Space*).

Analyzing the *runtime*, according to the results using 32 workers, the PEDRO algorithm took nearly two hours to execute 500 test cases against over 135000 mutants of the *Space* program, while the DMBO and GMOD algorithms took almost three hours to perform the same task. On average, PEDRO achieved a result 30% faster than its competitors. In summary, we can observe that using PEDRO in larger programs is certainly a better option. This certainty is no longer applicable when we split the work, executing several small programs.

Another point to be discussed is that, as in the original experiment, it was observed that the use of a heterogeneous infrastructure, despite presenting similar results, tends to cause a loss of performance in the results presented by PEDRO algorithm.

Finally, it should be pointed out the low number of subjects (just four) used during the experimental evaluation performed by Reales and Polo [5]. As mentioned

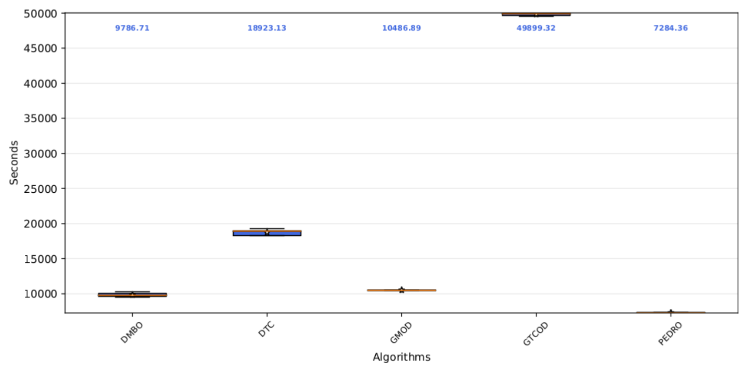


Fig. 18: *Boxplot* comparing the difference between groups (32 Workers - *Space*).

in the threats to validity of that experiment, the nature and size of the programs used as experimental subjects limit the generality of the results, requiring analysis of other types of programs so that results can in fact be generalized and conclusive. Thus, the present work collaborates to extend the results previously obtained.

7. Threats to Validity

The lack of representatives of the subject programs used in the experiment can be a threat to validity. Unfortunately, this is a problem that always affects research related to software engineering since there is no grounded theory to ensure that a determined set of programs is considered a representative sample for experimentation. In order to reduce this risk, the results analyzed were presented based on the comparison of the results from a set of 39 small programs and a relatively larger program (*Space*). The set of subjects was sought, which could cover a large number of applications with different sizes and domains. In addition, our view considering not only the subjects of the experiment in an individual way but also considering all the subjects in a general analysis tells us something about scalability.

Concerning the threats to validity by construction, possible mistakes can be pointed out in the implementation of the architecture proposed and the algorithms analyzed. To minimize this threat, various tests were performed among the modules during the construction period, to ensure that while the experiment was being executed, the implementation in parallel would run smoothly. The *Proteum* tool has been widely used in academic circles in recent years, and therefore, this threat can be ruled out.

Another point to be considered is the number of executions used to evaluate the results from the experiments. The data presented refers to the average of the executions of the experiment. For the homogeneous configuration, 15 executions were performed, while for the heterogeneous configuration and for the *Space* program, 5 executions were performed. This was necessary due to time limitations. It is important to mention that, despite the low number of executions, it has been shown by statistical analysis that it does not affect the data (see Subsection 6.2).

To ensure the reliability of the presented results, the statistical tests presented were computed using a 95% confidence interval in order to guarantee that the conclusions drawn from this work enable us to paint a clear and quantitative picture of the subject.

In addition, the number of executions performed in this experiment is considerably higher than the number of executions carried out in the original experiment by Reales and Polo (2013) [5].

Threats to internal validity define the level of confidence among the expected results and the results obtained. All the variables in the experiments were controlled to minimize the threats. To increase the confidence, the data was analyzed not only in tables and graphs but also by statistical tests to ensure that the results are correctly interpreted and to ensure that the relations presented in the tables and graphs are actually significant and not simply generated by chance.

8. Conclusion and Future Works

This research helped to produce evidence about the applicability of mutation testing using distributed infrastructures to process mutants. To achieve this objective, an architecture was proposed, which made it possible to apply mutation testing using parallel programming techniques. The experiments conducted served to support the hypotheses established in the definition of this research. They were useful to validate the approach developed, proving the efficiency of using parallel computation to support mutation testing and to compare it with previous results in the literature.

The main contribution of this work focuses on strengthening the evidence that using parallel programming techniques can help in the implementation of mutation testing, overcoming bottlenecks regarding the processing time issue. This fact boosts the confidence to encourage implementing mutation testing in real environments. This can also motivate the use of techniques to generate test data together with the mutation testing criterion, which requires a large number of mutant executions. The same is true regarding carrying out experiments, which typically require a high number of executions with many programs. All these factors contribute directly to developing new research in the area.

