Characterizing Fault-tolerance in Evolutionary Algorithms

Daniel Lombraña González, Juan Luis Jiménez Laredo , Francisco Fernández de Vega and Juan Julián Merelo Guervós

1 Introduction

Genetic Algorithms (GAs) and Genetic Programming (GP) are well known representatives of Evolutionary Algorithms (EAs), frequently used to solve optimization problems. Both require a large amount of computing resources when the problem faced is complex. The more complex the problem, the larger the computing requirements. This fact leads to a sometimes prohibitively long time to solution that happens, for example, when tackling real-world problems. In order to reduce the execution time of EAs, researchers have applied parallel and distributed programming techniques during the last decades.

There are two main advantages in exploiting the inherent parallelism of EAs: (i) the computing load is distributed among different processors, which improves the execution time, and (ii) the algorithm itself may suffer of structural changes allowing to outperform the sequential counterpart (see for instance [51]).

Parallel algorithms, and thus parallel GAs and GP, must be run on platforms that consists of multiple computing elements or *processors*. Although supercomputers can be employed, usually comodity clusters and distributed systems are used instead, due to both good performance and cheaper prices. One of the most popular distributed systems nowadays are the Desktop Grid Systems (DGSs). The term "desktop grid" is used to refer to distributed networks of heterogeneous single systems that contribute idle processor cycles for computing.

D . Lombraña
Citizen Cyberscience Centre
e-mail: teleyinex@gmail.com

J.L.J. Laredo and J.J. Merelo
University of Granada.
e-mail: {juanlu,jmerelo}@geneura.ugr.es

F. Fernández de Vega
Centro Universitario de Mérida, Universidad de Extremadura.
Sta. Teresa Jornet, 38. 06800 Mérida (Badajoz), Spain. e-mail: fcofdez@unex.es

Perhaps the most well known desktop grid system is the Berkeley Open Infrastructure for Network Computing (BOINC) [4], which supports among other projects the successful Einstein@Home [29] . DGSs are also known as *volunteer grids* because they aggregate the computing resources (commodity computers from offices or homes) that volunteers worldwide willingly donate to different research projects (such as Einstein@Home).

One of the most important features of DGSs is that they provide large-scale parallel computing capabilities, only for specific types of applications –bag of tasks mainly–, at a very low cost. Therefore DGSs can provide parallel computing capabilities for running demanding parallel applications, which is frequently the case of EAs. A good example of the combination of PEAs and DGSs is the Milky-Way@Home project [13].

But with large scale comes a higher likelihood that processors suffer a failure [44], interrupting the execution of the algorithm or crashing the whole system (in this chapter we use the term "failure" and do not make the subtle distinction between "failure" and "fault", which is not necessary for our purpose). Such an issue is characteristic of DGSs: computers join the system, contribute some resources and leave it afterwards causing a collective effect known as churn [46]. Churn is an inherent property of DGSs and has to be taken into account in the design of applications, as these interruptions (computer powered off, busy CPU, etc.) are interpreted by the application as a failure.

To cope with failures, researchers have studied and developed different mechanisms to circumvent the failures or restore the system once a failure occurs. These techniques are known as *Fault-Tolerance mechanisms* and enforce that an application behave in a well-defined manner when a failure occurs [20]. Nevertheless, not many efforts have been applied to study the fault tolerance features of PEAs in general, and of PGAs and PGP in particular.

In previous works [36, 27] we firstly analyzed the fault-tolerance nature of Parallel Genetic Programming (PGP) under several simplified assumptions. These initial results suggested that PGP exhibits a fault-tolerant behavior by default, encouraging to go a step further and run PGP on large-scale computing infrastructures that are subject to failures without requiring the employment of any fault-tolerance mechanism. This work was lately improved [22, 23, 25] by studying the fault-tolerance nature of PGP and PGAs using real data from one of the most high churn distributed systems: the Desktop Grids. The results again showed that PGP and PGAs can cope with failures without using any fault-tolerance mechanism, concluding that PGP and PGAs are fault tolerant by nature since it implements by default the fault-tolerance mechanism called *graceful degradation* [21].

