Parallel mutation testing for large scale systems

Cluster Computing (2024) 27:2071–2097
https://doi.org/10.1007/s10586-023-04074-y

Parallel mutation testing for large scale systems

Pablo C. Cañizares[1] · Alberto Núñez[2] · Rosa Filgueira[3] · Juan de Lara[1]

Received: 5 November 2022 / Revised: 1 June 2023 / Accepted: 2 June 2023 / Published online: 20 June 2023

Abstract
Mutation testing is a valuable technique for measuring the quality of test suites in terms of detecting faults. However, one of its main drawbacks is its high computational cost. For this purpose, several approaches have been recently proposed to speed-up the mutation testing process by exploiting computational resources in distributed systems. However, bottlenecks have been detected when those techniques are applied in large-scale systems. This work improves the performance of mutation testing using large-scale systems by proposing a new load distribution algorithm, and parallelising different steps of the process. To demonstrate the benefits of our approach, we report on a thorough empirical evaluation, which analyses and compares our proposal with existing solutions executed in large-scale systems. The results show that our proposal outperforms the state-of-the-art distribution algorithms up to 35% in three different scenarios, reaching a reduction of the execution time of—at best—up to 99.66%.

Keywords Mutation testing · Parallel mutation testing · Large scale systems · High performance computing · Distributed systems · Testing

1 Introduction

Testing is one of the most widely extended techniques for analysing the correctness and improving the robustness of software systems [1]. During the last decades, testing has been incorporated into various software quality standards, such as, just to name a few, DO-178/ED-12 for safety-critical avionic software [2], IEC 62304 for medical devices [3] and ISO 26262 for road vehicles [4]. Among the strengths of testing, it is worth to mention its scalability and versatility, since it can be used to analyse the correctness of a broad spectrum of systems [5-8].

However, selecting a suitable test suite for properly checking the correctness of systems is complex. Usually, these systems consist of thousands of lines of code, which entails the construction of large test suites to achieve a high percentage of coverage. The size of the test suite has a large impact on the performance of the testing process, so it is therefore required to create effective test suites that can be executed in a reasonable time [9].

Fortunately, there exist techniques to evaluate the strength of test suites for finding errors. Mutation Testing (in short, MuT) is a testing technique that measures the effectiveness of test suites in terms of its ability to detect errors [10-12]. The main idea is to generate syntactically valid variations of a program, each containing a single fault. These faulty programs, called mutants, are executed using the test suite under study in order to determine its effectiveness in finding errors. The MuT process consists of the following stages: (i) generating a test suite; (ii) executing the test cases over the application under study; (iii) generating mutants; (iv) compiling the mutants; (v) detecting equivalent mutants; (vi) executing the test cases over the mutants and comparing the obtained results

✉ Pablo C. Cañizares
pablo.cerro@uam.es

Alberto Núñez
alberto.nunez@pdi.ucm.es

Rosa Filgueira
rf208@st-andrews.ac.uk

Juan de Lara
juan.delara@uam.es

[1] Computer Science Department, Autonomous University of Madrid, Madrid, Spain

[2] Software Systems and Computation Department, Complutense University of Madrid, Madrid, Spain

[3] School of Computer Science, University of St. Andrews, St. Andrews, UK

with the ones obtained in stage (ii) and check if the test cases can detect the mutants.

The adoption of MuT by the industry is limited, mainly, due to the high computational cost associated to this technique and, in a secondary way, by the generation of equivalent mutants and the lack of tool integration [13-15]. We illustrate the computational power needed to perform MuT over a real-world application called *iText* (v 5.0.6).[1] According to the study carried out by Just et al. [13], this application consists of 76,229 lines of code, the number of generated mutants is 126,781, and the test suite consists of 75 test cases, requiring an average execution time of 8.4 s per test. Considering this data, the total time needed to completely conduct the MuT process—in the worst-case scenario—is 126,781 (mutants)*75 (test cases)*8.4 (seconds per test) = 79,872,030 s = 22186,67 h = 924,44 days = 2.53 years, which is clearly unfeasible.

A recent survey that collects the latest advances and trends of MuT reports that leveraging on parallel processing for MuT is an alternative to speed-up the testing process [16]. Hence, it is required to apply computing paradigms that allow exploiting the available computational resources of distributed systems to improve the overall performance. High Performance Computing (in short, HPC) is considered as one of the most suitable solutions to reduce long execution times while providing a good balance between price and performance. The relevance of this paradigm can be shown at the TOP-500 list, where the 500 most powerful commercially available computer systems are clusters [17, 18].

Referring to the previous example, where the execution of the complete MuT process takes 2.53 years, the total execution time could be reduced to 72–96 h—considering that the same process is executed in a cluster with 500 processors—which heavily depends on the distribution algorithm applied and the mutation environment analysed. The use of parallel MuT techniques is therefore desirable to reduce the computational cost of the MuT process without losing effectiveness.

Several techniques have been proposed for improving the performance of the MuT process [19]. However, the parallelisation of this process has been recognised as *an old idea that has not been investigated much* [16]. To the best of our knowledge, the available studies found in the literature only focus on small-size clusters, conducting the experiments by deploying, as maximum, 80 processes [20-22]. The reduced size of these environments does not allow analysing potential drawbacks and bottlenecks, which may hamper the scalability and, consequently, the performance of the MuT process. In previous work [20, 23], we presented several techniques for improving the performance of MuT that were analysed and compared with other works found in the current literature. The experimental setup used in these works included different distribution algorithms. These proposals mitigated several problems that were identified when the MuT process is executed in parallel, like the sequential execution of the test suite over the program under test, and the low computational efficiency achieved by the existing algorithms in the literature. The distribution algorithm and the described techniques were appropriate for small scale environments, providing promising results. However, we identified several bottlenecks that arise when these approaches are applied in large-scale systems to carry out the MuT process.

In this paper, we present an approach to mitigate these potential issues and to improve the performance of MuT in large-scale systems. The idea is to reduce inter-process communications by using a dynamic and adaptive distribution algorithm that minimises the number of interchanged messages while maintaining a high level of resource usage. Additionally, we propose two different techniques to speed-up several stages of the MuT process, the compilation stage and the detection of equivalent mutants. Hence, this work makes the following contributions:

- Designing a load distribution algorithm that orchestrates the execution of the test cases over the generated mutants. The idea is to maximise the exploitation of the computational resources of the target system, while reducing the inter-process communications, for improving the overall performance.
- Providing a technique for parallelising the compilation phase of the MuT process. Applying MuT on complex applications can lead to a massive generation of mutants that requires the compilation of a high number of programs. This fact causes a bottleneck on the CPU of the system, since all the mutants are required to be compiled at the same time. Hence, it may hamper the feasibility of applying this testing technique. In order to alleviate this issue, we provide an improvement for compiling the mutant set in parallel using the available resources in the system.
- Designing a technique to improve the detection of equivalent mutants, which is currently considered as one of the main limitations for adopting this technique due to the computational waste caused by these class of mutants [14]. Equivalent mutants have the same behaviour as the original program and, hence, their execution does not provide useful information in the analysis of the test suite. This improvement is performed by parallelising the well-known technique called Trivial Compiler Analysis [24].

[1] https://itextpdf.com/.

- Developing a framework for automatically carrying out the MuT process using the approach presented in this paper. This fact alleviates the problem of lack of tool integrations reported by the industry [15]. For this, we have designed and implemented a tool-supported framework that includes all the proposed optimisations—the load distribution algorithm and the two techniques to speed-up the MuT process—and allows to easily deploy and execute the MuT process over large scale systems. Let us remark that this proposal is not focused on a specific MuT engine and, thus, existing MuT tools, such as MuJava [25] and Milu [26], can be easily adapted to be used in this framework. Additionally, this method is automatic in the sense that the intervention of the user is not required to carry out the testing process.
- Conducting a thorough experimental study to evaluate the effectiveness of the proposed approach in large-scale systems for testing different applications. To the best of our knowledge, the largest experiment to date for improving the MuT process uses 80 processes at most [53]. In contrast, our study executes the MuT process using 1024 processes deployed in a cluster consisting of 280 physical machines. For the sake of understanding the scale of the experimental study presented in this article, let us mention that it is up to 7 times larger—if we consider the size of the largest program—12 times larger with respect to the maximum number of nodes, and 153 times larger if we take into account the sequential execution.
- Achieving promising results in the field of cost execution reduction in MuT. The optimisation techniques presented in this work reduce the execution time—in the best case scenario—from up to 10 days to 51 min, which represents a time saving of the 99,66%.

The rest of the paper is structured as follows. Section 2 presents an overview of large-scale systems for testing and introduces the main concepts of MuT. Section 3 analyses the state of the art. Section 4 presents a detailed description of the proposed strategies to improve the MuT process. Section 5 describes a thorough experimental study using the improvements presented in the previous section. The threats to the validity of the experiment are discussed in Sect. 6. Finally, the conclusions and prospects for future work are presented in Sect. 7.

2 Background

This section provides a brief introduction to the main concepts used in this work. Section 2.1 introduces preliminary concepts of parallel execution environments, while Sect. 2.2 provides basic notions of the MuT process.

2.1 Large scale systems

During the last decades, large-scale systems have been continuously evolving, being initially a part of the experimental equipment in laboratories and becoming a day-to-day tool in the life of computational scientists.

The rapid growth of large-scale systems performance can be seen in the TOP 500 ,[2] which is a well-known ranking list of large-scale systems all over the world. Since 1993, the rank list is updated and released twice a year. It creates a solid statistical foundation of research on supercomputing power on Earth. This evolution has been stimulated by several factors, including the physical limits of a single processor generated by heat dissipation and the good price/performance ratio of the computational resources.

These factors have facilitated the emergence of commodity clusters, which are reliable computing infrastructure that consists of commodity computers interconnected through a communication network. Commodity clusters represent more than 80% of all the systems on the Top 500 list and a larger part of commercial scalable systems. In general, these systems are used to reduce the computational time required to execute a program, leveraging the available computational resources of the cluster running the application in parallel. *Cirrus*, which has been selected for running our experiments, is an example of this kind of clusters (see details in Sect. 5).

The Message-Passing Interface (MPI) [27, 28] is the predominant programming model for multi-node communication in scientific computing [29]. Since its conception in 1994, nine versions of the MPI Standard have been released and more than a dozen implementations of MPI exist between open-source and vendor implementations. MPI has three major goals: portability, scalability, and high performance. MPI is able to run on almost every distributed architecture, whether large or small, and each operation is optimized for the specific hardware on which it runs, yielding the greatest speed available. All of this combined, it is easy to see why MPI is the most convenient communication protocol used in commodity clusters.

The message-passing model fits well on separate processors—interconnected through a communication network—to exchange data. MPI is extensively used in the major part of parallel supercomputers and commodity clusters [27, 28, 30]. The most compelling reason that justifies that MPI is an important part of the parallel computing environments is the performance. Hence, it has been shown that MPI is faster than other similar communication mechanisms existing in the literature, such as Java-RMI, RPC and PVM [31, 32].

[2] https://www.top500.org/lists/top500/2021/06/.

2.2 Mutation testing

MuT is a fault-based testing technique, whose primary goal is to measure the effectiveness of a test suite in terms of its ability to detect faults. For this purpose, it provides a testing coverage criterion called the *mutation adequacy score*. Hence, MuT is a feasible alternative to other conventional testing coverage techniques, like statement/branch coverage [33], for evaluating test suites aimed at testing complex systems.

The faults are injected into the code using *mutation operators*, which correspond to rules for transforming the syntax of the language. The idea is to create copies of the original program, where each copy presents some syntactic modification. The objective is to determine the number of program variations, called *mutants*, which behave differently from the original program with respect to a test suite. When a mutant behaves differently from the original program, for some test t, it is said that t *kills* the mutant, otherwise, it is said that the mutant is *alive*.

Let us consider the example of Table 1, where the original program is mutated in such a way that the condition $a > 0 \wedge b > 0$ is replaced by $a > 0 \vee b > 0$. This modification generates a mutant. In order to kill this mutant, it is necessary to create a test that causes a different result in the execution of the mutant with respect to the original program. If we apply the test $a = -3, b = 5$, we obtain different results from the original program and, therefore, the mutant is killed. However, the test $a = 3, b = 5$ does not detect the failure because we obtain the same result in both the original program and the mutant. Hence, the mutants allow evaluating how good the tests are, that is, the more mutants killed by the selected set of tests, the higher the quality of the tests.

Some mutants, called *equivalent* mutants, present the same behaviour than the original program for any input and, consequently, cannot be killed by any test. The detection of *equivalent* mutants is an undecidable problem, so they must be detected manually, which means a high cost in the application of this technique [34]. In order to mitigate this problem, there exist several techniques, such as those proposed in the work of Offutt and Craft [35], where algorithms are detailed to determine different classes of equivalent mutants. These algorithms are based on data flow analysis and compiler optimisation techniques. In another relevant work in this field, Robert Hierons and Mark Harman present the use of *slicing programming* techniques applied to the detection of equivalent mutants, which allow reducing their generation cost [36]. In the same line, Papadakis et al. proposed the Trivial Compiler Analysis (TCE), a technique for detecting equivalent mutants analysing the binary files resulting of the compilation of the source codes [24].

Figure 1 presents the general operation scheme of MuT. Given a set of mutants, generated from applying a set of mutation operators over a program, and a test suite, the adequacy of the test suite is calculated by analysing its ability to detect failures. First, the test suite must be applied to the original program to ensure its correct behaviour ①. If the results are incorrect, the original program must be corrected ②a . Otherwise, the mutation operators are applied to the original program for the generation of all possible mutants ②b . Each mutant will be executed using the provided test suite ③. At the end of this process, the *mutation score* is calculated, indicating the percentage of non-equivalent mutants that have been killed by the test suite. The goal is to achieve a mutation score of 100%, i.e. all non-equivalent mutants have been killed. Alive mutants indicate a lack of adequacy of the test suite to detect potential failures in the program. Therefore, taking the example of Table 1, for a test where $a = 8, b = 7$ both p and p' would have the same result. Hence, p' would be considered as an alive mutant. In this case, the user must analyse whether there exist equivalent mutants and discard them. Then, if after discarding the equivalent mutants, alive mutants still remain, the initial test suite should be increased with additional tests. These tests are directed to kill the mutants that remain alive ④a and should be checked on the original program, too. This process is repeated until the user is satisfied with the *mutation score* obtained ④b .

3 State of the art

Mutation testing is recognised as a computationally expensive technique [15, 16]. Hence, it is necessary to design mechanisms for reducing its cost, and during the last few years the scientific community has proposed different strategies to alleviate this issue. Jia and Offut [19] categorised these proposals in two main groups: *mutant reduction* and *execution reduction* techniques. We review them next.

3.1 Mutant reduction

These techniques are focused on reducing the total number of generated mutants. Hence, in this category, we can find techniques such as mutant sampling [10, 37], mutant clustering [38, 39], selective mutation [40], and high order mutation [41], which reduce the total execution time by decreasing the number of mutants generated during the MuT process.

