OUTRIDER: Optimizing the mUtation Testing pRocess In Distributed EnviRonments

Pablo C. Cañizares[1], Alberto Núñez[1], and Juan de Lara[2]

[1] Universidad Complutense de Madrid, Madrid, Spain.
[2] Universidad Autónoma de Madrid, Madrid, Spain.
pablocc@ucm.es, alberto.nunez@pdi.ucm.es, Juan.deLara@uam.es

Abstract

The adoption of commodity clusters has been widely extended due to its cost-effectiveness and the evolution of networks. These systems can be used to reduce the long execution time of applications that require a vast amount of computational resources, and especially of those techniques that are usually deployed in centralized environments, like testing. Currently, one of the main challenges in testing is to obtain an appropriate test suite. Mutation testing is a widely used technique aimed at generating high quality test suites. However, the execution of this technique requires a high computational cost.

In this work we propose `OUTRIDER`, an HPC-based optimization that contributes to bridging the gap between the high computational cost of mutation testing and the parallel infrastructures of HPC systems aimed to speed-up the execution of computational applications. This optimization is based on our previous work called `EMINENT`, an algorithm focused on parallelizing the mutation testing process using MPI. However, since `EMINENT` does not efficiently exploit the computational resources in HPC systems, we propose 4 strategies to alleviate this issue. A thorough experimental study using different applications shows an increase of up to 70% performance improvement using these optimizations.

Keywords: Parallel and Distributed Computing, High Performance Computing, Mutation testing

1 Introduction

Wired communications have experienced a notorious growth with the use of fiber optic, reaching in experimental environments a speed of 560 Gbit/s [8]. The low latency and very high bandwidth of these networks have made commodity clusters a cost-effective solution, currently being the main source for high performance computing (in short, HPC). This trend has enabled advances in scientific computing by using HPC techniques [12]. Moreover, techniques that require a long execution time and are usually executed in centralized environments, like testing, can be deployed in clusters to reduce its high computational cost [1].

Nowadays, testing is one of the most extended mechanisms to check the correctness of software [5]. However, in order to properly check the validity of a program, an appropriate test suite (in short, TS) needs to be created, which in most cases is a difficult and challenging task. Moreover, a large TS requires a very long execution time. Fortunately, there are mechanisms, like mutation testing (in short, MT), focusing on improving the quality of a TS [2]. MT is based on applying mutation operators, each producing small syntactic changes in the program under test. This way, a set of *mutants* are produced. The idea is that if a TS is able to distinguish between the program under test and the generated mutants[1], it should be good at detecting a faulty implementation. Thus, the effectiveness of a TS is established on the number of mutants that can be distinguished from the original program (i.e., the mutants it kills).

Recently, MT has been successfully applied in several fields [3, 7]. However, it is computationally expensive because a large TS needs to be executed over a vast set of mutants. Hence, means to optimize and speed-up MT are required for its practical application to large projects.

In this paper we present `OUTRIDER`, an HPC-based optimization to improve the overall performance of the MT process. Our approach uses the `EMINENT` algorithm as a basis [1], which focuses on reducing the execution time of MT by parallelizing the testing process in HPC systems. However, since `EMINENT` does not properly exploit the resource usage in HPC systems, we propose four optimization strategies:

- Parallelizing the execution of the TS over the original application. While existing works in the literature sequentially execute the TS over the original application, we propose to distribute the test cases among different processes to be executed in parallel.
- Sorting the TS using the execution time of each test case. Since the TS is executed over the original application in the initial phase of the testing process, the time required to execute each test case can be collected. In the next phase of the testing process, where the TS is executed over each mutant, this information can be used to specify the order of execution for each test case. The rationale for starting with the shorter tests cases first is that, assumming that each test has equal probability to kill a mutant, executing more test cases maximizes the detection probability in a given time spam.
- Enhancing the test case distribution. This strategy maximizes the number of different mutants executed in parallel. A mutant is executed in parallel when different test cases are executed over this mutant, in different processes, at the same time. In this case, the resource usage efficiency may decrease if the test case that kills the mutant is executed in parallel with other test cases. Consequently, the execution of those test cases that do not kill the mutant is useless for obtaining the final results and, therefore, the computational resources are not efficiently used.
- Grouping cloned and equivalent mutants. Two mutants are clones when the resulting compiled binaries are identical. A mutant is considered equivalent to the original program when there is no test case that can kill it. The goal of this strategy is to avoid the complete execution of the TS on both equivalent and cloned mutants.