This chapter is a summary of the main results obtained for PGAs and PGP regarding the study of fault-tolerance and their intrinsic fault-tolerant nature. To this aim, we have chosen a fine-grained master-worker model of parallelization [51]. A server, "the master", runs the main algorithm and hosts the whole population. The server is in charge of sending non-evaluated individuals to workers in order to obtain their fitness values. This approach is effective because one of the most time-consuming steps of GAs or GP is the evaluation –fitness computation– phase. The

master waits until all individuals in generation n are evaluated before going to the next generation $n+1$ and run the genetic operations.

We assume that the system only suffers from omission failures [21]:

- the master sends N individuals with $N > 0$ to a worker, and the worker never receives them, e.g., due to network transmission problems; or
- the master sends N individuals with $N > 0$ to a worker, the worker receives them but never returns them. This can occur because the worker crashes or the returned individuals are lost during the transmission.

In order to study the behavior of PGAs and PGP under the previous assumptions, we are going to simulate the failures using real-world traces of host availability from three DGSs. We have chosen Desktop Grid availability data because these systems exhibit large amounts of failures, and thus if it is possible to run inside them PGAs or PGP without using any fault-tolerance mechanism, PGAs and PGP will be able to exploit any parallel or distributed systems to its maximums.

The rest of the chapter is organized as follows. Section 2 reviews related work; section 3 describes main fault tolerance techniques. Section 4 presents the setup of the different scenarios and experiments; section 5 shows the obtained results and their analysis; and, finally, section 6 concludes the chapter with a discussion of the results and future directions.

2 Background and related work

When using EAs to solve real-world problems researchers and practitioners often face prohibitively long times-to-solution on a single computer. For instance, Trujillo *et al.* required more than 24 hours to solve a computer vision problem [53], and times-to-solution can be much longer, measured in weeks or even months. Consequently, several researchers have studied the application of parallel computing to Spatially Structured EAs in order to shorten times-to-solution [17, 51, 9]. Such PEAs have been used for decades, for instance, on the Transputer platform [6], or, more recently, via software frameworks such as Beagle [19], grid based tools like Paradiseo [41], or BOINC-based EA frameworks for execution on DGSs [38].

Failures in a distributed system can be local, affecting only a single processor, or they can be communication failures, affecting a large number of participating processors. Such failures can disrupt a running application, for instance imposing the application to be restarted from scratch. As distributed computing platforms become larger and/or lower-cost through the use of less reliable or non-dedicated hardware, failures occur with higher probability [43, 45, 54]. Failures are, in fact, the common case in DGSs. For this reason, fault-tolerant techniques are necessary so that parallel applications in general, and in our case PEAs, can benefit from large-scale distributed computing platforms. Failures can be alleviated, and in some cases completely circumvented, using techniques such as checkpointing [15], redundancy [26], long-term-memory [28], specific solutions to message-passing [2]

or rejuvenation frameworks [48]. It is necessary to embed the techniques in the application and the algorithms. While some of these techniques may be straightforward to implement (e.g., failure detection or restart from scratch), the most common ones typically lead to an increase in software complexity. Regardless, fault tolerance techniques always requires extra computing resources and/or time.