Mutant sampling randomly selects a small number of mutants from the original set. This technique was proposed by Acree and Budd [10], where they suggested to select a fixed percentage of mutants from the original set. More recent studies show that these techniques start to be effective by selecting, at least, a 10% of the mutants [37].

The total number of mutants can be also reduced by selecting a subset of mutation operators. This technique is known as *selective mutation*, and several studies have been conducted by omitting different number of mutation operators, like 2-selective, 4-selective and 6-selective [42]. The main idea is to select a reduced set of mutation operators which, in some languages, is able to achieve a high percentage of mutation score, and at the same time reducing the total execution time. For instance, in the C language, a subset of 28 operators (out of 108) is able to properly predict the quality of the test suites, with very low effectiveness reduction. Finally, one of the most important techniques to reduce mutants is *High Order Mutation* (HOM). The technique initially presented by Jia and Harman [43] consists of seeding multiple faults in a single mutant. In this line, Polo et al. [41] focused their efforts on analysing the second-order mutants (mutants with two seeded faults), and the empirical results concluded that execution times were decreased up to 50%, with a reduced loss of test effectiveness.

Although these techniques reduce the total execution time by avoiding the execution of mutants, they generally reduce the effectiveness of MuT. In fact, several studies emphasize the need for careful use of these techniques, which—in some cases—has been considered harmful [44]. For instance, Gopinath et al. sentence that *we caution practising testers against applying mutation reduction strategies without adequate justification*.

Table 1 Example of mutated code

Program p	Program p'
.	.
if(a>0 **and** b>0)	if(a>0 **or** b>0)
return 1	return 1
else	else
return 2	return 2
.	.

3.2 Execution reduction

The computational cost of MuT can be reduced not only by decreasing the total number of generated mutants but optimizing the execution process. These techniques can be categorised into three main groups: *mutation type*, *runtime optimization techniques* and *advanced platform support for mutation testing*.

3.2.1 Mutation type

Some cost reduction strategies exploit the mutation type. *Weak mutation* approaches do not require executing the whole mutant, but the execution stops immediately after the mutated statement is processed. In this way, the execution costs are reduced in comparison with the traditional mutation—known as *strong mutation*—where the complete mutant is executed.

The experimental study conducted by Chekam et al. [45] showed that the propagation—not present in weak mutation—is responsible for finding 36% of the failures that can be detected by strong mutation. For this reason, there is a potential risk of significant information loss during the testing process.

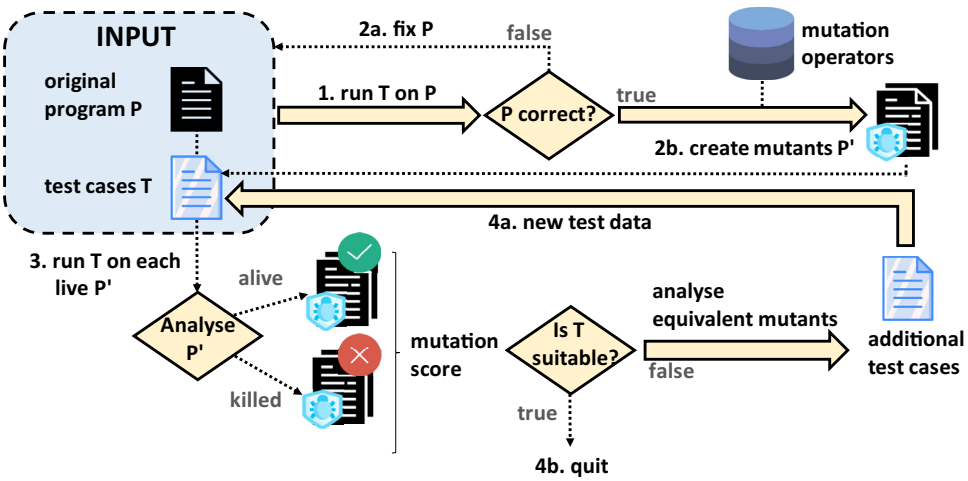


Fig. 1 General scheme of the MuT process

3.2.2 Run-time optimization techniques

One of the first optimisations oriented to reducing cost in mutation tools is the *interpreted-based* technique [46]. This technique interprets the mutant output directly from the source code, rather than executing it. However, the approach does not scale well due to the high cost of interpretation.

Mutant schemata aims at reducing compilation costs [47, 48]. Instead of creating a file for each mutant, this technique allows creating a metaprogram that contains all the faults seeded by the mutation operators. In this way, in the compilation phase, only a single file must be compiled, which saves a high percentage of compilation time. However, some mutation operators produce mutants that cannot be combined into a mutant schema (like AMC, IOD, and IPC [49]), and in addition, a configuration phase is necessary to execute this technique. This phase can be complex, and some environments must be manually modified for the sake of the proper functioning of the technique. Finally, since Mutant Schemata only generates a single binary file, some valuable techniques for detecting equivalent mutants, like TCE, can not be applied. It is worth mentioning that TCE can discard on average between 7 and 21% of all the mutants as being equivalent [24]. Previously, it was necessary to choose between reducing the compilation time by using Mutant Schemata or detecting equivalents through TCE. However, using the techniques proposed in this article it is possible to perform both optimisations at the same time.

3.2.3 Advanced platforms support for mutation testing

In order to improve the speedup, our main idea is to distribute mutants to processes in such a way that each process completes a similar amount of execution. One of the main issues of this technique is to predict the number of test cases required to kill a mutant. Therefore, considering the complexity of predicting the total number of executions needed to complete the MuT process, developing a technique without knowing these details is challenging.

During the last years, several distribution algorithms have been proposed to alleviate this issue. These distribution algorithms can be categorised into two main groups: static and dynamic distribution algorithms. We review them next.

The static distribution algorithms assign—approximately—the same number of mutants to each process. The particularity of this kind of algorithms lies in the fact that the division of the workload, that is, the test cases that must be executed over the mutants, is carried out before starting the MuT process. Among the best-known approaches is the Distribute Mutants Between Operators (in short, DMBO) algorithm [50]. This was proposed by Offut et al., and categorises the mutants by MuT operator, assigning equally to each process different mutants from each group. Hence, each process receives roughly the same quantity of live mutants and the MuT operators are as evenly distributed as possible. Distribute Test Cases (in short, DTC) [51] is a static algorithm similar to DMBO. Nevertheless, in this case, the test suite execution is divided into the available processes. In this way, each process executes the same test cases over all the mutants. This fact causes that several mutants could be killed multiple times by different test cases, leading to a waste of computational resources. Taking into account that the distribution of the workload is carried out only once, the static algorithms reduce the network communications. However, in heterogeneous environments, these algorithms hamper the achievement of the optimal distribution, since faster CPUs are more likely to finish before the slower ones, unbalancing the distribution process.

The dynamic distribution algorithms focus on optimising the distribution of the workload among the available processes while reducing the execution time. In such way, the workload is divided into portions, which are dynamically delivered to the available processes until all the workload is entirely processed. Give Mutants On Demand (in short, GMOD) [52] is a dynamic algorithm that divides the executions into parts, where each part contains one mutant and all the test cases. This algorithm increases the network communications, because the process in charge of coordinating the execution of the testing process should deliver the parts of the workload to the processes until all the workload is processed. The Give Test Cases On Demand (in short, GTCOD) is a dynamic algorithm that splits the workload into parts, where each one consists of one test case with all the mutants. Similarly to DTC, this algorithm may cause that several mutants are killed multiple times. The Parallel Execution with Dynamic Ranking and Ordering (in short, PEDRO) [53] is a dynamic algorithm based on the ideas of the Factoring Self-Scheduling algorithm [54], but redesigned to be applicable on MuT. PEDRO divides the workload into parts, where each part contains a set of mutants and all the test cases. EMbarrassINgly parallEl mutatioN Testing (in short, EMINENT) [23] is a dynamic algorithm where the workload is divided into different parts, each one consisting of a single test case, in contrast to the PEDRO algorithm, where the parts consist of the complete test suite. This algorithm is considered as more adaptable to heterogeneous environments and allows to maximise the resource usage and, thus, the overall time of the testing process is reduced. Optimizing the mUtation Testing pRocess In Distributed EnviRonments (in short, OUTRIDER) [20] maximises the resource usage using EMINENT as a basis. For this, the

authors propose an enhanced distribution algorithm and three techniques for improving the performance: parallelising the test suite, sorting the test suite and detecting equivalent mutants.

During the last years, these distribution algorithms have been included in several solutions proposed by researchers to alleviate the high computational cost associated with the testing process. *Bacterio*^{P} [21] is a MuT framework for multi-class Java systems, which supports strong mutation, BB-Weak/1, BB-Weak/N, functional qualification and flexible weak mutation and uses the PEDRO algorithm to distribute the workload. *Bacterio*^{P} uses Java-RMI [55] to communicate processes through the network. The performance and the scalability achieved in this contribution are better than previous studies, dated from mid-1990s, which are based on shared-memory approaches. However the communication mechanism used in this approach acts as a system bottleneck and, consequently, causes a limitation on the overall system performance [31].

HadoopMutator is a cloud-based MuT framework, proposed by Saleh and Nagi, which relies on Map-Reduce ideas to parallelise the MuT process [56]. HadoopMutator is based on the Hadoop engine and Pitest, and uses a static distribution algorithm to distribute the workload. Therefore, since the inclusion of dynamic distribution schemas in this proposal is not considered, it is not suitable to be used in heterogeneous and dynamic environments.

OUTRIDER-tool [20, 23] is a framework for improving the performance of MuT in HPC systems. The main goal of this solution is to exploit the resource usage in HPC systems by using MPI to communicate the processes. The main disadvantage of this proposal concerns the high quantity of messages exchanged between processes, which may cause, in some cases, a bottleneck in the communication network.

DiMuTesTas is a solution for improving the performance of the MuT process using a cloud infrastructure [22]. In this proposal, the master and the worker processes are encapsulated into Docker containers [57], using RabbitMQ [58] for orchestrating the communication between them. This approach uses a dynamic distribution where the workload is divided in parts, which consist of the complete test suite applied over a single mutant. Although DiMuTesTas has been designed to minimise both the idle-time of the nodes and the network load, there is some point that needs further discussion. The authors sentence that *DiMuTesTas copies the build dependencies from the file server to the own local storage in each worker. This is done to prevent the network connection from becoming a bottleneck, as otherwise each worker has to download the build dependencies separately.*. This fact might lead to different drawbacks. Firstly, the MuT environments usually consist of thousands of files, that require a large amount of storage. Secondly, there exist several HPC systems whose local storage systems is virtual or non-existent and, therefore, this fact hampers the usage of the proposed schema on a certain type of system.

4 Large scale mutation testing

In this section, we explain our approach to parallelise the MuT process. First, we present an overview in Sect. 4.1. Then, we describe the three main elements in the method: a distribution algorithm to improve the execution of the MuT process (Sect. 4.2), an optimisation based on parallel compilation (Sect. 4.3) and the parallelisation of the detection of equivalent mutants (Sect. 4.4). Finally, Sect. 4.5 details tool support.

4.1 Overview of our approach

The main efforts conducted by the research community in the MuT field have targeted reducing the computational cost of the process [59, 60]. However, while the parallelisation of MuT to increase the speed-up has obtained good results, it has not been investigated much, according to Papadakis and collaborators in a recent survey [16].

The MuT process involves several stages, such as generating tests cases and mutants, compiling mutants, detecting equivalent mutants, executing the test cases and comparing the obtained results. Parallelising this process requires deploying processes among the CPU cores of the target system and executing each stage by efficiently exploiting the shared resources of the system, like CPUs, network, and storage systems. In particular, compiling mutants, detecting equivalent ones and executing the test cases are considered the most CPU-expensive stages of the MuT process [16, 19], and hence our method focuses on these three phases.

The parallelisation of the execution stage can be achieved by distributing the execution of the test suite over the mutants among the processes involved in the MuT process, in such a way that the test cases are executed in parallel. The key aspect to achieve a good performance lies in optimising the number of operations assigned to each process. Hence, we define the *grain size* as the maximum number of test cases that must be executed by a CPU process before checking if the MuT is completed. Establishing a proper grain size involves a trade-off between maximizing the exploitation of computational resources and reducing inter-process communication.

Depending on the load distribution algorithm used, the grain size can vary. Among the best-known execution grains are the *mutant-level grain* [53] and the *test-level*

grain [22, 61]. The former assigns the execution of the entire test suite—over a set of mutants—to a single process. In contrast, the latter assigns the execution of one test case—over a mutant—to a single process. In Sect. 4.2 we will present a dynamic and adaptive technique that combines these two strategies in order to maximize CPU usage.

Reducing the cost associated with the compilation phase requires parallelising the compilation of the mutant set among the available CPU cores of the system. Section 4.3 explains our approach for this parallelisation. Similarly, in order to speed-up the detection of equivalents, it is necessary to parallelise the equivalence detection technique used by maximizing the exploitation of computing resources. In our case, we use the Trivial Compiler Equivalence (TCE, in short), and its parallelisation will be described in Sect. 4.4.

4.2 Dynamic and adaptive load distribution algorithm

One of the key aspects to improve the overall performance of MuT lies in the distribution of the test cases, which must be executed among the available CPU cores of the target system. For this, the *grain size* is a trade-off between wasting computational resources—when a large grain size is established and, at the end of the MuT process, some CPUs are idle—and saturating the network—when the grain size is small and a large number of messages must be sent between processes.

This way, the distribution algorithm, which allocates tests on mutants to the processes, has a substantial influence on the overall execution time of the process. A wrong selection of the distribution strategy may lead to bottlenecks or cause a low usage of the available computational resources. The former case occurs when there is a massive exchange of messages caused by a small grain size. It should be noted that the smaller the grain size, the higher the number of messages exchanged. Hence, selecting a small grain size increases the network traffic and can lead to network bottlenecks. The latter case occurs when some computational processes finish their associated tasks—becoming idle—while other computational processes continue processing tasks. Therefore, we propose several improvements to alleviate these issues.

On the one hand, in the mutant-level grain, each processor executes the whole test suite over several mutants. On the other, in the test-level grain each processor executes a single test case over a single mutant. Although both execution grains have led to performance improvements [20, 23, 53] they also present some limitations: the mutant-level grain is not efficient in the final stage of the process, and the test-level grain increases the communication traffic.

To reduce the network communications—in a dynamic approach—the most suitable grain is the mutant-level. This is because it divides the workload in larger portions (where each portion consists of a test suite), and so it reduces the number of grains that must be sent to each processor, reducing the number of messages exchanged between processes. However, when the number of mutants remaining is less than the number of available processes, the percentage of idle processes increases, wasting computational resources. This fact is especially challenging when the program under study has associated a large test suite, and the total time required to execute each test case is high.

To maximise the number of active CPUs during the process, the test-level grain works better. In this case, regardless of the remaining workload to be processed, it minimises the time during which the computational cores are idle. Since each process only executes one test over a mutant, the number of portions that must be distributed to the available processes increases with respect to the mutant-level grain. Unfortunately, this strategy generates high network traffic. Dividing the workload in smaller portions increases the number of grains that must be sent to each processor, and consequently, the number of messages exchanged between processes is increased.