Another contribution arising from this work is reusing the proposed architecture adopted in the presented experiments. The authors encourage and facilitate the replication of the experiments presented hereby. Replication requires only the implementation of the interface responsible for performing communication between the mutation tool to be assessed and the architecture, for parallel execution. This enables us to carry out new experiments in different contexts to verify the results and comparative studies discussed in this paper.

As future research, we intend to improve the level of communication between the machines involved in the architecture. This phase aims to provide more freedom between the machines involved in the process, facilitating asynchronous execution, and solving problems related to fault tolerance during execution. Moreover, we intend to incorporate techniques to generate test data into the parallel mutation testing execution process. This phase will be another reason to adopt and use the approach.

We plan to extend the conducted experiments to a larger number of worker machines to identify an ideal number of machines that ensure the best performance possible when using the architecture, given that the processing capacity of parallel structures does not increase linearly. From a certain point, increasing the number of machines in the infrastructure would no longer contribute to increasing the efficiency of the approach. This behavior could not be identified with the number of machines used in the experiment conducted in this research, therefore, it is expected that by extending the experiment using more machines, this information can be obtained. There is also interest in integrating the proposed infrastructure to approaches that allow automatic generation of data tests, such as the research proposed by Souza et al. [41]. Finally, we intend to evaluate the approach in terms of a new set of programs aiming at evaluating and corroborating it to generalize the results presented here.

Acknowledgements

Stevão A. Andrade research was funded by FAPESP (Sao Paulo Research Foundation), process numbers 2013/16242-3 and process number 2017/19492-1. This study was also financed by FAPESP process number 2019/06937-0 and Brazilian National Council of Scientific and Technological Development (CNPq) process 308615/2018-2.

References

[1] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes and G. Fraser, Are mutants a valid substitute for real faults in software testing?, in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, *FSE 2014*, (ACM, New York, NY, USA, 2014), pp. 654–665.

[2] J. H. Andrews, L. C. Briand and Y. Labiche, Is mutation an appropriate tool for testing experiments?, in *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, (St Louis, Missouri, 2005), pp. 402 – 411.

[3] T. A. Budd, *Mutation Analysis of Program Test Data*, PhD thesis, Yale University, (Yale University, New Haven, CT, USA, 1980). AAI8025191.

[4] L. Madeyski, W. Orzeszyna, R. Torkar and M. Józala, Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation, *IEEE Transactions on Software Engineering* **40** (Jan 2014) 23–42.

[5] P. M. Reales and M. U. Polo, Parallel mutation testing, *Software Testing, Verification and Reliability* **23**(4) (2013) 315–350.

[6] N. Juristo and O. S. Gómez, Empirical software engineering and verification, in *Empirical Software Engineering and Verification*, eds. B. Meyer and M. Nordio (Springer-Verlag, Berlin, Heidelberg, 2012) pp. 60–88.

[7] F. J. Shull, J. C. Carver, S. Vegas and N. Juristo, The role of replications in empirical software engineering, *Empirical Softw. Engg.* **13** (April 2008) 211–218.

[8] J. Siegmund, N. Siegmund and S. Apel, Views on internal and external validity in empirical software engineering, in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, *ICSE '15*, (IEEE Press, Piscataway, NJ, USA, 2015), pp. 9–19.

[9] R. A. DeMillo, R. J. Lipton and F. G. Sayward, Hints on test data selection: Help for the practicing programmer, *Computer* **11** (April 1978) 34–41.

[10] R. DeMillo, *Mutation Analysis as a Tool for Software Quality Assurance*GIT-ICS, GIT-ICS (School of Information and Computer Science, Georgia Institute of Technology, 1980).

[11] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur and E. Spafford, Design of mutant operators for the c programming language, techreport SERC-TR-41-P, Purdue University (West Lafayette, Indiana, 1989).

[12] M. E. Delamaro, J. C. Maldonado and A. P. Mathur, Interface mutation: An approach for integration testing, *IEEE Transactions on Software Engineering* **27** (May 2001) 228–247.

[13] S. Kim, J. A. Clark and J. A. McDermid, Assessing test set adequacy for object oriented programs using class mutation, in *Proceedings of the 3rd Symposium on Software Technology (SoST'99)*, (Buenos Aires, Argentina, 1999).

[14] Y.-S. Ma, A. J. Offutt and Y.-R. Kwon, Mujava: An automated class mutation system, *Software Testing, Verification & Reliability* **15** (June 2005) 97–133.

[15] A. J. Offutt, Investigations of the software testing coupling effect, *ACM Transactions on Software Engineering and Methodology* **1** (January 1992) 5–20.