Currently, available PEA frameworks employ fault tolerant mechanisms to tolerate failures in distributed systems such as DGSs. For instance ECJ [40], ParadisEO [8], DREAM [7] or Distributed Beagle [19]. These frameworks have distinct features (programming language, parallelism models, etc.) that may be considered in combination with DGSs, and provide different techniques to cope with failures:

- ECJ [40] is a Java framework that employs a master-worker scheme to run PEAs using TCP/IP sockets. When a remote worker fails, ECJ handles this failure by rescheduling and restarting the computation to another available worker.
- ParadisEO [8] is a C++ framework for running a master-worker model using MPI [18], PVM [47], or POSIX threads. Initially, ParadisEO did not provide any fault-tolerance. Later on, developers implemented a new version on top of the Condor-PVM resource manager [42] in order to provide a checkpointing feature [15]. This framework, however, is not the best choice for DGSs because these systems are: (i) loosely coupled and (ii) workers may be behind proxies, firewalls, etc. making it difficult to deploy a ParadisEO system.
- DREAM [7] is a Java Peer-to-Peer (P2P) framework for PEAs that provides a fault-tolerance mechanism called *long-term-memory* [28]. This framework is designed specifically for P2P systems. As a result, it cannot be compared directly with our work since we focus on a master-worker architecture on DGSs.
- Distributed BEAGLE [19] is a C++ framework that implements the master-worker model using TCP/IP sockets as ECJ. Fault-tolerance is provided via a simple time-out mechanism: a computation is re-sent to one or more new available workers if this computation has not been completed by its assigned worker after a specified deadline.

While these PEA frameworks provide fault-tolerant features, the relationship between fault tolerance and specific features of PEAs has not been studied.

So far, EA researchers have not employed massively DGSs. Nevertheless, there are several projects using DGSs like the MilkyWay@Home project [13] which uses GAs to create an accurate 3D model of the Milky way, a ported version of LilGP [11](a framework for GP [14]) to one of the most employed DGSs, BOINC [4], or the *custom execution environment* facility proposed and implemented by Lombraña et. al. in [37, 24] for BOINC.

Other EA researchers have focused their attention on P2P systems [35], which are very similar to DGSs because the computing elements are also desktop computers in its majority. However these systems are different because there is not a central server as in DGSs.

In all the described proposals –to the best of our knowledge– none of them have specifically addressed the problem of failures within PGAs or PGP. Nevertheless, some of those solutions internally employ some fault-tolerance mechanisms. In this

sense, only Laredo et al. have analyze the resilience to failures of a parallel Genetic Algorithm in [34], following the Weibull degradation of a P2P system (failures are the host-churn behavior of these systems as well as DGSs) proposed by Stutzbach and Rejaie in [46]. Therefore, PGAs or PGP have not been analyzed before under real host availability traces (a.k.a. host-churn). Hence, this chapter assesses fault tolerance in PGAs and PGP using host-churn data collected in three real-world DGSs [30]. Therefore, the key contribution of this chapter is the full characterization of PGAs and PGP from the point of view of fault-tolerance with the aim of studying if PGAs can be run in parallel or distributed systems without using any fault-tolerance mechanism.

3 Fault Tolerance

Fault tolerance can be defined as the ability of a system to behave in a well-defined manner once a failure occurs. In this chapter we only take into account failures at the process level. A complete description of failures in distributed systems is beyond the scope of our discussion. In this section, we describe different failure models as well as different techniques to circumvent failures.

3.1 Failure Models

According to Ghosh [21], failures can be classified as follows: crash, omission, transient, Byzantine, software, temporal, or security failures. However, in practice, any system may experience a failure due to the following reasons [21]: (i) *Transient failures*: the system state can be corrupted in an unpredictable way; (ii) *Topology changes*: the system topology changes at runtime when a host crashes, or a new host is added; and (iii) *Environmental changes*: the environment – external variables that should only be read – may change without notice. Once a failure has occurred, a mechanism is required to bring back the system into a valid state. There are four major types of such fault tolerance mechanisms: masking tolerance, non-masking tolerance, fail-safe tolerance, and graceful degradation [21].

To discuss fault-tolerance in the context of PEAs, we first need to specify the way in which the GP or GA application is parallelized. Parallelism has been traditionally applied to GP and GAs at two possible levels: the individual level or the population level [51, 9, 50, 3]. At the individual level, it is common to use a master-worker scheme, while at the population level, a.k.a. the “island model”, different schemes can be employed (ring, multi-dimensional grids, etc.).