The rest of the paper is organized as follows. Section 2 presents the state of the art. Section 3 describes `OUTRIDER`. Section 4, presents some performance experiments to analyse the suitability of the proposed optimization. Finally, Section 5 ends with the conclusions and future work.

[1]A test case within a TS is said to *kill* the mutant if it signals an error when executed over the mutant.

2 State of the art

Since the first contributions in MT, there has been a constant effort to alleviate its high computational cost. As a result, different works aimed at improving the performance of the MT process can be found in the literature. These contributions can be classified in two main groups.

The first group focuses on reducing the total number of mutants without losing a significant effectiveness. The most relevant include *mutant sampling*, a technique based on randomly selecting a subset of mutants [18]; *mutant clustering*, which selects a collection of mutants by using clustering algorithms [9]; and *high order mutation*, that generates a reduced set of mutants, created by applying multiple mutation operators [19].

The second group consists of contributions focusing in reducing the execution time of the MT process. In this field, there are different proposals based on shared-memory, which can be divided in two different categories: Single Instruction Multiple Data systems [4] and Multiple Instruction Multiple Data systems [6, 17]. The main issue with this kind of systems is the lack of scalability in the number of processors and in the memory system.

Moreover, there are multiple contributions based on distributed memory [15, 16]. The contribution of Mateo and Usaola is worth mentioning. They presented a dynamic distribution algorithm, called PEDRO (Parallel Execution with Dynamic Ranking and Ordering) that uses Factoring Self-Schedulling ideas [15]. Also, they introduce *Bacterio*^{P}, a parallel extension of the MT tool *Bacterio* [14], which uses *Java-RMI*in order to communicate processes through the network. However, although the performance and the scalability achieved in this contribution is better than those obtained in shared-memory approaches, the used communication mechanism acts as a system bottleneck, limiting the overall system performance [13].

In our previous work we proposed `EMINENT` [1], a dynamic distributed algorithm focused on HPC systems and designed to reduce the high computational cost of MT. In `EMINENT` we use MPI to interchange information between processes, which alleviates the previously described bottleneck issue [13]. However, the distribution of the workload can be improved in order to increase both the resource usage efficiency and the level of parallelism. `OUTRIDER` is the optimization we propose in this paper to achieve this goal.

3 Description of `OUTRIDER`

In this paper we propose an optimization, called `OUTRIDER`, bridging the gap between one of the main limitations of MT, its high computational cost, and the main advantages provided by HPC systems, parallel infrastructures to speed-up the execution of computational applications. Next, we present the 4 strategies of `OUTRIDER` to improve the performance of the MT process.

3.1 Parallelizing the execution of the TS over the original application

Usually, the sequential execution of a TS over the original program is an issue that hampers the scalability of the MT process [15]. This becomes specially relevant in those cases where the application under test requires a long execution time and when the TS consists of a large number of test cases. Consequently, the scalability of the system is compromised due to the lack of parallelism, which is generally reflected in a low system performance.

In order to alleviate this issue, we propose exploiting the resources of the system by executing the TS over the original program in parallel. Basically, this strategy consists in distributing the execution of each test over the original program among different processes, which are executed in the available CPU cores of the system.