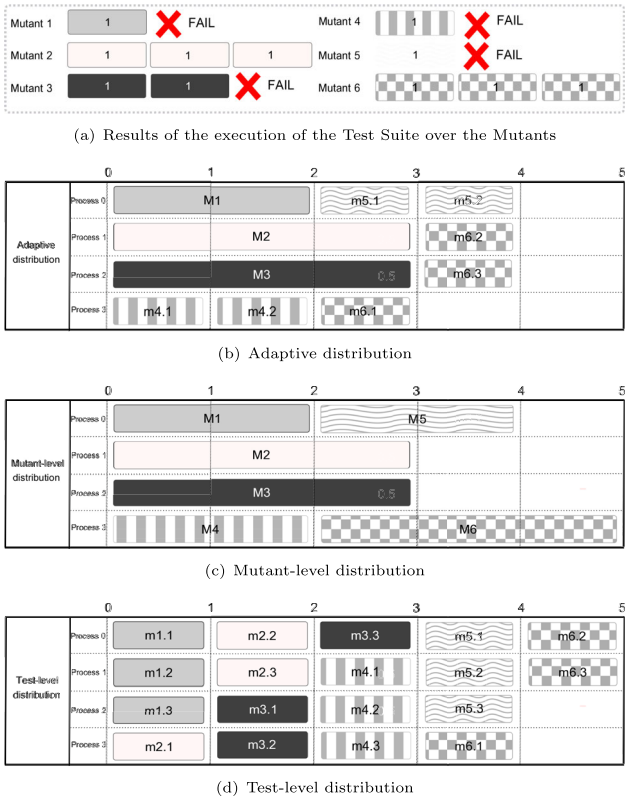
Therefore, our proposal combines the benefits of both execution grains, minimising the exchange of messages between the master and the worker processes, while maximising the exploitation of the available resources by reducing the time during with the CPUs are idle. For this purpose, we have designed an algorithm that combines the mutant-level and test-level execution grains. This proposal is novel since the most relevant works related to the parallel MuT field do not explore the combination of execution grains [52, 53, 62]. In the literature, we can find several works that using a single execution grain obtain suitable results in terms of scalability during the experimentation phase [20, 23, 53]. Since none of them target large-scale systems, the drawbacks detected in these proposals do not lead to performance degradation due to the small size of the execution environment used in the experiments. However, their application at a larger scale would be problematic.

Algorithm 1 presents the main steps of the adaptive distribution algorithm proposed in this paper. In this approach, two different processes are involved: master and workers. The *master* coordinates the workers, that is, assigns the tasks that must be achieved by the workers (lines 3–8 and 17–26), while the *worker* processes execute the tasks provided by the master (lines 10–15). The master calculates the total number of mutants and tests (lines 5 and 6), and performs the load distribution. This step is conducted in two phases, the initial phase where all workers

are served with an execution block (lines 5–6). In the second phase, once all the worker processes have received an initial load, the master process waits for the workers to finish processing their execution block, for assigning a new one, until all the remaining blocks of the workload have been processed (lines 7–8). This fact maximises the usage of the computing environment, especially in the case of heterogeneous processes. In order to maximise the distribution process, the master process must select the execution grain to be used (lines 17–26). For this, it analyses whether the number of processes is less than the number of remaining mutants (line 18). In that case, it selects the mutant-level grain (line 19). Otherwise, when the number of mutants to be processed is less than the number of worker processes, the master process selects the test-level grain, providing a single test case to execute on a mutant, until the process is completed (lines 21–25). This grain size seeks to maximise the number of active processors of the environment, since in the case of further using the mutant-level grain when the number of remaining mutants is less than the available processors, those processors that finish the execution of its associated tasks become idle.

Algorithm 1 Adaptive distribution algorithm

```
   Data: testSuite, mutantSet
1  Function distributionAlgorithm:
2      numProcs = MPI_Comm_size() myId = MPI_Comm_rank() // Master process
3      if myId == MASTER then
4          mutantSize = getMutantSize(mutantSet) testSuiteSize = getTestSuiteSize(testSuite) // Initial distribution
5          while i<numProcs and currentMutant < mutantSize do
6              execBlock = selectGrain(currentTest, currentMutant) MPI_Send (execBlock, i) i = i+1
           // Distribute and process the remaining mutants
7          while remainingBlocks(currentTest, currentMutant)> 0 do
8              result = MPI_Recv (ANY, status) execBlock = selectGrain(currentTest, currentMutant) MPI_Send (execBlock, status.MPI_SENDER)
9      else
           // Worker processes
10         continueProcessing = true while continueProcessing do
11             MPI_Recv (execBlock, MASTER)
12             if execBlock.continueProcessing then
13                 result ← execute (execBlock) MPI_send (MASTER, result)
14             else
15                 continueProcessing = false
16     return 1
17 Function selectGrain(currentTest, currentMutant):
18     if if numProcs < getRemainingMutants (currentMutant, mutantSize) then
           // Selects the mutant-level grain
19         execBlock = buildExecBlock (currentMutant, 0, testSuiteSize) currentMutant = currentMutant +1
20     else
           // Selects the test-level grain
21         execBlock = buildExecBlock (currentMutant, currentTest, currentTest+1)
22         if currentTest+1 < testSuiteSize then
23             currentTest = currentTest +1
24         else
25             currentTest = 0 currentMutant = currentMutant + 1
26     return execBlock
```



(a) Results of the execution of the Test Suite over the Mutants

(b) Adaptive distribution

(c) Mutant-level distribution

(d) Test-level distribution

Fig. 2 Workload distribution using the adaptive, mutant-level and test-level distributions

Figure 2 shows the differences between using distributions that apply the test-level, mutant-level and the adaptive grain we propose. Each distribution is analysed by using 4 worker processes for testing six different mutants, which are analysed with a test suite consisting of three tests cases. The mutants 2 and 6 pass all the test cases in 3 units of time ($t_{c1} = 1$, $t_{c2} = 1$, $t_{c3} = 1$), mutants 1, 4 and 5 fail the second test in 1 unit of time and mutant 3 fails the third test case (see Fig. 2a). We denote by mX.Y the execution of the test case Y over the mutant X, and MX is the execution of the whole test suite over the mutant X. In this scenario, the test cases corresponding with the executions m1.2, m3.3, m4.2, and m5.2 fail, and consequently all of them kill the processed mutant.

The distribution that uses adaptive grain (see Fig. 2b) starts using the mutant-level grain. The first process executes M1 (all the test cases executed over mutant 1) and in parallel, the processes 1 and 2 execute M2 and M3, respectively. At this point, the number of remaining mutants to be processed is less than the processes and, therefore, the grain size is adapted to test-level. The process 3 executes m4.1 and m4.2. Then, when no more tests are available to execute over mutants 1 and 4, the processes 0 and 3 execute m5.1 and m6.1, respectively. Finally, processes 0, 1 and 2 execute m5.2 and m6.2 and m6.3, respectively. The total time required to process the whole test suite over the mutants is 4, and the total number of exchanged messages is 10.

With respect to the approach that applies the mutant-level grain in the distribution (see Fig. 2c) the processes 0, 1, 2 and 3 execute, in parallel, M1, M2, M3 and M4, respectively. Once all test cases have been applied over M1 and M4, the processes 0 and 3 execute M5 and M6, respectively. The total time elapsed is 5 time units, and the total number of messages exchanged between the processes is 6.

The distribution involving the test-level grain is based on the idea that the computational efficiency may decrease if the number of remaining mutants is lesser than the available CPUs. In this way, all the test cases are executed in parallel in different processes, as depicted in Fig. 2d. However, several test case executions could have been saved. In such way that, for example, the mutant 1 (m1.1 m1.2 and m1.3), mutant 4 (m4.1, m4.2, m4.3) and mutant 5 (m5.1, m5.2, m5.3) are executed in parallel at the same unit of time. Therefore, it was not detected until the next unit of time that the mutants 1, 4 and 5 had been killed in the tests m1.2, m4.2 and m5.2, so that the execution of the tests m1.3, m4.3 and m5.3 was indeed not necessary. The total time required to process the whole test suite over the mutants is 5, and the total number of exchanged messages is 18.

In this particular example, it can be observed that, in terms of time, the adaptive grain saves a 20% of the time compared to using the mutant-level and test-level grains. With respect to the total quantity of messages exchanged between processes, the mutant-level grain saves 66% and 200% of messages compared with adaptive and test-level grain, respectively. It is worth mentioning that, for the sake of simplicity, the overhead generated by inter-process communications has been omitted. Hence, the total time required to execute the test-level distribution algorithm in a real environment would be increased in comparison to the adaptive and mutant-level grains. In summary, the adaptive grain achieves a good compromise between the percentage of computational resources usage and the number of messages interchanged during the MuT process.

4.3 Parallel compilation

Since generating a high number of mutants is necessary for achieving a high level of reliability, MuT is considered a computationally expensive testing technique. In addition to the time required to execute the test suite over the mutant set, the time required to compile all the mutants must also be taken into consideration. Hence, it has been recognized that the compilation phase has an adverse effect on the speedup [52]. Let us remark that the compilation is

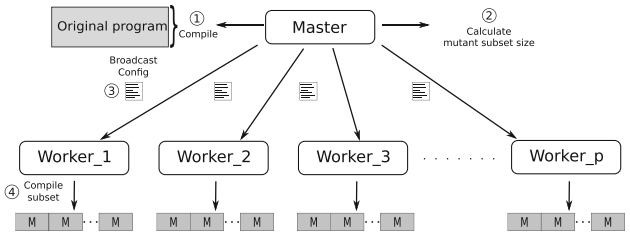


Fig. 3 Scheme of the parallel compilation

performed at runtime, as part of the MuT process and therefore, it contributes to the total execution time of the MuT process.

For illustration, consider again the *iText* application mentioned in the introduction. The entire mutant set consists of 126,781 mutants and the compilation of each mutant takes an average of 2.1 s. In this case, the time required to complete the compilation phase, using a single processor, exceeds 73.95 h.

In order to mitigate this issue, we propose to use all the available processes in the compilation phase. In this way, since the processes do not need to send back data to the process that coordinates the compilation, the overall time is reduced—approximately—as many times as processes are available.

Figure 3 illustrates the parallel compilation. Initially, the master process compiles the original program ① and calculates the number of mutants that must be compiled by each worker process ②. The main idea is to provide to each worker the same quantity of mutants to compile. Therefore, this quantity is calculated as the ratio between the number of mutants and the total number of worker processes. Then, the master process sends the range of mutants to the workers ③. At this point, the workers sequentially compile the mutants that correspond to the provided range ④. Finally, the master process waits, until all the worker processes have finished the compilation task, to continue the MuT process.

4.4 Parallel detection of equivalent and duplicated mutants

Discerning whether two programs are equivalent is an undecidable problem [34]. Hence, within the scheme of MuT, this implies an additional human effort to detect equivalent mutants. It is worth mentioning that testing this kind of mutants requires a high quantity of computational resources. This is because their behaviour is identical to the original program, and so, in order to test these mutants the complete test suite needs to be executed. Moreover, these executions do not provide additional information to the MuT process, since this kind of mutants cannot be killed. In fact, some studies show that—depending on the program and the selected mutation operators—the percentage of equivalent mutants generated in a MuT process can range from 17 to 48% [63], which can be considered a high waste of time and computational resources.

For these reasons, this problem is considered as one of the main drawbacks for the adoption of MuT by the industry [13]. During the last years, several heuristics have been proposed to alleviate this problem. One of the most relevant—due to its simplicity and effectiveness—is TCE [24], which is a simple but powerful technique for detecting both equivalent and duplicated mutants. This technique consists in comparing the compilation binaries of the original program and the mutant. Its rationale is that compiler optimisations may result in the same compiled binaries for several syntactically different programs that perform the same function. However, the application of these heuristics requires a long execution time, usually due to I/O bottlenecks, which limits the overall performance of MuT.

The complexity of applying detection heuristics is shown in the *iText* application, where applying the TCE technique over the 126,781 mutants could take an overall time of 7 min (around 0.0033 s per mutant). In order to face this challenge, we focus on speeding-up the MuT process by avoiding the total time needed to test the detected equivalent mutants, while reducing the time required to apply the equivalent detection techniques. Thus, we propose to adapt the scheme of the TCE for analysing the mutant set to be executed over large-scale systems.

Figure 4 illustrates the main scheme of the proposal. Initially, the master process calculates the checksum of the original program ①. Then, the master splits the mutant set in parts ②, dividing the set equally between the number of worker processes, and distributes the parts among the workers ③. At this point, each worker receives a subset of mutants and sequentially applies the TCE technique to all the mutants assigned ④. Once the TCE has been completely applied, the workers send a list of checksums corresponding to each assigned mutant ⑤. Finally, once the master process receives all the checksum lists ⑥, it performs two analyses: the comparison between the checksum

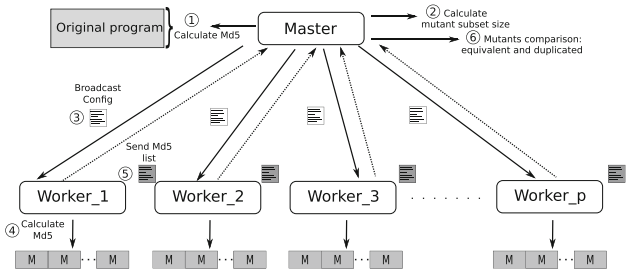


Fig. 4 Scheme of the parallel TCE

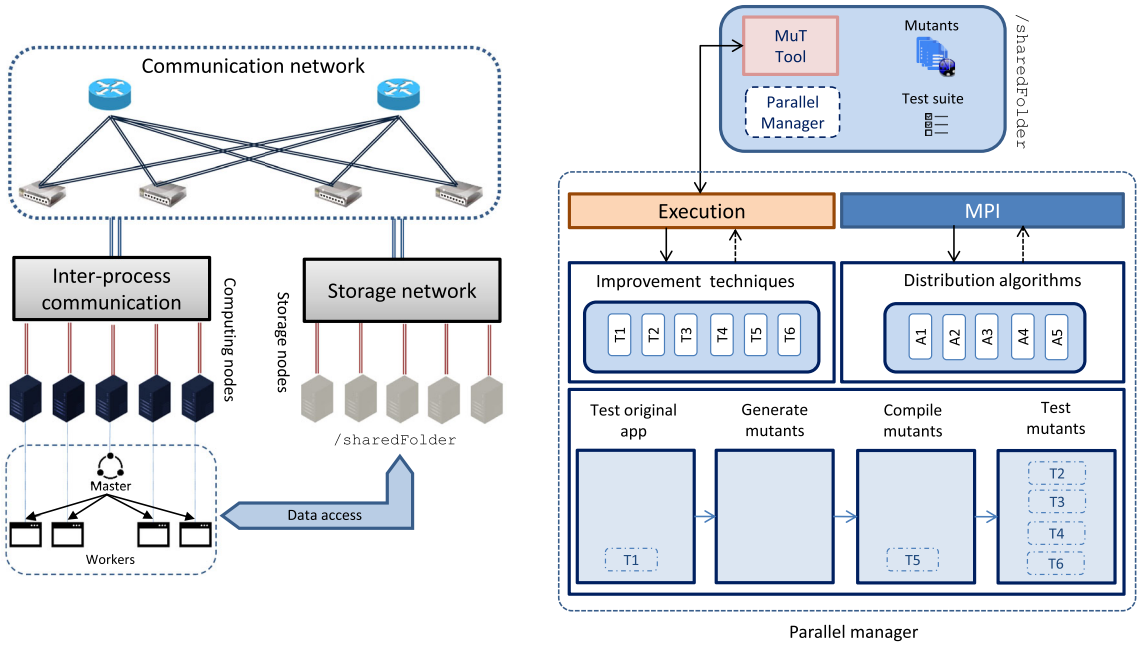


Fig. 5 Tool support for the parallel MuT for large scale systems

of the original program and the checksum of all the mutants, and the comparison between checksum of all the mutants between them. The former targets detecting equivalent mutants, and the latter detects duplicated mutants, that is, mutants that are equivalent between them.