In light of previous studies [51, 50] and taking into account the specific parallel features of DGSs [31, 30], we focus on parallelization at the individual level. Indeed, DGSs are loosely-coupled platforms with volatile resources, and therefore ideally suited to and widely used for embarrassingly parallel master-worker applications.

Furthermore parallelization at the individual level is popular in practice because it is easy to implement and does not require any modification of the evolutionary algorithm [9, 50, 3].

The server, or "master", is in charge of running the main algorithm and manages the whole population. It sends non evaluated individuals to different processes, the "workers," that are running on hosts in the distributed system. This model is effective as the most expensive and time-consuming operation of the application is typically the individual evaluation phase. The master waits until all individuals in generation n are evaluated before generating individuals for generation $n+1$. In this scenario, the following failures may occur:

- *A crash failure* – The master crashes and the whole execution fails. This is the worst case.
- *An omission failure* – One or more workers do not receive the individuals to be evaluated, or the master does not receive the evaluated individuals.
- *A transient failure* – A power surge or lighting affects the master or worker program, stopping or affecting the execution.
- *A software failure* – The code has a bug and the execution is stopped either on the master or on the worker(s).

We make the following assumptions: (i) we consider all the possible failures that can occur during the transmission and reception of individuals between the master and each worker, but we assume that all software is bug-free and that there are no transient failures; (ii) the master is always in a safe state and there is no need for master fault tolerance (unlike for the workers, which are untrusted computing processes). This second assumption is justified because the master is under a single organization/person's control, and, besides, known fault tolerance techniques (e.g., primary backup [26]) could easily be used to tolerate master failures.

Our system only suffers from omission failures: (i) the master sends $N > 0$ individuals to a worker, and the worker never receives them (e.g., due to network transmission problems); or (ii) the master sends $N > 0$ individuals to a worker, the worker receives them but never returns them (e.g., due to a worker crash or to network transmission problems).

3.2 Fault-Tolerant and Non-Fault-Tolerant Strategies

Since our objective in this work is to study the implicit fault-tolerant nature of the PEA paradigm, we need to perform comparison with the use of a reasonable and explicit fault-tolerant strategy. In the master-worker scheme, four typical approaches can be applied to cope with failures:

1. Restart the computation from scratch on another host after a failure.
2. Checkpoint locally (with some overhead) and restart the computation on the same host from the latest checkpoint after a failure.

3. Checkpoint on a checkpointing server (with more overhead) and move to another host after a failure, restarting the computation from the last checkpoint.
4. Use task replication by sending the same individual to two or more hosts, each of them performing either 1, 2, or perhaps even 3 above. The hope is that one of the replicas will finish early, possibly without any failure.

Based on the analysis in Section 2 of existing PEA frameworks that are relevant in the context of DGSs, namely ECJ and Distributed Beagle, the common technique to cope with failures is the first one: re-send lost individuals after detecting the failure. The advantage of this technique is that it is low overhead, very simple to implement, and reasonably effective. More specifically, its modus-operandi is as follows:

1. After assigning individuals to workers, the master waits at most T time-units per generation. If all individuals have been computed by workers before T time-units have elapsed, then the master computes fitness values, updates the population, and proceeds with the next generation.
2. If after T time-units some individuals have not been evaluated, then the master assumes that workers have failed or are simply so slow that they may not be useful to the application. In this case:
   a. individuals that have not been evaluated are resent for evaluation to available workers, and the master waits for another T time-units for these individuals to be evaluated.
   b. If there are not enough available workers to evaluate all unevaluated individuals, then the master proceeds in multiple phases of duration T. For instance, if after the initial period of T time-units there remain 5 unevaluated individuals and there remain only 2 available workers, the master will use $\lceil \frac{5}{2} \rceil = 3$ phases (assuming that all future individual evaluations are successful).