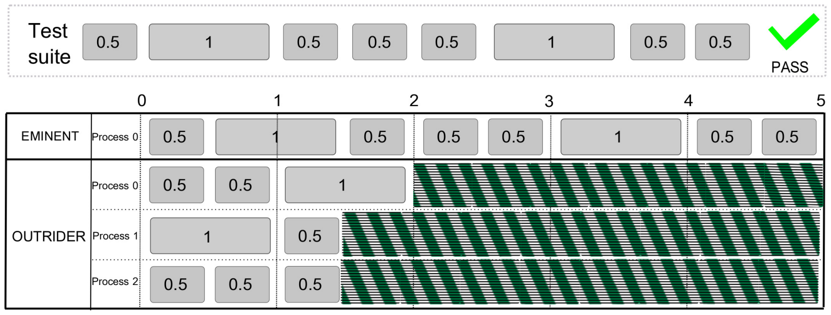


Figure 1: Execution of a TS over the original program using `EMINENT` and `OUTRIDER`

Figure 1 presents an example comparing the execution of a TS over the original program using `EMINENT` and `OUTRIDER`. The schema at the top shows the sequential execution of the TS, where each rectangle represents a test case and the number inside it shows the time slots required for its execution. The TS consists of 8 test cases and has a total duration of 5 time slots. While the TS is sequentially executed in one processor using `EMINENT`, `OUTRIDER` parallelizes the execution of the TS using 3 processes. Notably, this example shows that the distribution of the TS improves the overall performance, obtaining a speed-up of 2.5.

3.2 Sorting the TS

In MT, test cases are executed over a mutant until the mutant is killed or the TS is completely executed, and the mutant remains alive. It is therefore desirable that the mutant gets killed as soon as possible. Unfortunately, we cannot determine which test cases will kill the mutant before executing them. However, we can use information gathered from the execution of the TS over the original program, like the execution time of each test case.

This strategy uses this information to specify the execution order of the test cases. Thus, test cases are sorted by execution time as sorting criteria. As a result, the fastest test case is processed in the first place, while the slowest test case is executed last. The idea is to minimize the required time to kill a mutant. Although sorting the TS has a computational cost, we assume that applying this strategy would reduce the overall execution time.

3.3 Enhancing the test case distribution strategy

In order to increase both the level of parallelism and the resource usage efficiency, we propose a strategy that improves the workload distribution presented in `EMINENT`, which additionally considers the number of remaining mutants to be completely executed, the number of processes involved in the testing process and the number of processes that are executing each mutant. The idea is to improve the resource usage by maximizing the number of different mutants executed in parallel. Thus, when the number of remaining mutants to be executed is greater or equal than the number of available processes, each single process executes a different mutant. Otherwise, the remaining mutants to be completely processed are proportionally distributed among the available processes.

Figure 2 shows an example comparing two different distribution strategies. In this example a TS consisting of 3 test cases is executed over 4 mutants using 2 processes, each one having a dedicated CPU core. The schema at the top depicts the execution of the TS over each mutant,