4.5 Tool support

We have developed a tool, called *Parallel manager*, which realizes the framework proposed in this work to automatically conduct parallel MuT. Figure 5 shows the basic architecture of *Parallel manager*. The left part depicts how the master and worker processes are deployed among the available computing nodes of the cluster. These nodes must contain a shared folder that is allocated in the different storage nodes of the cluster. Let us remind that the access to the data heavily depends on how the file system of the cluster is configured. Thus, parallel and distributed file systems should significantly improve the overall system performance. In essence, the shared folder stores the *Parallel manager* tool and the MuT tool to create the mutants and the test suite, which are also stored in the same folder.

The main phases of the MuT process are managed by this tool and are depicted at the right of the figure, namely, executing the test cases over the application under study, generating mutants, compiling the mutants, analysing equivalent mutants, executing the test cases over the mutants and comparing the obtained results with the ones obtained in stage to determine whether the test cases can detect the mutants.

In addition to the improvements proposed in this paper, *Parallel manager* includes some of the most important techniques and distribution algorithms existing in the current literature. With respect to the distribution algorithms, the tool includes both static and dynamic strategies, an adapted version of DMBO [52], DTC [51], PEDRO [53], EMINENT [23], and the *Dynamic adaptive algorithm* (labelled as *A1*, *A2*, *A3*, *A4* and *A5*, respectively). Regarding the improvement techniques, the tool includes *distributing the test suite among the available processes* (T1), *sorting the test suite before executing test cases* (T2), *scatter improvement* (T3), *clustering equivalent mutants* (T4) [20], *parallel compilation* (T5) and *parallel checksum* (T6).

It is important to remark that *Parallel manager* is independent from the mutation engine used to generate both mutants and test suites, such as MuJava [25] and Milu [26], which can be easily included in this framework. The unique requirement that the MuT tool must fulfil is command line support. In this way, *Parallel manager* invokes the MuT engine via command line for carrying out different operations, such as creating the mutants set and executing the test suite over the mutants. The execution of the test cases can also be conducted in standalone mode, that is, the test cases—once created—are executed directly by *Parallel manager* without using an additional MuT tool.

Parallel manager has been designed to be used in large HPC environments. For this purpose, the tool uses MPI to interchange data between processes. The logical process schema is shown at the bottom-left of Fig. 5, and is based on a master/worker pattern. In this way, the master process orchestrates the MuT process and distributes the workload among the workers, following one of the distribution algorithms supported by the tool (A1, A2, A3, A4 or A5). Finally, the tool also supports sequential MuT execution—not based on a master/worker pattern, but on a single process—which can be used to calculate the speed-up obtained in the parallel execution.

5 Experiments

This section presents a thorough empirical study to show the efficiency of our approach for improving the performance of the MuT process in large-scale systems. Firstly, in Sect. 5.1, we formulate the research questions we aim to answer. Next, Sect. 5.2 describes the experimental setting and procedure of the study. The results obtained are presented in Sect. 5.3. Finally, we discuss the results and answer the research questions in Sect. 5.4.

5.1 Research questions

Our empirical study seeks to answer the following research questions (RQs):

RQ1 *How suitable are the current distribution algorithms to parallelise the MuT process?* Evaluating large test suites for testing real-world applications requires vast amounts of computational power. Currently, there exist several distribution algorithms that exploit the parallelism of multi-core systems—and small clusters—to reduce the time required to complete this task. However, we think that these algorithms do not consider the potential drawbacks that may appear when the MuT process is executed over a distributed system using a large number of processes (see Sect. 3). Hence, we formulate the hypothesis that the current distribution algorithms are not suitable to be applied in large-scale system and, thus, we investigate how suitable are them for parallelizing the testing process.

RQ2 *How effective is our proposed distribution algorithm to improve the overall performance of the MuT process in large-scale systems?* The proposed distribution algorithm has been designed to maximise the exploitation of the shared computational resources—of the target system—and to minimise the inter-process communications, avoiding system bottlenecks and speeding-up the execution of the MuT process. We are interested in studying the scalability and efficiency of this algorithm when it is executed among the available CPU-cores of a large-scale system. To answer this question, we have designed a wide spectrum of experiments where both an existing algorithm, and our approach, are deployed and executed in a large-scale system using different configurations.

RQ3 *What techniques are appropriate to improve the performance of the MuT process in large-scale systems?* In essence, the MuT process includes different tasks, like generating the test cases and the mutants, compiling the mutants, and executing the test cases over the mutants. Different techniques can be applied to speed-up the execution of these tasks, hence improving the overall performance of the MuT process. However, some of these tasks can be significantly improved when the MuT process is executed in large-scale systems. Since we differentiate the critical stages of the testing process to locate potential bottlenecks, that is, compiling the mutants and executing the test cases, we are interested in investigating how our two proposed techniques, focusing on improving the compilation of mutants and detecting equivalent mutants, impact on the overall performance of the MuT process when it is executed in a large-scale system. Additionally, we study how the current techniques improve the overall performance in large-scale systems.

5.2 Experimental setting and procedure

In order to check the scalability and performance of our approach, we have carried out a thorough performance study, for evaluating test suites, aimed at testing three different real-world applications written in C. These applications have been chosen to cover some of the most representative computational paradigms (data-intensive and CPU-intensive) [64-66]. The MuT process has been conducted by using the core of MuTomVo [5], a framework for mutating C applications, which has been adapted to run as a standalone application in cluster environments.

Table 2 Summary of the applications and environments used to carry out the MuT process

Application data			Test suite data			MuT process results			
Name	Execut. size (kb)	LOC	Env. size (Gb)	Seq. time (s)	TS size	Mutants	Killed	Equiv	MS (%)
Filter	12	500	1.7	3570	1000	250	152	16	65
Bzip2	212	7000	3.2	320,740	800	13,000	4060	5000	51.6
FFT	354	13,500	48	919,825	975	75,000	31,723	28,896	68.8

Table 2 summarizes some features of the three applications used in this study, where the first three columns show the name, size of the executable file, and the number of lines of code (LOC), respectively. The next three columns refer to the test suite environment, where *Env. Size* represents the size of the testing environment, which involves the input files required to execute the tests and the executable files of the mutants, *Seq. Time* is the time to completely execute the MuT process sequentially, that is, one process—using the framework proposed in this work—executes the test suite over the mutants using one CPU-core, and *TS Size* is the number of test cases in the test suite. The last four columns report the MuT process results, where *Mutants* is the number of mutants generated to evaluate the test suite, *Killed* is the number of mutants that have been detected with the test suite, *Equiv* is the number of equivalent mutants in the test suite, and *MS* is the mutation score used to measure the effectiveness of a test suite in terms of its ability to detect faults.

The first application (*Filter*) is a basic image filtering program that transforms greyscale images into black and white images. This filter is applied over BMP images [67]. In essence, this application reads the input image, applies the filter to transform each input—greyscale—pixel into an output—black or white—pixel, and writes the filtered image to disk. This application consists of 500 LOC and its executable is 17 kBytes. In order to apply the test cases, we have generated a collection of 800 different BMP greyscale images. The total size of these images is 1.7 GBytes. The second application is the *Bzip2* compressor,[3] which uses the Burrows–Wheeler block sorting text compression algorithm and Huffman coding to perform the compression process [68]. This application has 7000 LOC and, once compiled, its executable size is 212 kBytes. Basically, this application loads a file stored in disk and compresses it contents. The compressed data is stored in memory and, therefore, new files are not created in the file system. Similar to the previous application, we have generated a collection of 800 input test files that requires 1.7 GBytes of disk space. The executable files of the 13,000 mutants require 1.5 GBytes of disk space. The third application (*FFT*) is an open source library that calculates the Fast Fourier Transform (in short, FFT).[4] This application consists of 13,500 LOC and its executable is 354 kBytes. This application does not require input files for executing the test cases, but input parameters are passed through the execution script. The total size required to store the executable files of the 75,000 mutants is 48 GBytes.

To evaluate the test suites for testing these applications, we have generated one mutant set for each application (see column *Mutants* in Table 2). These mutants have been created by seeding faults in the original code of these applications using the MuToMVo framework [5]. The idea is to execute the test cases over the original applications, and over the mutants, analysing the obtained outputs, and checking if the test cases can detect the mutants, or not. It is worth mentioning that the results shown in Table 2 have been obtained without any improvement applied.

These experiments have been performed in *Cirrus*, which is a state-of-the art SGI ICE XA system with 280 compute nodes, and Lustre as the file system. *Cirrus* runs the CentOS Linux distribution, and has a range of software, libraries and tools available.[5] *Cirrus* standard compute nodes contain two 2.1 GHz, 18-core Intel Xeon E5-2695 (Broadwell) series processors. Each core supports two hardware threads (Hyperthreads), which are enabled by default. The standard compute nodes on *Cirrus* have 256 GB of memory shared between both processors.

Since we are interested in investigating how suitable is our approach to speed-up the MuT process, we first need to calculate the time required to evaluate a test suite using a sequential execution. This value—measured in seconds—is shown in the column labelled as *Seq. Time* of Table 2.

In this study we use two load distribution algorithms for executing the MuT process. The first algorithm assigns tasks—to each process—consisting in executing one test case over a single mutant. Since the test cases are assigned to each process individually, this algorithm requires to send and receive a high number of messages between processes.

[3] Available at: https://sourceforge.net/projects/bzip2/.

[4] Available at: https://www.kurims.kyoto-u.ac.jp/~ooura/fft.html.

[5] https://www.cirrus.ac.uk/about/.

Table 3 Summary of the techniques applied to improve the different stages of the MuT process

Source	ID	Stage	Description
N/A	T_0	N/A	No improvement is applied
Existing techniques	T_1	Test execution	Distributing the test suite among the available processes
	T_2	Test execution	Sorting the test suite before executing test cases
	T_3	Test execution	Scatter distribution
	T_4	Detection of equivalent mutants	Clustering equivalent mutants
New proposed techniques	T_5	Compilation	Parallel compilation
	T_6	Detection of equivalent mutants	Parallel checksum

From now on, we refer to this algorithm as *Test-level slicing algorithm*. The second algorithm used in this study is the one we propose in this paper, which we call the *Dynamic adaptive algorithm*. Basically, this algorithm has been designed to maximise the exploitation of the computational resources of the target system while reducing the inter-process communications.

In order to check the scalability these algorithms, the MuT process has been executed using a different number of processes over *Cirrus*. It is important to mention that the processes are executed in a dedicated CPU-core and, thus, there is no competition for obtaining the CPU. Additionally, we are interested in investigating the suitability of the existent techniques—focused on improving the different stages of the MuT process—to speed-up the execution of the MuT process in large-scale systems. Table 3 summarises the techniques investigated in this study, where the first column represents the source of the strategy, the second column represents the strategy ID, and the third column shows a description of the technique. The first row (T_0) is the baseline strategy, where no improvements are applied, that is, the test cases are assigned to each process using a load distribution algorithm and no additional techniques are applied during the MuT process. The next four rows (T_1–T_4) represent strategies that have been analysed in the past [20]. T_1 parallelises the execution of the test suite over the original application, T_2 sorts the test suite by using the execution time of each test case as sorting criteria, T_3 is the technique that enhances the test case distribution maximising the number of mutants that are being analysed at the same time, that is, with this improvement each single process executes single test case over a different mutant. T_4 implements the TCE technique to detect equivalent mutants.

The next two rows represent our proposed techniques, namely T_5 and T_6, which focus on parallelising the compilation of mutants and detecting equivalent mutants, respectively. These strategies were described in detail in Sects. 4.3 and 4.4, respectively. T_5 compiles the mutant set in parallel and T_6 parallelises the detection of equivalent mutants.

Overall, the procedure used in this empirical study is the following:

(i) First, we choose an application to apply the testing process. In this case we use the applications shown in Table 2.
(ii) For each application, we create a test suite and a set of mutants.
(iii) The test suite is executed over the application under test.
(iv) The test cases are sequentially executed over each mutant. Once finished, the mutation score is calculated. The time required for this task (*Seq. Time*) is taken as the baseline to measure the speed-up when the testing process is executed in a distributed environment.
(v) The MuT process is executed—for each application—in a distributed system using a different number of processes, ranging from 32 to 1024. Additionally, different techniques (those in Table 3) are applied in different stages of the MuT process to analyse its suitability for improving the overall performance in large-scale systems.
(vi) Finally, the results are analysed and discussed in detail.

5.3 Measuring the quality of test suites by applying mutation testing over large-scale systems

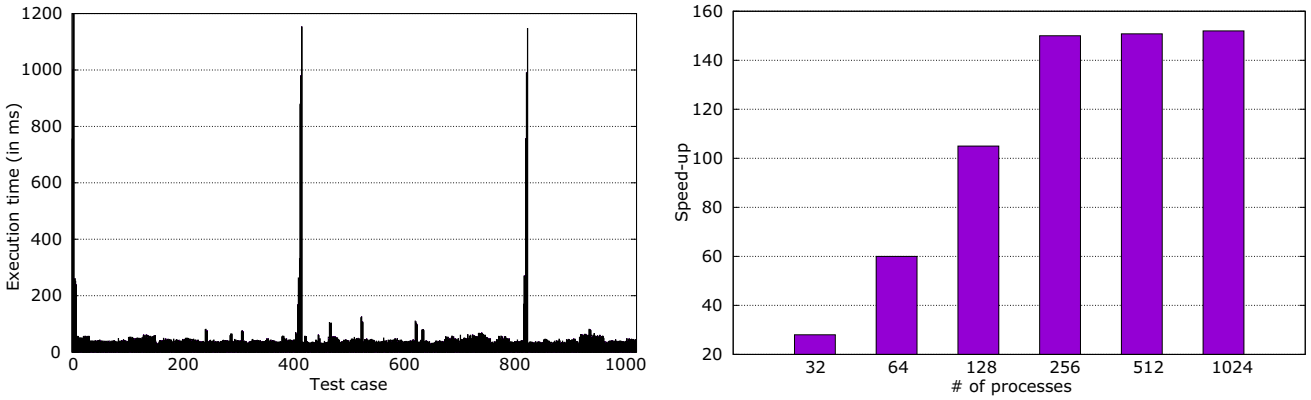
Next, we present the MuT evaluation results on each application of Table 2. In particular, the MuT results for *Filter* are described in Sect. 5.3.1, those for *Bzip2* are shown in Sect. 5.3.2, and the ones for *FFT* in Sect. 5.3.3.