This method provides a simple fault-tolerant mechanism for handling worker failures as well as slow workers, which is a common problem in DGSs due to high levels of host heterogeneity [4, 21, 5]. For the sake of simplicity, we make the assumption that individuals that are lost and resent for evaluation to new workers are always evaluated successfully. This is unrealistic since future failures could lead to many phases of resends. However, this assumption represents a best-case scenario for the fault-tolerant strategy. The difference between the failure-free and the failure-prone case is the extra time due to resending individuals. In the failure-free case, with G generations, the execution time should be $T_{executiontime} = G \times T$, while in a failure-prone case it will be higher.

By contrast with this fault-tolerant mechanism, we propose a simple non-fault-tolerant approach that consists in ignoring lost individuals, considering their loss just a kind of dynamic population feature [52, 16, 39, 32]. In this approach the master does not attempt to detect failures and no fault tolerance technique is used. The master waits a time T per generation, and proceeds to the next generation with the available individuals at that time, likely losing individuals at each generation. The

hope is that the loss of individuals is not (significantly) detrimental to the achieved, while the overhead of resending lost individuals for recomputation is not incurred.

4 Experimental methodology

All the experiments presented in this chapter are based on simulations. Simulations allows us to perform a statistically significant number of experiments in a wide range of realistic scenarios. Furthermore, our experiments are repeatable, via "replying" host availability trace data collected from real-world DG platforms [30], so that fair comparison between simulated application execution is possible.

4.1 Application and failure model

We perform experiments for one GA and one GP well-known problems. The GP problem is the even parity 5 (EP5) problem, which tries to build a program capable of calculating the parity of a set of 5 bits. For the GA problem we used the 3-trap instance [1], which is a piecewise-linear function defined on unitation (the number of ones in a binary string).

Two kind of experiments are carried out:

1. for the failure-free case (i.e. assuming no worker failures occur)
2. replying and simulating failure traces from real-world DGSs.

In the failure free case the available computing power is kept steady throughput execution, while in the other cases it varies among generations.

The simulation of host availability in the DG is performed based on three real-world traces of host availability that were measured and reported in [30]: *ucb*, *entrfin*, and *xwtr*. These traces are time-stamped observations of the host availability in three DGSs. The *ucb* trace was collected for 85 hosts in a graduate student lab in the EE/CS Department at UC Berkeley for about 1.5 months. The *entrfin* trace was collected for 275 hosts at the San Diego Supercomputer Center for about 1 month. The *xwtr* trace was collected for 100 hosts at the Université Paris-Sud for about 1 month. See [30] for full details on the measurement methods and the traces, and Tab. 1 for a summary of its main features.

Table 1 Features of Desktop Grid Traces

Trace	Hosts	Time in months	Place
entrfin	275	1	SD Supercomputer Center
ucb	85	1.5	UC Berkeley
xwtr	100	1	Université Paris-Sud

Figure 1 shows example availability data from the *ucb* trace: the number of available hosts in the platform versus time over 24 hours. The figure shows the typical churn phenomenon, with available hosts becoming unavailable and later becoming available again. We performed our experiments over such 24-hour segments of our availability traces.

We use two different scenarios when simulating host failures based on trace data. In the first scenario a stringent assumption is used: hosts that become unavailable never become available again. An example of the number of hosts throughout time for this scenario is shown in Figure 1, as the curve "trace without return." For this scenario, we arbitrarily select the time within each 24-hour segment with the largest number of available hosts as the beginning of application execution. In the second scenario hosts can become available again after a failure and reused by the application. This phenomenon is called "churn," and is typical of what is implemented in real-world DG systems. For this scenario, application execution starts at an arbitrary time in the segment. Note that in the first "no churn scenario," population size (i.e., the number of individuals) becomes progressively smaller as the application makes progress, while population size may fluctuates in the "churn scenario."