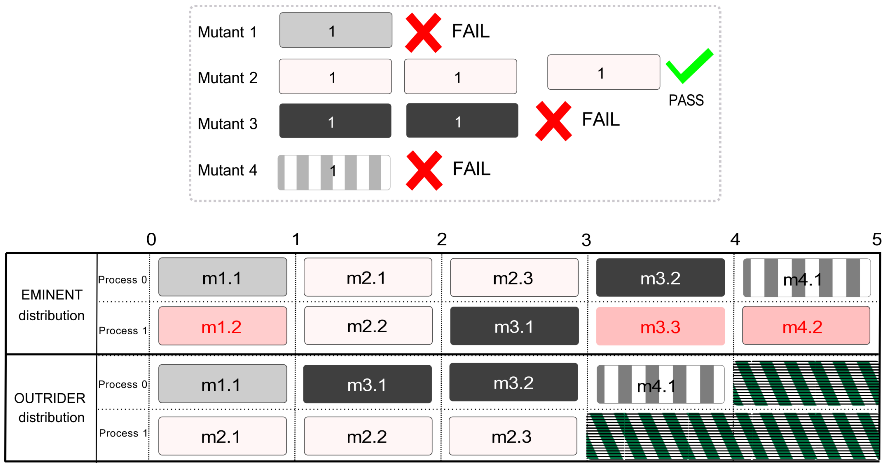


Figure 2: Workload distribution using `EMINENT` and `OUTRIDER`

where test case 1 kills mutant 1 and 4, test case 2 kills mutant 3, and mutant 2 remains alive. The schema at the bottom shows the strategies used by `EMINENT` and `OUTRIDER` to distribute the workload in the MT process. We denote by mX.Y the execution of the test case Y over the mutant X. In this scenario, executions m1.1, m3.2 and m4.1 kill the processed mutant.

The distribution strategy used in `EMINENT` shows that the execution of some test cases is useless. For instance, m1.1 and m1.2 are executed in parallel. Although the former execution kills mutant 1, process 1 is wasting computational resources by executing m1.2, which is not necessary to kill the mutant. Since the strategy used in `OUTRIDER` maximizes the number of different mutants executed in parallel, this situation is avoided in most scenarios. In this example, `OUTRIDER` obtains an improvement of 20% in the total execution time.

3.4 Categorizing equivalent mutants using TCE

In MT, a mutant is considered equivalent to the original program when it can not be distinguished from it through testing. The equivalence problem is one of the main obstacles in the practical use of MT. Although it is well known that deciding whether two programs are equivalent is a non-decidable problem [10], there are several heuristics that aid in finding patterns to identify this kind of mutants. In this case, due to its simplicity and its computational efficiency, we have selected the trivial compiler equivalence technique (in short, TCE), to detect both equivalent and cloned mutants [11]. This technique uses compiler optimizations to detect patterns that help identifying the equivalence between programs using a black-box scheme.

Hence, in this strategy we detect two kinds of mutants. On the one hand, mutants that are equivalent to the original program, known as *equivalents*. On the other hand, mutants that differ from the original program but are equal to other mutants, called *cloned* mutants.

We apply this technique after the compilation phase, where both equivalent and cloned mutants are detected. Mutants identified as equivalents are discarded and none of them are executed. On the contrary, cloned mutants are grouped in domains, where a single mutant is selected as *representative* of the domain. During the testing phase, only those mutants that do not belong to a domain are executed, which are handled as usual. Next, for each domain, only representative mutants are processed. Once the execution of a representative mutant ends, if the mutant is killed, only the killer test is applied to the rest of the mutants of the domain, which substantially reduces the number of test case executions. On the contrary, if the representative mutant remains alive, the rest of the mutants of the domain are managed as usual.

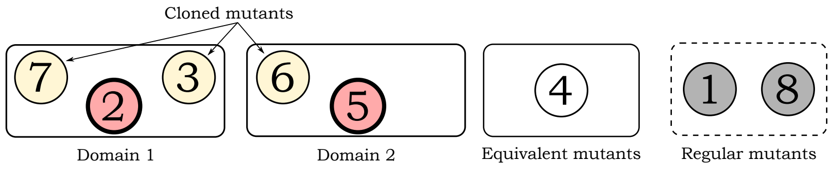


Figure 3: Categorization of cloned and equivalent mutants

Figure 3 shows an example with the execution of a MT process with 8 mutants. We applied our strategy to categorize these mutants, obtaining two different domains. Domain 1 consists of mutants 2, 3 and 7, and Domain 2 consists of mutants 5 and 6. Mutants with a bold border are the representatives of its domain (mutant 2 for Domain 1, and 5 for Domain 2). Mutant 4 has been detected to be equivalent, while mutant 1 and 8 are categorized as regular mutants.