5.3.1 Testing the *Filter* application

In the first place, we perform the testing process over the *Filter* application. In this case, we execute a test suite consisting of 1000—randomly created—test cases over the original source code of this application. The total time required to execute the test suite is 57 s, and the time required to completely execute the MuT process is 3570 s. Figure 6a shows the execution time of each test case over the original program, where the x-axis represents the test case ID—ranging from 0 to 999—and the y-axis shows the execution time measured in milliseconds. The majority of the test cases requires few milliseconds (30–40) to be executed. However, we noticed that the test cases with IDs 1–8, 401–408 and 801–818, require a significantly greater amount of time—between 250 and 1100 milliseconds—for being executed. In these particular cases, the filter is applied to large images, which require more CPU power to be processed.

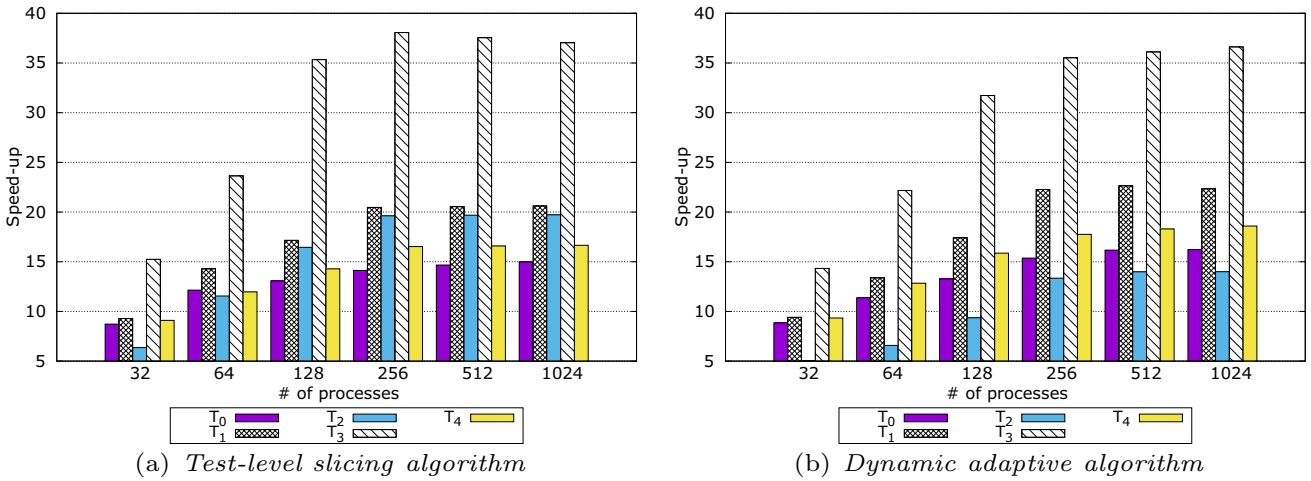
Figure 6b shows the time required to compile the mutants for testing the *Filter* application. The speed-up obtained grows linearly when the number of processes used for this task is below 256. However, since the total number of mutants to be compiled is 250, and one mutant can only be compiled by one process, using a greater number of processes keeps some CPU cores idle, hence limiting the overall performance. In the best case scenario, the speed-up obtained for this task reaches 158 when 256 processes are used.

Next, we study the obtained performance when the two distribution algorithms are used to check the test suite for testing the *Filter* application. In this case, 1000 test cases are executed over 250 mutants. Additionally, we use the current existing techniques (T_1, T_2, T_3 and T_4) to analyse its suitability to improve the performance of the MuT process in large-scale systems. Let us remind that T_0 means that no improvement is applied during the MuT process. Figure 7 depicts the results of this experiment, where Fig. 7a shows



(a) Execution time of the test suite (b) Compilation time of the mutants set

Fig. 6 Execution of the test suite over the original *Filter* application (left) and compilation of the mutants (right)



(a) *Test-level slicing algorithm* (b) *Dynamic adaptive algorithm*

Fig. 7 Speed-up of the test execution stage when techniques T_0–T_4 are applied to test the *Filter* application

the speed-up when the *Test-level slicing algorithm* is used, and Fig. 7b shows the speed-up when *Dynamic adaptive algorithm* is used. The x-axis depicts the number of processes involved in the MuT process, while y-axis shows the speed-up obtained.

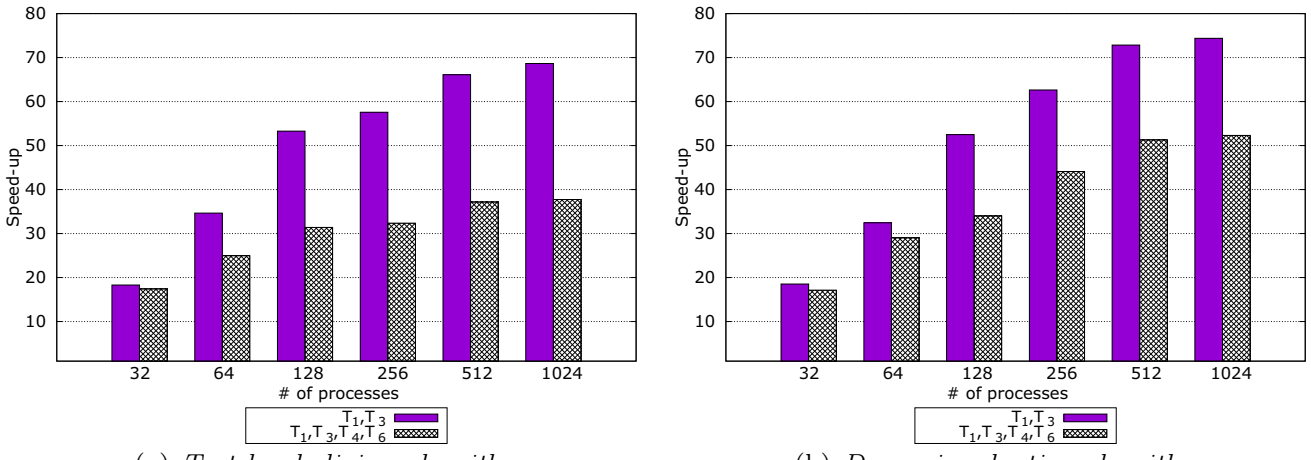
Broadly speaking, both algorithms provide similar results. Using load distribution algorithms without applying additional techniques (see the series for T_0) provides a speed-up of 17 in the best case scenario, that is, using the *Dynamic adaptive algorithm* and 1024 processes. The speed-up obtained for the techniques T_1, T_2 and T_4 follows a similar tendency for both algorithms. However, it is noticeable that T_1 is slightly better than T_2 and T_4. In this case, sorting the test suite (T_2) before executing the test cases is not suitable to boost the execution of the MuT process. This figure shows that the technique that provides the best results is T_3, which avoids the execution of the same test case over the same mutant in different processes. Thus, combining T_3 with both algorithms significantly improves the overall performance, reaching a speed-up of 38 and 36 when the *Test-level slicing algorithm* and *Dynamic adaptive algorithm* are used, respectively. We notice that the speed-up is not improved when more than 256 processes are used to execute the MuT process. This situation occurs because the I/O system is saturated and acts as a system bottleneck. Since each test reads an image from disk and, once filtered, writes the resulting image to disk, the I/O operations performed during the execution of a test case are significantly more time consuming than the computational operations executed to carry out the filtering process. Consequently, a high number of processes concurrently reading and writing images from/to the file system saturates the storage systems, causing a system bottleneck, hence limiting the speed-up obtained by the exploitation of the computational resources.

The following experiments aim at studying how the new proposed techniques (T_5 and T_6), focused on the stages of compiling mutants and detecting equivalent mutants, respectively, improve the overall performance of the MuT process. For this, both distribution algorithms are used in two different scenarios. First, we analyse the results obtained when no technique for improving the detection of equivalent mutants is applied, that is, only T_1 and T_3 are used. Secondly, we combine the techniques T_1, T_3, T_4 and T_6 to execute the MuT process. These results are shown in Fig. 8. For the sake of clarity, the results obtained when T_5 is applied are shown in Table 4, where the time required to execute the different stages of the MuT process are presented.

The results of this experiment clearly show that the *Dynamic adaptive algorithm* outperforms *Test-level slicing algorithm*, reaching a speed-up of 72 when 1024 processes are used. However, it is interesting to note that the current techniques, that is, T_1 and T_3, provide better results than the ones obtained when the new technique T_6 is used. Since testing the *Filter* application requires reading and writing a high number of images from/to the storage system, hence collapsing the storage system, the improvement obtained when the computational part of the testing process is parallelised, remains hidden. Thus, the major part of the test execution stage is spent in I/O operations, which clearly saturates the system. Nevertheless, these techniques provide better results when the *Dynamic adaptive algorithm* is used to distribute the test cases.

5.3.2 Testing the *Bzip2* application

Next, we report on the MuT process results for the *Bzip2* application. Figure 9a shows the time required to execute each test case over the original program, while Fig. 9b



(a) *Test-level slicing algorithm* (b) *Dynamic adaptive algorithm*

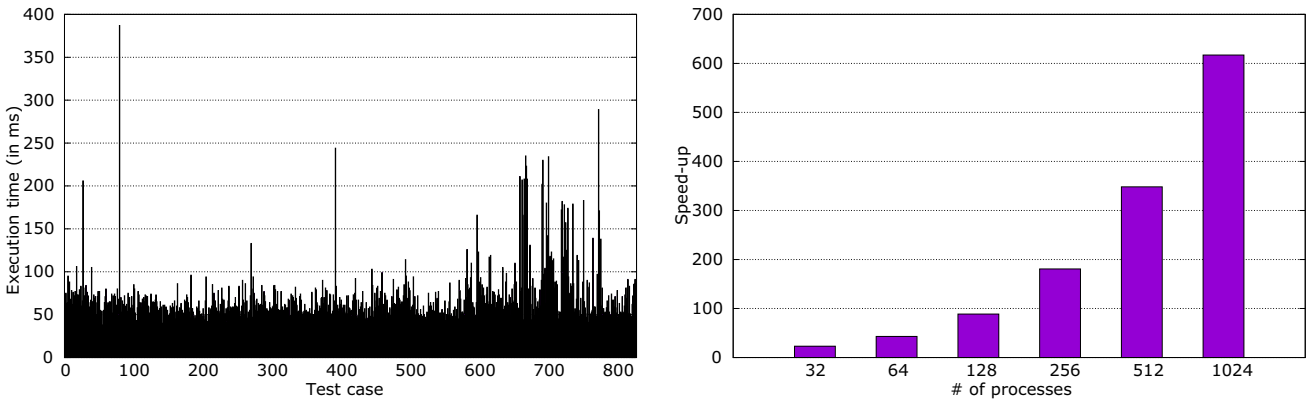
Fig. 8 Speed-up of the test execution stage when techniques T_1, T_3, T_4 and T_6 are applied to test the *Filter* application

Table 4 Summary of the results obtained in the empirical study

Application	Algorithm	# Proc.	Techniques	Gen. Mut.	Comp.	Det. Equiv.	Exec.	Total
Filter	*Test-level slicing*	512	T_1,T_3	0.32	39.44	–	54	93.76
			T_1,T_3,T_5	0.32	0.27	–	54	54.59
			T_1,T_3,T_4	0.32	39.44	0.5	95.9	136.1
			T_1,T_3,T_4,T_6	0.32	39.44	0.074	95.9	142.18
			T_1,T_3,T_4,T_5,T_6	0.32	0.27	0.074	95.9	96.59
		1024	T_1,T_3	0.32	39.44	–	52	91.76
			T_1,T_3,T_5	0.32	0.24	–	52	52.56
			T_1,T_3,T_4	0.32	39.44	0.5	93.9	134.18
			T_1,T_3,T_4,T_6	0.32	39.44	0.074	93.9	133.76
			T_1,T_3,T_4,T_5,T_6	0.32	0.24	0.074	93.9	94.56
	Dynamic adaptive	512	T_1,T_3	0.32	39.44	–	49	88.76
			T_1,T_3,T_5	0.32	0.27	–	49	49.59
			T_1,T_3,T_4	0.32	39.44	0.5	68.9	109.18
			T_1,T_3,T_4,T_6	0.32	39.44	0.074	68.9	108.76
			T_1,T_3,T_4,T_5,T_6	0.32	0.27	0.074	68.9	69.59
		1024	T_1,T_3	0.32	39.44	–	48	87.76
			T_1,T_3,T_5	0.32	0.24	–	48	**48.56**
			T_1,T_3,T_4	0.32	39.44	0.5	67.9	108.18
			T_1,T_3,T_4,T_6	0.32	39.44	0.074	67.9	107.76
			T_1,T_3,T_4,T_5,T_6	0.32	0.24	0.074	67.9	68.56
Bzip2	*Test-level slicing*	512	T_1,T_3	10.5	28,555	–	2883	31,448.5
			T_1,T_3,T_5	10.5	82	–	2883	2975.5
			T_1,T_3,T_4	10.5	28,555	19.5	2216.55	30,801.55
			T_1,T_3,T_4,T_6	10.5	28,555	0.45	2216.55	30,782.5
			T_1,T_3,T_4,T_5,T_6	10.5	82	0.45	2216.55	2309.5
		1024	T_1,T_3	10.5	28,555	–	2700	31,265.5
			T_1,T_3,T_5	10.5	62	–	2700	2772.5
			T_1,T_3,T_4	10.5	28,555	19.5	2037.75	30,622.75
			T_1,T_3,T_4,T_6	10.5	28,555	0.25	2037.75	30,603.5
			T_1,T_3,T_4,T_5,T_6	10.5	62	0.25	2037.75	2110.5
	Dynamic adaptive	512	T_1,T_3	10.5	28,555	–	2650	31,215.5
			T_1,T_3,T_5	10.5	82	–	2650	2742.5
			T_1,T_3,T_4	10.5	28,555	19.5	2037.55	30,622.55
			T_1,T_3,T_4,T_6	10.5	28,555	0.45	2037.55	30,603.5
			T_1,T_3,T_4,T_5,T_6	10.5	82	0.45	2037.55	2130.5
		1024	T_1,T_3	10.5	28,555	–	2500	31,065.5
			T_1,T_3,T_5	10.5	62	–	2500	2572.5
			T_1,T_3,T_4	10.5	28,555	19.5	1922.75	30,507.75
			T_1,T_3,T_4,T_6	10.5	28,555	0.25	1922.75	30,488.5
			T_1,T_3,T_4,T_5,T_6	10.5	62	0.25	1922.75	**1995.5**
FFT	*Test-level slicing*	512	T_1,T_3	150.4	405,584	–	4934	410,128.4
			T_1,T_3,T_5	150.4	1233	–	4934	5777.4
			T_1,T_3,T_4	150.4	405,584	90	3299.5	416,522.5
			T_1,T_3,T_4,T_6	150.4	405,584	0.5	3299.5	409,034.4
			T_1,T_3,T_4,T_5,T_6	150.4	1233	0.5	3299.5	4683.4
		1024	T_1,T_3	150.4	405,584	–	3950	409,684.4
			T_1,T_3,T_5	150.4	698	–	3950	4798.4
			T_1,T_3,T_4	150.4	405,584	90	2999.7	416,232.5