[16] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon and M. Harman, Mutation testing advances: an analysis and survey, in *Advances in Computers*, **112** (Elsevier, 2019) pp. 275–378.

[17] R. H. Untch, Mutation-based software testing using program schemata, in *Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE'92)*, (Raleigh, North Carolina, 1992), pp. 285–291.

[18] M. E. DELAMARO, R. A. P. OLIVEIRA, E. F. BARBOSA and J. C. MALDONADO, *Introdução ao Teste de Software – Capítulo 5 – Teste de Mutação*, 2 edn. (Campus, Rio de Janeiro, 2016).

[19] A. Grama, *Introduction to Parallel Computing*Pearson Education, Pearson Education (Addison-Wesley, 2003).

[20] P. Pacheco, *An Introduction to Parallel Programming*An Introduction to Parallel Programming, An Introduction to Parallel Programming (Elsevier Science, 2011).

[21] J. JéJé, *An introduction to parallel algorithms* (Reading, MA: Addison-Wesley, 1992).

[22] M. J. Harrold, Testing: A roadmap, in *Proceedings of the Conference on The Future of Software Engineering*, *ICSE '00*, (ACM, New York, NY, USA, 2000), pp. 61–72.

[23] A. J. Offutt and R. H. Untch, Mutation 2000: Uniting the orthogonal, in *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, (San Jose, California, 2001), pp. 34–44.

[24] Y. Jia and M. Harman, An analysis and survey of the development of mutation testing, *IEEE Transactions of Software Engineering* **37**(5) (2011) 649–678.

[25] M. J. Flynn, Very high-speed computing systems, *Proceedings of the IEEE* **54** (Dec 1966) 1901–1909.

[26] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes and M. Ribeiro, A systematic literature review of techniques and metrics to reduce the cost of mutation testing, *Journal of Systems and Software* (2019).

[27] A. P. Mathur and E. W. Krauser, Modeling mutation on a vector processor, in *Pro-*

ceedings of the 10th International Conference on Software Engineering, *ICSE '88*, (IEEE Computer Society Press, Los Alamitos, CA, USA, 1988), pp. 154–161.

[28] E. W. Krauser, A. P. Mathur and V. J. Rego, High performance software testing on simd machines, *IEEE Transactions on Software Engineering* **17** (May 1991) 403–423.

[29] B. Choi and A. P. Mathur, High-performance mutation testing, *Journal of Systems and Software* **20** (February 1993) 135–152.

[30] A. J. Offutt, R. P. Pargas, S. V. Fichter and P. K. Khambekar, Mutation testing of software using a mimd computer, in *Proceedings of the International Conference on Parallel Processing*, (Chicago, Illinois, 1992), pp. 255–266.

[31] I. Saleh and K. Nagi, Hadoopmutator: A cloud-based mutation testing framework, in *International Conference on Software Reuse*, Springer2015, pp. 172–187.

[32] P. C. Cañizares, M. G. Merayo and A. Núñez, Eminent: Embarrassingly parallel mutation testing, *Procedia Computer Science* **80** (2016) 63 – 73, International Conference on Computational Science 2016, {ICCS} 2016, 6-8 June 2016, San Diego, California, {USA}.

[33] G. M. Kapfhammer, Automatically and transparently distributing the execution of regression test suites, in *Proceedings of the 18th International Conference on Testing Computer Software*, 2001.

[34] S. F. Hummel, E. Schonberg and L. E. Flynn, Factoring: A method for scheduling parallel loops, *Commun. ACM* **35** (August 1992) 90–101.

[35] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering: An Introduction* (Kluwer Academic Publishers, Norwell, MA, USA, 2000).

[36] M. E. Delamaro, J. Offutt and P. Ammann, Designing deletion mutation operators, in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, *ICST '14*, (IEEE Computer Society, Washington, DC, USA, 2014), pp. 11–20.

[37] M. Delamaro, L. Deng, V. Serapilha Durelli, N. Li and J. Offutt, Experimental evaluation of sdl and one-op mutation for c, in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, March 2014, pp. 203–212.

[38] P. Ammann, M. E. Delamaro and J. Offutt, Establishing theoretical minimal sets of mutants, in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, 2014, pp. 21–30.

[39] V. J. Easton and J. H. McColl, Statistics glossary v1.1 (1997).

[40] Y. Benjamini and H. Braun, John tukey's contributions to multiple comparisons, *ETS Research Report Series* **2002**(2) (2002) i–27.

[41] F. C. M. Souza, M. Papadakis, Y. Le Traon and M. E. Delamaro, Strong mutation-based test data generation using hill climbing, in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, *SBST '16*, (ACM, New York, NY, USA, 2016), pp. 45–54.