Mutant ID	**Exec.Time**	**Time$_{EMI}$**	**Killer test time**	**Time$_{OUT}$**
1	432	432	-	432
2	245	677	-	677
3	245	922	46	723
4	456	1378	-	723
5	532	1910	-	1255
6	532	2442	164	1419
7	245	2687	46	1465
8	591	**3278**	-	**2056**

Table 1: Execution of 8 mutants using `EMINENT` and `OUTRIDER` with TCE

Following the example, Table 1 presents the execution time of the MT process. The first two columns, *Mutant ID* and *Exec.Time*, represent the mutant ID and its execution time, respectively. $Time_{EMI}$ refers to the accumulated time when the testing process is executed using `EMINENT`. The next column refers to the execution time of the test that kills the mutant, which is calculated from the representative mutant of each domain. Finally, $Time_{OUT}$ refers to the accumulated time when the testing process is executed using `OUTRIDER`.

These results show that `OUTRIDER` executes 37% faster than `EMINENT`. That is, while `EMINENT` requires 3278 seconds to completely execute the testing process, `OUTRIDER` requires 2056 seconds. This performance improvement is obtained because `OUTRIDER` executes less test cases than `EMINENT`. In this case, mutant 3, 6 and 7 are not completely executed because only the test case that kills them is executed instead.

4 Experiments

This section presents a thorough experimental study to analyze the scalability and performance of `OUTRIDER`. The mutant set used in these experiments has been created using the Milu mutation framework [6]. We use two different applications in the MT process. First, an image filtering application consisting of 3 algorithms to filter BMP images. Initially, all the images are located in a remote repository, which has a total size of 2,5 GB. In order to check this application, a TS consisting of 3200 test cases are executed over 250 mutants. The second is a CPU-intensive application, performing the multiplication of two large matrices. In this case, a TS consisting of 2000 test cases are executed over 100 mutants.

These experiments have been performed in a cluster of 9 nodes, 8 computing nodes and 1 storage node, interconnected through a Gigabit Ethernet network. Each node contains a Dual-Core Intel(R) Core(R) i5-3470 CPU at 3.4 Ghz with hyper-threading, 8 GB of RAM and 500GB HDD.

4.1 Performance evaluation of each single strategy in OUTRIDER

In this section, each proposed strategy in OUTRIDER is individually analyzed. We use the following notation: S1 refers to the strategy that parallelizes the TS execution over the original program (see Section 3.1), S2 refers to the strategy that sorts the TS (see Section 3.2), S3 is the strategy that enhances the test case distribution (see Section 3.3) and S4 is the strategy that categorizes both cloned and equivalent mutants (See Section 3.4).

Figure 4 shows the overall speed-up in EMINENT and OUTRIDER, with respect to a sequential execution, using 2, 4, 8, 16 and 32 processes. Each process is always executed in a dedicated CPU core. In the charts the X-axis represents the number of processes and the Y-axis represents the speed-up. The first row of charts shows the results of the image filtering application, while the second row represents the results of the CPU-intensive application. Each chart analyzes a different strategy of OUTRIDER. That is, charts 4(a) and 4(e) analyze S1, charts 4(b) and 4(f) evaluate S2, charts 4(c) and 4(g) analyze S3 and charts 4(d) and 4(h) evaluate S4.

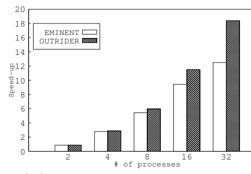
In general terms, OUTRIDER outperforms EMINENT in the major part of the evaluated scenarios. There is only one scenario where EMINENT executes faster than OUTRIDER, that is, testing the filtering application using the strategy S2 in OUTRIDER (see chart 4(b)). In this case, the major part of the mutants are killed by the first test cases of the TS without applying S2 and consequently, EMINENT requires less time to kill the mutants than OUTRIDER using a sorted TS.