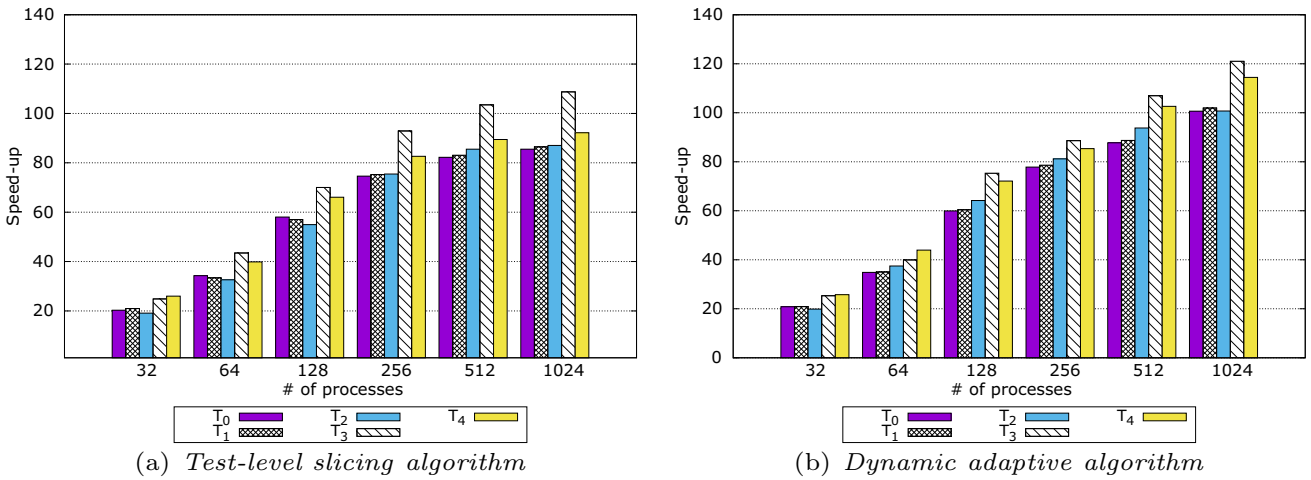
Table 4 (continued)

Application	Algorithm	# Proc.	Techniques	Gen. Mut.	Comp.	Det. Equiv.	Exec.	Total
			T_1,T_3,T_4,T_6	150.4	405,584	0.3	2999.7	408,734.4
			T_1,T_3,T_4,T_5,T_6	150.4	698	0.3	2999.7	3848.4
	Dynamic adaptive	512	T_1,T_3	150.4	405,584	–	3575	409,309.4
			T_1,T_3,T_5	150.4	1233	–	3575	4958.4
			T_1,T_3,T_4	150.4	405,584	90	2689.5	415,912.5
			T_1,T_3,T_4,T_6	150.4	405,584	0.5	2689.5	408,424.4
			T_1,T_3,T_4,T_5,T_6	150.4	1233	0.5	2689.5	4073.4
		1024	T_1,T_3	150.4	405,584	–	3150	408,884.4
			T_1,T_3,T_5	150.4	698	–	3150	3998.4
			T_1,T_3,T_4	150.4	405,584	90	2236.7	408,061.18
			T_1,T_3,T_4,T_6	150.4	405,584	0.3	2236.7	407,971.4
			T_1,T_3,T_4,T_5,T_6	150.4	698	0.3	2236.7	**3085.4**



(a) Execution time of the test suite (b) Compilation time of the mutants set

Fig. 9 Execution of the test suite over the original *Bzip2* application (left) and compilation of the mutants (right)



(a) *Test-level slicing algorithm* (b) *Dynamic adaptive algorithm*

Fig. 10 Speed-up of the test execution stage when techniques T_0–T_4 are applied to test the *Bzip2* application

depicts the speed-up obtained when compiling all the generated mutants over a range of 32–1024 processes. In this case, we notice that the major part of the test cases requires less than 100 milliseconds to be executed. Some peaks appear in this chart, which represent the execution of some test cases that compress large files and, consequently, require more computational operations to be completed. Regarding the compilation of the mutants, the performance obtained to accomplish this task linearly grows with the total number of processes used. In this particular case 13,000 mutants are compiled and the technique T_5 efficiently exploits the parallelization of the computational resources, reaching a speed-up of 600.

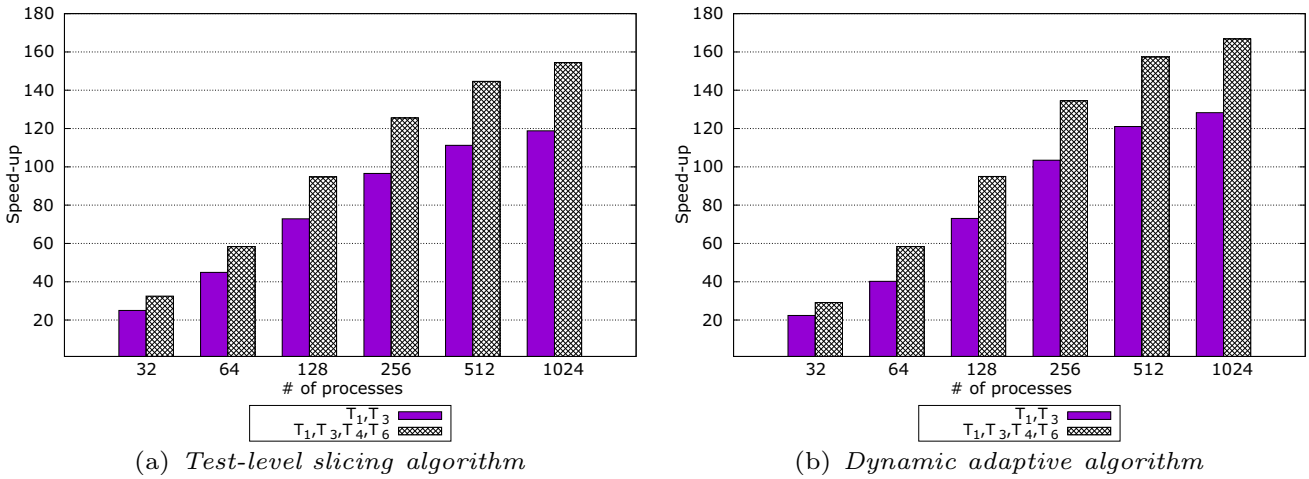
Figure 10 shows the results obtained from testing the *Bzip2* application using the current techniques to improve the performance of the MuT process. In contrast to the previous experiment, where the *Filter* application was tested, each test case reads a file to be compressed and the generated data is allocated in memory, and not stored in disk. Consequently, the storage system is less saturated than in the previous experiment and the bottleneck in the storage system is alleviated, hence allowing to increase the overall system performance. In this particular case, we notice that the applied techniques provide similar results. It is worth mentioning that T_3 shows a slightly better performance than the rest of the techniques. Regarding the load distribution algorithm, we notice that the *Dynamic adaptive algorithm* provides better results than the *Test-level slicing algorithm*, especially when a high number of processed are used, reaching in the best case scenario a speed-up of 120.

Figure 11 depicts the results obtained when different techniques are combined to improve the MuT process. In contrast to the previous application, combining T_1, T_3, T_4 and T_6 with the *Dynamic adaptive algorithm* provides the best results, reaching in the best case scenario a speed-up of 126 (using 1024 processes). In general, the *Dynamic adaptive algorithm* provides slightly better results than the ones obtained when the *Test-level slicing algorithm* is used. Both algorithms show a similar scalability when the number of processes is increased.

5.3.3 Testing the *FFT* application

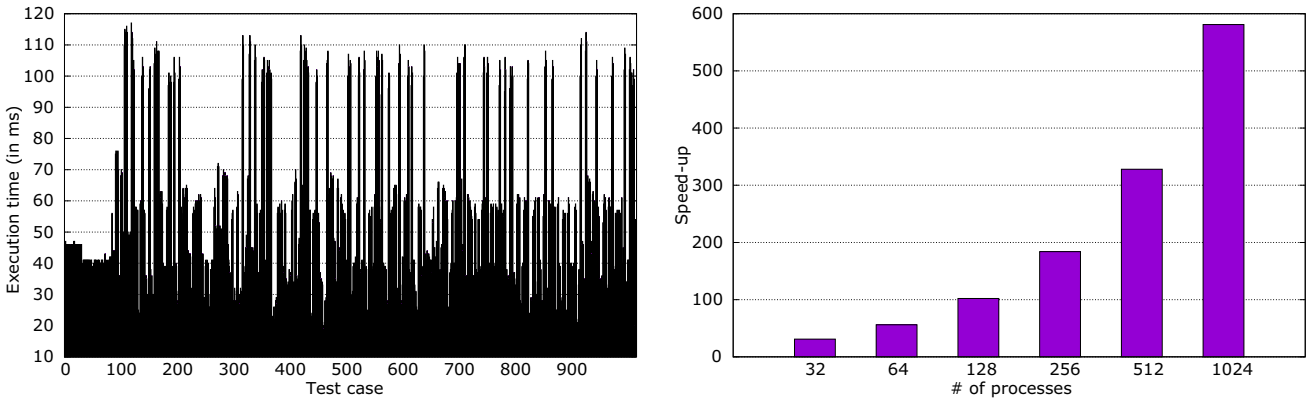
The last application to be tested is *FFT*. The chart in Fig. 12a shows the execution time over the original program of each test case. Although the average execution time of these test cases is approximately 50 milliseconds, there is a significant number of test cases that require more than 100 milliseconds to be executed. Figure 12b shows the time required to compile the 75,000 mutants involved in the MuT process. The number of mutants is significantly greater than the number of processes and, therefore, this stage can be parallelised to exploit the parallelism of the computational resources provided by the target system. Let us remark that 1024 CPU-cores are used for this task, reaching a speed-up of approximately 600 in this case.

Figure 13 shows the results obtained to carry out the MuT process over *FFT* when the *Test-level slicing algorithm* (see Fig. 13a) and *Dynamic adaptive algorithm* (see Fig. 13b) are used. As with the previous application, in this case techniques T_0, T_1, T_2 and T_4 provide similar results. On the contrary, when T_3 is applied, a noticeable increment in the overall performance is shown in these charts. Since this application performs computing operations massively, the techniques focused on optimizing the execution of the test cases have a clear positive impact on the overall performance. This scenario occurs for both algorithms. However, the *Dynamic adaptive algorithm* clearly



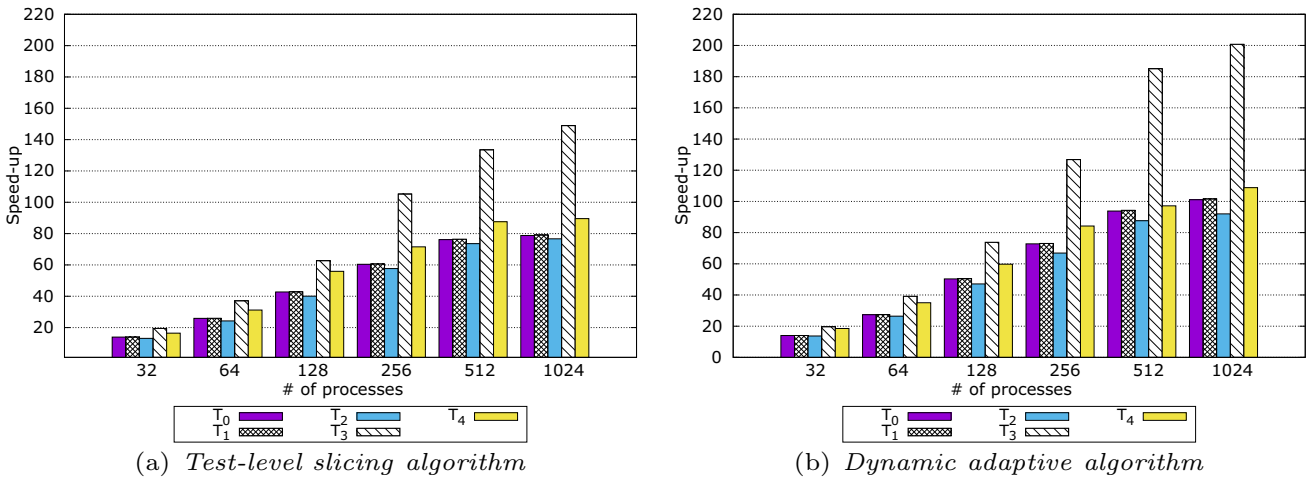
(a) *Test-level slicing algorithm* (b) *Dynamic adaptive algorithm*

Fig. 11 Speed-up of the test execution stage when T_1, T_3, T_4 and T_6 are applied to test the *Bzip2* application



(a) Execution time of the test suite (b) Compilation time of the mutants set

Fig. 12 Execution of the test suite over the original *FFT* application (left) and compilation of the mutants (right)



(a) *Test-level slicing algorithm* (b) *Dynamic adaptive algorithm*

Fig. 13 Speed-up of the test execution stage when techniques T_0–T_4 are applied to test the *FFT* application

outperforms *Test-level slicing algorithm*, obtaining a speed-up of 200 when 1024 processes are involved in the MuT process.