The strategy that provides the best results is S1, reaching an improvement with respect to EMINENT of 40% in the overall execution time. This result is obtained using 32 processes for testing the CPU-intensive application (see chart 4(e)). Strategy S3 also provides valuable results, reaching in some scenarios an improvement between 10% - 38% in the overall performance. However, strategy S4 provides only slightly better results than EMINENT. Its best scenario is the testing process of the image filtering application, where 2 equivalent and 19 cloned mutants, divided in 13 domains, have been detected, obtaining a reduction of 20% in execution time.

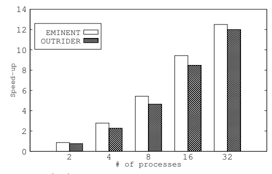
In conclusion, for the tested applications, OUTRIDER provides better resource usage efficiency than EMINENT. These experiments show that strategies S1 and S3 clearly provide a significant improvement in the overall system performance. Moreover, in terms of scalability – the performance obtained when the computational resources are increased – these strategies are better than S2 and S4. The main reason of this behaviour is two-fold. First, strategies S1 and S3 are less dependent of both the TS and the mutant set, which mainly focus on exploiting the resources of the system. Second, strategies S2 and S4 strongly depend of the TS and the mutant set, respectively. In particular, OUTRIDER using S2 and S4 executes faster than EMINENT only in those cases where the killer test is not located in the first positions of the TS and the mutant set contains several cloned and equivalent mutants.

4.2 Performance evaluation using different strategies in OUTRIDER

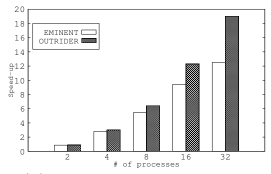
Next, we analyze the performance of OUTRIDER when different strategies are used. We only show the configurations obtaining the most representative results, which are depicted in Table 2.



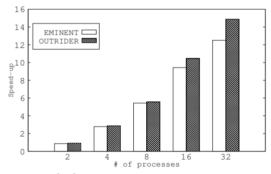
(a) S1: Parallelizing TS



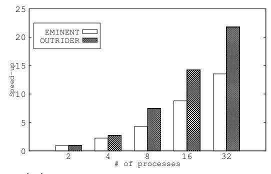
(b) S2: Sorting TS



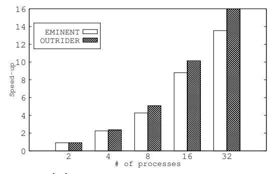
(c) S3: Distributing TS



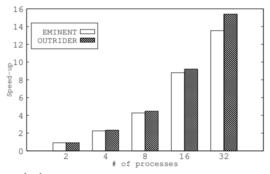
(d) S4: Grouping



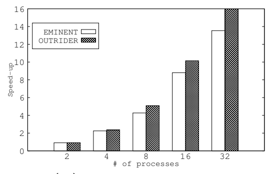
(e) S1: Parallelizing TS



(f) S2: Sorting TS



(g) S3: Distributing TS



(h) S4: Grouping

Figure 4: Performance of the testing process using EMINENT and OUTRIDER with a single strategy

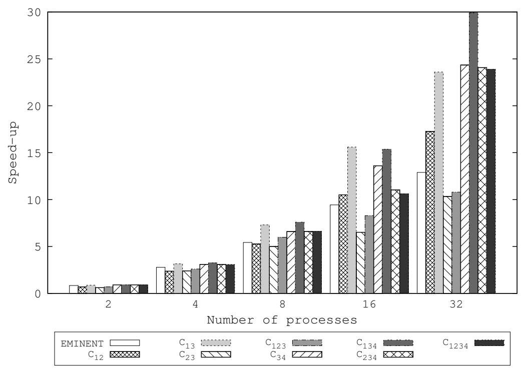
	Configuration							
Strategy	C_{12}	C_{13}	C_{23}	C_{123}	C_{34}	C_{134}	C_{234}	C_{1234}
S1: Parallelizing TS	✓	✓		✓		✓		✓
S2: Sorting TS	✓		✓	✓			✓	✓
S3: Distribution		✓	✓	✓	✓	✓	✓	✓
S4: Grouping					✓	✓	✓	✓

Table 2: Configuration of the strategies used in OUTRIDER

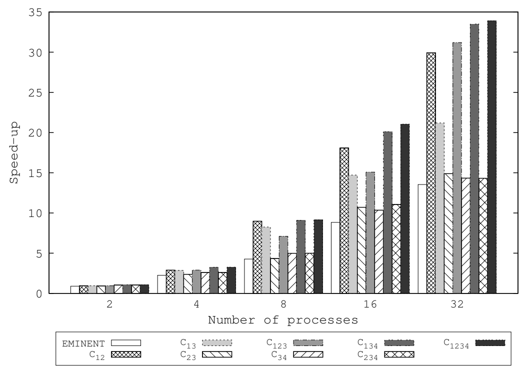
Figure 5 shows the results of executing the testing process with EMINENT and OUTRIDER using different configurations. In these charts, the X-axis shows the number of processes used and the Y-axis shows the obtained speed-up with respect to a sequential execution.

Figure 5(a) presents the results of the testing process using the image filtering application. In general, OUTRIDER achieves better performance than EMINENT. Both approaches provide similar performance when 2 and 4 processes are used. However, when the number of computational resources increases, the difference of performance between EMINENT and OUTRIDER increases as well. For instance, when 32 processes are used, OUTRIDER with $C_{13}, C_{34}, C_{134}, C_{1234}$ obtains a reduction in the total execution time, with respect to EMINENT, of 50%, 50.5%, 60% and 50%, respectively. Configuration C_{134} is particularly relevant. In this case, OUTRIDER achieves a speed-up of 2.3 with respect to EMINENT. This improvement in the overall testing performance is mainly reached because of the S4 strategy, which accelerates the MT process by avoiding the complete execution of some mutants (see Section 3.4). However, these configurations using the strategy S2 do not guarantee an improvement in the total execution time. For instance, OUTRIDER using C_{23} executes 19% slower than EMINENT.

Figure 5(b) shows the results of the testing process using the CPU-intensive application. Similarly to the previous experiments, we obtain a similar tendency in the system scalability. That is, using few computational resources provides almost the same performance for both approaches. However, increasing the number of computational resources provides a proportional improvement in the overall system performance. In these experiments, all the configurations used in OUTRIDER provide a better performance than EMINENT. In particular, $C_{12}, C_{123}, C_{134}, C_{1234}$ are especially relevant because OUTRIDER using these configurations exe-



(a) Image filter application



(b) Computational intensive application

Figure 5: Performance evaluation of EMINENT and OUTRIDER using different strategies

cutes 62%, 67%, 70% and 71% faster than EMINENT, respectively.

In general, the overall system performance is increased when combining different strategies. Configuration C_{134} achieves a significantly better speed-up than EMINENT for the image filtering application, especially when 32 processors are used. However, configuration C_{1234} only achieves slightly better results than C_{134} when 8 and 16 processors are used to check the CPU-intensive application. It is important to remark that the best results are obtained when different strategies are combined using OUTRIDER, especially strategies S1 and S4. In conclusion, OUTRIDER provides a better resource usage efficiency than EMINENT, which is reflected in the scalability obtained, reaching in some cases a speed-up higher than the number of CPU cores used.

Some aspects may affect the validity of the experiment and our conclusions. Regarding internal validity, we tried to eliminate any possible bias: for the mutants generated we used Milu, developed by a third party, and for the test cases we used random generation. For the cluster configuration we used a typical configuration where each node mounts, through NFS, the home folder from a storage node that contains the user data. Regarding the generalizability of the results (external validity), we only used two applications, but these characterize a reasonable class of programs: one was data-intensive, while the other was CPU-intensive.