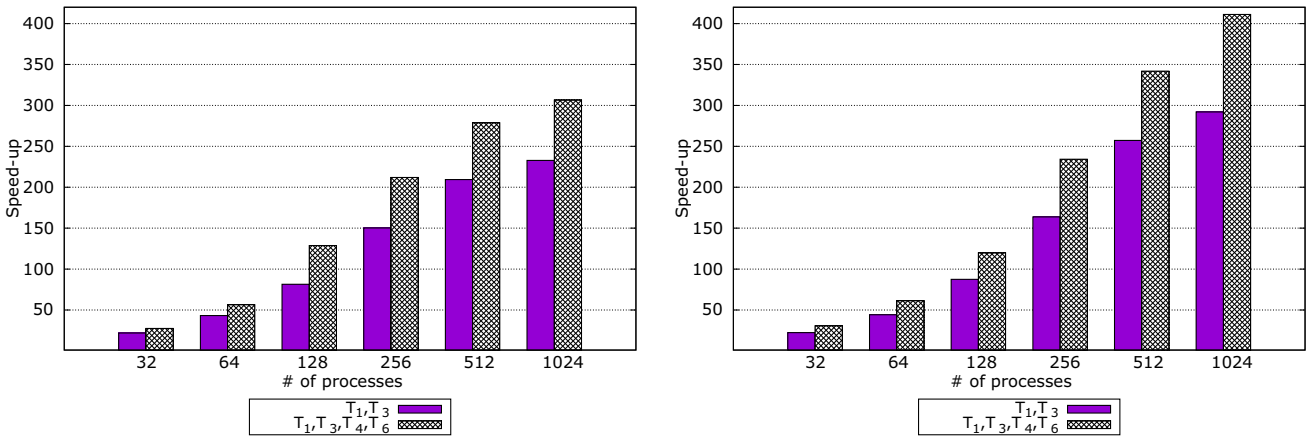
The results obtained when different techniques are applied to both distribution algorithms, for testing the *FFT* application, are depicted in the Fig. 14a and b. In this case, the *Dynamic adaptive algorithm* clearly provides the best results, reaching in the best case scenario a speed-up of 400. On the contrary, using the *Test-level slicing algorithm* provides—in the best case scenario—a speed-up of 300. The main reason for this difference lies in the nature of the application being tested, which massively executes CPU operations. Let us remind that this application does not perform I/O operations. Hence, the *Dynamic adaptive algorithm*, when combined with T_6, efficiently exploits the CPUs of the system, which provides promising results outperforming the rest of the techniques when combined with the *Test-level slicing algorithm*.

5.4 Discussion of the results and answers to the research questions

This section discusses the results of the empirical study, and answers the research questions formulated in Sect. 5.1.

5.4.1 Discussion of the experiment results

Table 4 presents a summary of the results of the experiments. For the sake of clarity, only the most representative stages of the MuT process are shown. In essence, these results show the time required to execute each stage of the MuT process using a different number of processes, two different load distribution algorithms and six techniques to speed-up its execution (see Table 3). The first three columns show the application being tested, the load distribution algorithm, and the number of processes involved in the MuT process (*# Proc.*). The column labelled as *Techniques*



(a) *Test-level slicing algorithm* (b) *Dynamic adaptive algorithm*

Fig. 14 Speed-up of the test execution stage when techniques T_1, T_3, T_4 and T_6 are applied to test the *FFT* application

lists the techniques applied during the MuT process. The next four columns represent the time (in seconds) required to execute each stage of the MuT process, that is, generating mutants (*Gen. Mut.*), compiling mutants (*Compilation*), detecting equivalent mutants (*Det. Equiv.*) and executing test cases over the mutants (*Exec.*). The last column, labelled as *Total*, shows the total time required to completely execute the MuT process, measured in seconds.

In this study, generating the mutants seems to be a task that requires low computational resources. Approximately, this stage requires between 0.6%—in the *Filter* application—and 5%—in the *FFT* application—of the total execution time.

On the contrary, the compilation stage is critical in the MuT process. Here, a large number of mutants need to be compiled, which requires high computational resources. When technique T_5 is not applied, this stage requires—approximately—39, 28,000 and 405,000 s to compile the mutants for testing the *Filter*, *Bzip2* and *FFT* applications, respectively. However, when the T_5 is applied, the time required to accomplish this task is reduced to 0.27, 82 and 1233 s, respectively. It is important to remark that this technique significantly improves the overall performance of the MuT process. For the sake of clarity, the experiments using T_5 are highlighted with a grey background in Table 4.

Detecting equivalent mutants is another stage—like generating mutants—that requires few computational resources. In the column labelled as *Det. Equiv.*, a '-' symbol indicates that no equivalent mutants were detected. In this study, the time required to accomplish this task for testing the *Filter*, *Bzip2* and *FFT* applications is reduced from 0.5 to 0.074 s, from 19.5 to 0.25 s, and from 90 to 0.3 s, respectively, when T_6 is applied. In general, this technique provides a slight improvement in the total execution time, which is more noticeable when a high number of processes are used.

Another critical stage of the MuT process is the execution of the test cases over the mutants. This stage is the most expensive one, in terms of computational resources, to be completed. In this study, it requires between 70 and 99% of the total execution time, when T_5 and T_6 are applied.

The empirical study has been designed to analyse the scalability of different load distribution algorithms to execute the MuT process in large-scale system and, therefore, after a careful analysis of the results, we can conclude that the *Dynamic adaptive algorithm* provides better performance than the *Test-level slicing algorithm*. Our proposed algorithm has been designed to maximise the exploitation of computational resources while reducing the inter-process communications. Table 4 shows in bold text the best MuT process times over each tested application. For the three applications, these results are obtained when the *Dynamic adaptive algorithm* is executed using 1024 processes. In addition, the scalability of the algorithm is reflected in the obtained results, where the execution time is reduced when the number of processes increases.

In conclusion, the proposed techniques clearly improve the overall performance of the MuT process. To check the effectiveness of these techniques, different programs with 500, 7000, and 13,500 LoC have been selected, with a sequential execution time of 3570, 320,740, and 919,825 s, that is, 60 min, 72 h, and 10 days, respectively. All this is translated in a reduction of the execution time from up to 10 days to 51 min, representing a time reduction of 99.66%.

5.4.2 Answer to the research questions

Next, we proceed to answer the formulated research questions.

RQ1 *How suitable are the current distribution algorithms to parallelise the MuT process?*

Regarding the *Test-level slicing algorithm* in terms of performance, we obtain—in the best-case scenario—a speed-up of 68, 156 and 303 for testing the *Filter*, *Bzip2* and *FFT* applications, respectively, when 1024 are used to execute the MuT process in parallel. These results show that this algorithm provides an acceptable scalability, increasing the speed-up of the MuT process when a high number of processes are deployed in a large system.

We notice that the speed-up obtained heavily depends on the nature of the application being tested. Thus, the applications that require a massive utilization of the storage system, like the *Filter* application, generate a system bottleneck, which limits the overall performance. This bottleneck is alleviated when the MuT process is applied over the *Bzip2* application which is clearly reflected in the obtained results. In the case where the storage system doesn't act as a system bottleneck, that is, when *FFT* is the application under test, the speed-up reaches 300.

Thus, the answer to this question is that *the Test-level slicing algorithm can be used in large-scale system, providing acceptable results.*

RQ2 *How effective is our proposed distribution algorithm to improve the overall performance of the MuT process in large-scale systems?*

The empirical study provides detailed results that show a clear improvement in the overall system performance when the *Dynamic adaptive algorithm* is applied to carry out the MuT process. In particular, our proposed algorithm outperforms the *Test-level slicing algorithm* by increasing the speed-up obtained by 7.3%, 4.4% and 36.9% when the applications *Filter*, *Bzip2* and *FFT*, respectively, are tested.

It is interesting to remark that the applications that massively perform computing operations, like *FFT*, provide the best results. In addition, in those cases where a subsystem—like the storage system—acts as a system bottleneck, our proposed algorithm efficiently exploits the shared resources, hence increasing the speed-up and, consequently, the overall performance.

After a careful analysis of the results presented in Table 4 we state that *our proposed algorithm efficiently exploits the shared resources of the system, clearly outperforming other algorithms, like the Test-level slicing algorithm, even in those cases where there is a bottleneck in the system.*

RQ3 *What techniques are appropriate to improve the performance of the MuT process in large-scale systems?*

As Table 4 shows, some stages of the MuT process are critical in the overall performance, like compiling the mutants and executing the test cases. In order to boost the execution of these stages, we have proposed techniques T_5 and T_6, which clearly show a significant improvement in the speed-up obtained when the MuT process is executed in a large-scale system. Let us remark that the results provided when technique T_5 is applied are shown with a grey background. Similarly, the rest of the techniques, which were studied in previous works [20, 23], have also been designed to improve the performance of the MuT process.

The answer to this question is that *T_5 and T_6 significantly improve the overall system performance when the MuT process is applied over the three applications. Similarly, technique T_3 also improves the overall performance, especially when it is applied to testing the Filter and FFT applications. Nevertheless, these techniques provide the best results when the Dynamic adaptive algorithm is used to distribute the test cases.*

6 Threats to validity

In this section, we discuss the threats to validity of our empirical study.

6.1 Internal threats

Internal validity concerns whether our findings, which are based on the obtained results from the empirical study, truly represent a cause-and-effect relationship. Thus, the internal validity of our study relies in the implementation of our experiments.

The techniques proposed in this paper might not reflect the expected behaviour due to implementation failures. In order to mitigate this possible threat, we have implemented different techniques found in the current literature, whose results have been compared with the ones shown in the original articles, and also compared with the results obtained in this paper to check their suitability. In addition, parallel executions of MuT must provide the same mutation score as the sequential version. This fact is important since the same MuT environment—consisting of the program under test, the test suite, and the mutants—executed both sequentially and in parallel, might provide different mutation scores due to misconfigured parameters, like timeouts. When parallel approaches provide a higher mutation score than the one provided by the sequential execution, it means that not all the test cases that have been sequentially executed are also executed in the parallel approach and, therefore, the overall execution time should be considerably reduced. For instance, if a mutant becomes mistakenly killed by the first test case, the rest of the test

cases are not executed and, consequently, the total execution time is reduced, hence providing a misinterpreted improved performance.

The framework developed to conduct the experimental phase has been tested by using unit tests, in addition to its incremental testing during the development phase.

6.2 External threats

External validity concerns the extent to which the results of a study can be generalised.

In our case, we have selected three different applications, which we consider representative of typical program behaviours (CPU-intensive, disk-intensive). However, further experiments with a wider variety of applications would be required, since there is no guarantee that the obtained results may be the same for other scenarios. In addition, we have used *Cirrus* to conduct our experiment, going up to 1024 processes. We believe this number of processes is enough to discover possible bottlenecks and issues in the distribution algorithms.

It is worth remarking that the mutant-level distribution has not been included in the comparison presented in Sect. 5, since we obtain better results using the test-level distribution, as it is shown in previous work [23]. Thus, we focus on exploring different combinations of strategies to analyse different scenarios using the test-level distribution and adaptive grains. To that end, the selected applications were tested by using a different number of processes—ranging from 2 to 1024—seven different techniques, and five combinations of two distribution algorithms. For this, a total of 200,000 h of computing time in the *Cirrus* supercomputer were required to execute the experiments.

Finally, our experiments have targeted applications written in the C language. However, we believe the results would be applicable to other programming languages as well. However, in our case, mutation has been performed at the source-code level, and hence each mutant requires recompilation. Programming languages like Java are compiled into a bytecode representation, and executed by a virtual machine. This way, there are MuT approaches for Java that perform mutation at the bytecode level [69], without the need to recompile. For this reason, the proposed compiling optimisation not applies to byte-code approaches. However, we believe that our *Dynamic adaptive algorithm* would have substantial beneficial effects in bytecode-level approaches to MuT.

6.3 Construct threats

Construct validity concerns whether the used measures are representative or not.

The performance of the proposed approach is measured using the speed-up obtained, which is a widely used measure in the community. The experiments were repeated several times to alleviate the experimental variability caused by the computational environment, that is, network overload, operating system noise, and applications executed by other users in the cluster, among others. In addition, we measure the quality of the test suite finding faults by using mutation score, a metric that is currently considered a standard.

7 Conclusions and future work

In this paper, we have presented an HPC-based optimization for the MuT process over large-scale systems. This optimization consists of a set of strategies aimed at improving the performance of the MuT process, such as parallel compilation and parallel equivalent detection, and a dynamic adaptive algorithm that adjust and distributes the testing workload to the different worker processes.

Our proposal has been validated by a thorough experimental study to evaluate its effectiveness and scalability. In essence, the improvements proposed in this work have been applied for testing three different applications in several computational environments, which consist of a different number of processes ranging from 2 to 1024. After a comprehensive analysis, we have detected that the speed-up of the MuT process depends on the nature of the application under study. That is, if the application performs a high number of storage operations, it may lead to a system bottleneck and, consequently, the overall performance is reduced. However, in those cases where the application under test mainly performs computational operations, the resources of the cluster are efficiently exploited, hence providing a significant improvement in the overall system performance. After a careful analysis of the obtained results, we can conclude that the proposed approaches are suitable to improve the MuT process in large scale system. In fact, the proposed techniques significantly reduce the total time required to conduct the MuT process. In addition, it is worth to remark that the *Dynamic adaptive algorithm* outperforms *Test-level slicing algorithm* increasing the speed-up of the MuT process up to 36.9%.

As future work, we plan to study and design compression techniques for reducing the size of the data exchanged between processes, hence reducing the network traffic. We are also interested in testing applications written in different programming languages, such as Java and Python. To that end, we will design more experiments with different well-known MuT frameworks, like MuJava [25], PiTest [69] and MutPy [70]. Finally, we plan to compare

the performance of the proposed techniques with other existing HPC architectures, such as Spark and Cloud computing.

Acknowledgements This work was supported by the Spanish MINECO/FEDER project under Grants PID2021-122270OB-I00, TED2021-129381B-C21 and PID2019-108528RB-C22, the Comunidad de Madrid project FORTE-CM under Grant S2018/TCS-4314, Project S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union and Comunidad de Madrid and the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme; in particular, the author gratefully acknowledges the support of David Henty of EPCC at University of Edinburgh and the computer resources and technical support provided by EPCC.

Author contributions The first draft of the manuscript was written by PCC and AN, and all authors commented and improved the previous versions of the manuscript. All authors read and approved the final manuscript. The study conception and design was conducted by PCC and AN. Material preparation, data collection and analysis were performed by PCC and AN.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work was supported by the Spanish MINECO/FEDER project under Grants PID2021-122270OB-I00, TED2021-129381B-C21 and PID2019-108528RB-C22, the Comunidad de Madrid project FORTE-CM under Grant S2018/TCS-4314, Project S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union and Comunidad de Madrid and the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme.