5 Conclusions

In this paper we have presented OUTRIDER, an HPC-based optimization for the MT process in HPC systems. This optimization consists of 4 strategies aimed at improving the resource usage efficiency, which uses EMINENT as basis. Also, an experimental phase has been carried out to evaluates the effectiveness and scalability of OUTRIDER. Thes experiments show that OUTRIDER outperforms previous proposals to improve performance of the MT process. In general, OUTRIDER provides the best results when different strategies are combined, specially S1, S3 and S4, obtaining in some scenarios an improvement of 70% in the overall performance with respect to EMINENT. On the contrary, the results obtained when S2 is used shows that an improvement in the overall performance is not guaranteed. For instance, there are scenarios where OUTRIDER executes 66% faster than EMINENT (see C_{12} and C_{123} in Figure 5(b)), while there are other scenarios where OUTRIDER executes 20% slower than EMINENT (see C_{23} in Figure 5(a)).

As future work, we will evaluate the possibility to include mechanisms for automatically selecting the strategies to be applied in a given MT process.

Acknowledgements

Research partially supported by the Spanish MEC project DArDOS (TIN2015-65845-C3-1-R) and the Comunidad de Madrid project SICOMORo-CM (S2013/ICE-3006).

References

[1] P. C. Cañizares, M. G. Merayo, and A. Núñez. Eminent: Embarrassingly parallel mutation testing. In *ICCS*, volume 80, pages 63–73. Elsevier, 2016.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[3] L. Deng, A.J. Offutt, P. Ammann, and N. Mirzaei. Mutation operators for testing android apps. *Information and Software Technology*, 2016.

[4] R. Gopinath, C. Jensen, and A. Groce. Topsy-turvy: a smarter and faster parallelization of mutation analysis. In *ICSE Companion*, pages 740–743. ACM, 2016.

[5] R. M. Hierons, M. G., Merayo, and M. Núñez. Controllability through nondeterminism in distributed testing. In *ICTSS*, pages 89–105, 2016.

[6] Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *TAIC*, pages 94–98. IEEE, 2008.

[7] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *SCAM*, pages 147–156. IEEE, 2016.

[8] F. Li, J. Yu, Z. Cao, J. Zhang, M. Chen, and X. Li. Experimental demonstration of four-channel wdm 560 gbit/s 128qam-dmt using im/dd for 2-km optical interconnect. *J. of Lightwave Technology*, 2016.

[9] Y. Ma and S. Kim. Mutation testing cost reduction by clustering overlapped mutants. *J. of Systems and Software*, 115:18–30, 2016.

[10] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability*, 7(3):165–192, 1997.

[11] M. Papadakis, Y. Jia, M. Harman, and Y. Le-Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE*, volume 1, pages 936–946. IEEE, 2015.

[12] D. Penas et al. Parallel metaheuristics in computational biology: an asynchronous cooperative enhanced scatter search method. In *ICCS*, pages 630–639, 2015.

[13] K. Qureshi and H. Rashid. A performance evaluation of RPC, Java RMI, MPI and PVM. *Malaysian J. of Computer Science*, 18(2):38–44, 2005.

[14] P. Reales and M. Polo. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *ICSM*, pages 646–649. IEEE, 2012.

[15] P. Reales and M. Polo. Parallel mutation testing. *STVR Journal*, 23(4):315–350, 2013.

[16] I. Saleh and K. Nagi. Hadoopmutator: A cloud-based mutation testing framework. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, pages 172–187. Springer, 2014.

[17] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *ICSE*, pages 297–298. ACM, 2009.

[18] E. Wong. *On mutation and data flow.* PhD thesis, Purdue University, 1993.

[19] F. Wu, M. Harman, Y. Jia, and J. Krinke. Homi: Searching higher order mutants for software improvement. In *SSBSE*, pages 18–33. Springer, 2016.