Data availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

Declarations

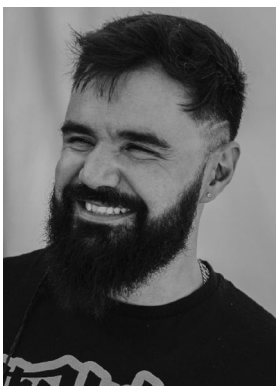
Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Hierons, R.M., Merayo, M.G., Núñez, M.: Controllability through nondeterminism in distributed testing. In: ICTSS, pp. 89–105 (2016). https://doi.org/10.1007/978-3-319-47443-4_6
2. EUROCAE (Agency). WG-12, RTCA (Firm). SC 167: Software considerations in airborne systems and equipment certification. Eurocae (1985). https://books.google.es/books?id=3CbxxwEACAAJ
3. Jordan, P.: Standard IEC 62304-medical device software–software lifecycle processes (2006)
4. Palin, R., Ward, D., Habli, I., Rivett, R.: ISO 26262 safety cases: compliance and assurance (2011)
5. Cañizares, P.C., Núñez, A., Merayo, M.G.: Mutomvo: mutation testing framework for simulated cloud and HPC environments. J. Syst. Softw. **143**, 187–207 (2018). https://doi.org/10.1016/j.jss.2018.05.010
6. Ali, M.M., Huda, S., Abawajy, J., Alyahya, S., Al-Dossari, H., Yearwood, J.: A parallel framework for software defect detection and metric selection on cloud computing. Clust. Comput. **20**(3), 2267–2281 (2017). https://doi.org/10.1007/s10586-017-0892-6
7. Rauf, A., Ramzan, M.: Parallel testing and coverage analysis for context-free applications. Clust. Comput. **21**(1), 729–739 (2018). https://doi.org/10.1007/s10586-017-1000-7
8. Braiek, H.B., Khomh, F.: On testing machine learning programs. J. Syst. Softw. **164**, 110542 (2020). https://doi.org/10.1016/j.jss.2020.110542
9. Fraser, G., Wotawa, F.: Redundancy based test-suite reduction. In: International Conference on Fundamental Approaches to Software Engineering, pp. 291–305. Springer (2007). https://doi.org/10.1007/978-3-540-71289-3_23
10. Budd, T.: Mutation analysis of program test data. PhD Thesis, Yale University (1980)
11. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer **11**(4), 34–41 (1978). https://doi.org/10.1109/C-M.1978.218136
12. Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Trans. Softw. Eng. **3**(4), 279–290 (1977). https://doi.org/10.1109/TSE.1977.231145
13. Just, R., Kapfhammer, G.M., Schweiggert, F.: Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In: 23rd International Symposium on Software Reliability Engineering, pp. 11–20. IEEE (2012). https://doi.org/10.1109/ISSRE.2012.31
14. Klischies, D., Fögen, K.: An analysis of current mutation testing techniques applied to real world examples. Full-scale Softw. Eng./Curr. Trends Release Eng. **13**, 52 (2016)
15. Možucha, J., Rossi, B.: Is mutation testing ready to be adopted industry-wide? In: International Conference on Product-Focused Software Process Improvement, pp. 217–232. Springer (2016). https://doi.org/10.1007/978-3-319-49094-6_14
16. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Mutation testing advances: an analysis and survey. In: Advances in Computers, vol. 112, pp. 275–378. Elsevier, Amsterdam (2019). https://doi.org/10.1016/bs.adcom.2018.03.015
17. Dongarra, J.J., Meuer, H.W., Strohmaier, E., et al.: TOP500 supercomputer sites. Supercomputer **13**, 89–111 (1997)
18. Strohmaier, E., Dongarra, J., Simon, H., Meuer, M.: TOP 500 list. Web page at http://www.top500.org (2020). Accessed 8 June 2020
19. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62
20. Cañizares, P.C., Núñez, A., de Lara, J.: OUTRIDER: optimizing the mutation testing process in distributed environments. In: 17th International Conference on Computational Science. ICCS'17, pp. 505–514, Zurich (2017). https://doi.org/10.1016/j.procs.2017.05.095
21. Reales, P., Polo, M.: Bacterio: Java mutation testing tool: a framework to evaluate quality of tests cases. In: 28th International Conference on Software Maintenance. ICSME'12,

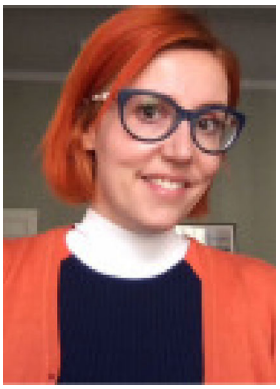
pp. 646–649. IEEE (2012). https://doi.org/10.1109/ICSM.2012.6405344
22. Vercammen, S., Demeyer, S., Borg, M., Eldh, S.: Speeding up mutation testing via the cloud: lessons learned for further optimisations. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–9 (2018). https://doi.org/10.1145/3239235.3240506
23. Cañizares, P.C., Merayo, M.G., Núñez, A.: EMINENT: embarrassingly parallel mutation testing. In: 16th International Conference on Computational Science. ICCS'16, pp. 63–73 (2016). https://doi.org/10.1016/j.procs.2016.05.298
24. Papadakis, M., Jia, Y., Harman, M., Traon, Y.L.: Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: 7th International Conference on Software Engineering. ICSE '15, pp. 936–946. IEEE Press (2015). https://doi.org/10.1109/ICSE.2015.103
25. Ma, Y., Offutt, A.J., Kwon, Y.R.: MuJava: an automated class mutation system: research articles. Softw. Test. Verif. Reliab. **15**(2), 97–133 (2005). https://doi.org/10.1002/stvr.308
26. Jia, Y., Harman, M.: MILU: a customizable, runtime-optimized higher order mutation testing tool for the full C language. In: Testing: Academic & Industrial Conference—Practice and Research Techniques. TAIC'08, pp. 94–98. IEEE (2008). https://doi.org/10.1109/TAIC-PART.2008.18
27. Gropp, W., Thakur, R., Lusk, E.: Using MPI-2: Advanced Features of the Message Passing Interface. MIT Press, Cambridge (1999)
28. Losada, N., González, P., Martín, M.J., Bosilca, G., Bouteiller, A., Teranishi, K.: Fault tolerance of MPI applications in exascale systems: the ULFM solution. Future Gener. Comput. Syst. **106**, 467–481 (2020). https://doi.org/10.1016/j.future.2020.01.026
29. Laguna, I., Marshall, R., Mohror, K., Ruefenacht, M., Skjellum, A., Sultana, N.: A large-scale study of MPI usage in open-source HPC applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '19. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3295500.3356176
30. Thakur, R., Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Hoefler, T., Kumar, S., Lusk, E., Träff, J.L.: MPI at exascale. Proc. SciDAC **2**, 14–35 (2010)
31. Qureshi, K., Rashid, H.: A performance evaluation of RPC, JAVA RMI, MPI and PVM. Malays. J. Comput. Sci. **18**(2), 38–44 (2005)
32. Khan, R.Z., Ali, M.F.: A comparative study on parallel programming tools in parallel distributed computing system: MPI and PVM. In: Proceedings of the 5th National Conference (2011)
33. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: 27th International Conference on Software Engineering. ICSE'05, pp. 402–411 (2005). ACM. https://doi.org/10.1145/1062455.1062530
34. Budd, T.A., Angluin, D.: Two notions of correctness and their relation to testing. Acta Inform. **18**(1), 31–45 (1982)
35. Offutt, A.J., Craft, W.M.: Using compiler optimization techniques to detect equivalent mutants. Softw. Test. Verif. Reliab. **4**(3), 131–154 (1994). https://doi.org/10.1002/stvr.4370040303
36. Hierons, R.M., Harman, M., Danicic, S.: Using program slicing to assist in the detection of equivalent mutants. Softw. Test. Verif. Reliab. **9**(4), 233–262 (1999)
37. King, K.N., Offutt, A.J.: A Fortran language system for mutation-based software testing. Softw. Pract. Exp. **21**(7), 685–718 (1991)
38. Hussain, S.: Mutation clustering. Masters Thesis, Kings College London, Strand, London (2008)
39. Ji, C., Chen, Z., Xu, B., Zhao, Z.: A novel method of mutation clustering based on domain analysis. In: SEKE, vol. 9, pp. 422–425. CiteSeer (2009)
40. Mresa, E.S., Bottaci, L.: Efficiency of mutation operators and selective mutation strategies: an empirical study. Softw. Test. Verif. Reliab. **9**(4), 205–232 (1999)
41. Polo, M., Piattini, M., García-Rodríguez, I.: Decreasing the cost of mutation testing with second-order mutants. Softw. Test. Verif. Reliab. **19**(2), 111–131 (2009)
42. Offutt, A.J., Rothermel, G., Zapf, C.: An experimental evaluation of selective mutation. In: Proceedings of 1993 15th International Conference on Software Engineering, pp. 100–107. IEEE (1993)
43. Jia, Y., Harman, M.: Constructing subtle faults using higher order mutation testing. In: 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 249–258. IEEE (2008)
44. Gopinath, R., Ahmed, I., Alipour, M.A., Jensen, C., Groce, A.: Mutation reduction strategies considered harmful. IEEE Trans. Reliab. **66**(3), 854–874 (2017)
45. Chekam, T.T., Papadakis, M., Traon, Y.L., Harman, M.: An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: 39th International Conference on Software Engineering (ICSE'17), pp. 597–608 (2017)
46. Offutt, A.J., King, K.N.: A Fortran 77 interpreter for mutation analysis. In: Papers of the Symposium on Interpreters and Interpretive Techniques, pp. 177–188 (1987)
47. Baruch, O., Katz, S.: Partially interpreted schemas for CSP programming. Sci. Comput. Program. **10**(1), 1–18 (1988)
48. Untch, R.H., Offutt, A.J., Harrold, M.J.: Mutation analysis using mutant schemata. In: Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 139–148 (1993)
49. Ma, Y.-S., Kwon, Y.-R., Offutt, J.: Inter-class mutation operators for Java. In: 13th International Symposium on Software Reliability Engineering, 2002. Proceedings, pp. 352–363. IEEE (2002)
50. Offutt, A.J., Pargas, R.P., Fichter, S.V., Khambekar, P.K.: Mutation testing of software using a MIMD computer. In: 21st International Conference on Parallel Processing. ICPP'92, pp. 255–266 (1992)
51. Kapfhammer, G.M.: Automatically and transparently distributing the execution of regression test suites. In: Proceedings of the 18th International Conference on Testing Computer Software, vol. 18 (2001)
52. Byoungju, C., Mathur, A.P.: High-performance mutation testing. J. Syst. Softw. **20**(2), 135–152 (1993)
53. Reales, P., Polo, M.: Parallel mutation testing. Softw. Test. Verif. Reliab. **23**(4), 315–350 (2013). https://doi.org/10.1002/stvr.1471
54. Hummel, S.F., Schonberg, E., Flynn, L.E.: Factoring: a method for scheduling parallel loops. Commun. ACM **35**(8), 90–101 (1992)
55. Grosso, W.: Java RMI, 1st edn. O'Reilly & Associates Inc., Sebastopol (2001). https://doi.org/10.5555/560800
56. Saleh, I., Nagi, K.: HadoopMutator: a cloud-based mutation testing framework. In: Software Reuse for Dynamic Systems in the Cloud and Beyond, pp. 172–187. Springer International Publishing Switzerland (2014). https://doi.org/10.1007/978-3-319-14130-5_13
57. Bernstein, D.: Containers and cloud: from LXC to Docker to Kubernetes. IEEE Cloud Comput. **1**(3), 81–84 (2014). https://doi.org/10.1109/MCC.2014.51
58. Dossot, D.: RabbitMQ Essentials. Packt Publishing Ltd, Birmingham (2014)
59. Usaola, M.P., Mateo, P.R.: Mutation testing cost reduction techniques: a survey. IEEE Softw. **27**(3), 80–86 (2010). https://doi.org/10.1109/MS.2010.79
60. Ma, Y., Kim, S.: Mutation testing cost reduction by clustering overlapped mutants. J. Syst. Softw. **115**, 18–30 (2016). https://doi.org/10.1016/j.jss.2016.01.007

61. Delamaro, M.E., Andrade, S.A., de Souza, S.R., de Souza, P.S.: Parallel execution of programs as a support for mutation testing: a replication study. Int. J. Softw. Eng. Knowl. Eng. **31**(03), 337–380 (2021). https://doi.org/10.1142/S0218194021500121
62. Krauser, E.W., Mathur, A.P., Rego, V.J.: High performance software testing on SIMD machines. IEEE Trans. Softw. Eng. **17**(5), 403–423 (1991). https://doi.org/10.1109/32.90444
63. Yao, X., Harman, M., Jia, Y.: A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Proceedings of the 36th International Conference on Software Engineering, pp. 919–930 (2014). https://doi.org/10.1145/2568225.2568265
64. Szalay, A.: Extreme data-intensive scientific computing. Comput. Sci. Eng. **13**(6), 34–41 (2011). https://doi.org/10.1109/MCSE.2011.74
65. Chen, C.L.P., Zhang, C.-Y.: Data-intensive applications, challenges, techniques and technologies: a survey on Big Data. Inf. Sci. **275**, 314–347 (2014). https://doi.org/10.1016/j.ins.2014.01.015
66. Peng, J., Dai, Y., Rao, Y., Zhi, X., Qiu, M.: Modeling for CPU-intensive applications in cloud computing. In: 17th International Conference on High Performance Computing and Communications, 7th International Symposium on Cyberspace Safety and Security, and 12th International Conference on Embedded Software and Systems, pp. 20–25. IEEE (2015). https://doi.org/10.1109/HPCC-CSS-ICESS.2015.128
67. Miano, J.: Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP. Addison-Wesley Professional, Boston (1999)
68. Seward, J.: Bzip2 and libbzip2. Web page at http://www.bzip.org (1996). Accessed 10 Apr 2021
69. Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: PIT: a practical mutation testing tool for Java (demo). In: International Symposium on Software Testing and Analysis (ISSTA), pp. 449–452. ACM (2016). https://doi.org/10.1145/2931037.2948707
70. Derezińska, A., Hałas, K.: Analysis of mutation operators for the python language. In: Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland, pp. 155–164. Springer (2014). https://doi.org/10.1007/978-3-319-07013-1_15



Pablo C. Cañizares is Assistant Professor at the Universidad Autónoma of Madrid. He was honoured in 2020 with the Extraordinary PhD Award from Universidad Complutense of Madrid and the Best PhD Thesis Award from the Spanish Society of Software Engineering and Software Development Technologies. He has published 25 papers in refereed journals and international venues. His research interests include modelling and testing distributed systems, specifically the use of metamorphic testing and model-driven engineering to model and check the correctness of complex systems.

Alberto Núñez received the M.Sc. degree in computer science from Carlos III University of Madrid, Spain, in 2005 and the Ph.D. degree in computer science from the same university, in 2011. He is currently an Associate Professor with the Software Systems and Computation Department, Complutense University of Madrid, Spain. He has published more than 40 research papers in journals, and national and international conferences. He regularly serves in the Program Committee of conferences such as SAC and MET. His research interests include formal testing, performance analysis and modeling of cloud systems, especially on how to perform models and simulations. Dr. Núñez won the IBM Ph.D. Fellowship award in 2009.



Rosa Filgueira has recently joined the School of Computer Science of the St. Andrews University as Associate Professor. Her research expertise is on improving the HPC applications' scalability and performance having contributed to several European and national projects in hazard forecasting and parallel processing. During the VERCE project she contributed to the design and optimization of dispel4py and pioneered several dispel4py applications. Currently, she is leading requirements capture for the ENVRIplus project (funded by EU Horizon2020) delivering common data functionality for 22 pan-European Research Infrastructures.

Juan de Lara is Full Professor at the Universidad Autónoma of Madrid, where he works in model-driven engineering. He has published more than 200 papers in international journals and conferences and has been the PC co-Chair of MODELS20, SLE20, ICMT12, FASE12, and ICGT17. He is associate editor of the Journal on Software and Systems Modeling (Springer) and the Journal of Object Technology.