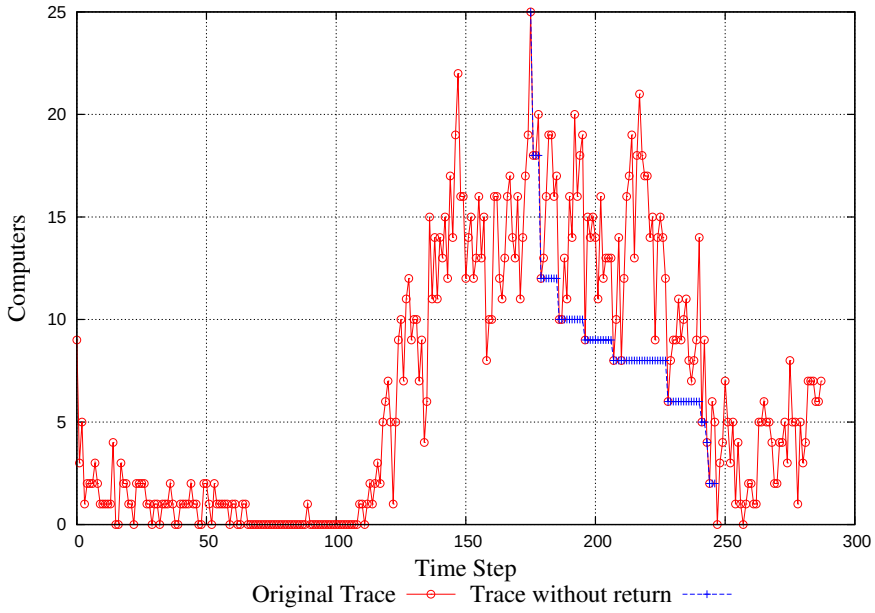


Fig. 1 Host availability for 1 day of the *ucb* trace.

4.2 Distribution of Individuals to Workers

At the onset of each generation the master sends as equal as possible numbers of individuals to each worker. This is because the master assumes homogeneous workers and thus strives for perfect load-balancing. We call this number I. Thus, if a worker does not return individual evaluations before time T, then those I individuals are considered lost. In the fault-tolerant approach in Section 3.2, such individuals are simply resent to other workers. In our non-fault-tolerant approach, these individuals are simply lost and do not participate in the subsequent generations.

Note that for our non-fault-tolerant approach the execution time per generation in the failure-free and the failure-prone case are identical: with P individuals to be evaluated at a given generation and W workers, the master sends $I = P/W$ individuals to each worker. When a worker fails I individuals are lost. Given that these individuals are discarded for the next generation and that the initial population size is never exceeded by new extra individuals, the remaining workers will continue evaluating I individuals each, regardless of the number of failures or newly available hosts.

Regardless of the approach in use, if there is host churn then population size can be increased throughout application execution due to newly available hosts. We impose the restriction that the master never overcomes the specified population size. This may leave some workers idle in case a large number of workers become available. In this case, it would be interesting to re-adjust the number I of individuals sent to each worker so as to utilize all the available workers. We leave such load-balancing study outside the scope of this work and maintain I constant.

In the churn scenario, one important question is: what work is assigned to newly available workers? When a new worker appears, the master simply creates I new random individuals and increases the population size accordingly (provided it remains below the initial population size). These new individuals are sent to the new worker. Note that when there are no available workers at all, the master loses all its individuals except the best one thanks to the elitism parameter. Then, the master proceeds to the next generation by waiting the specified time T for newly available workers.

4.3 Experimental Procedure

We have performed a statistical analysis of our results based on 100 trials for each experiment, accounting for the fact that different individuals can be lost depending on which individuals were assigned to which hosts. We have analyzed the normality of the results using the Kolgomorov-Smirnov and Shapiro-Wilk tests, finding out that all results are non-normal. Therefore, to compare two samples, the failure-free case with each trace (with and without churn), we used the Wilcoxon test. Table 5 and **??** present the Wilcoxon analysis of the data. The following sections discuss these results in detail.

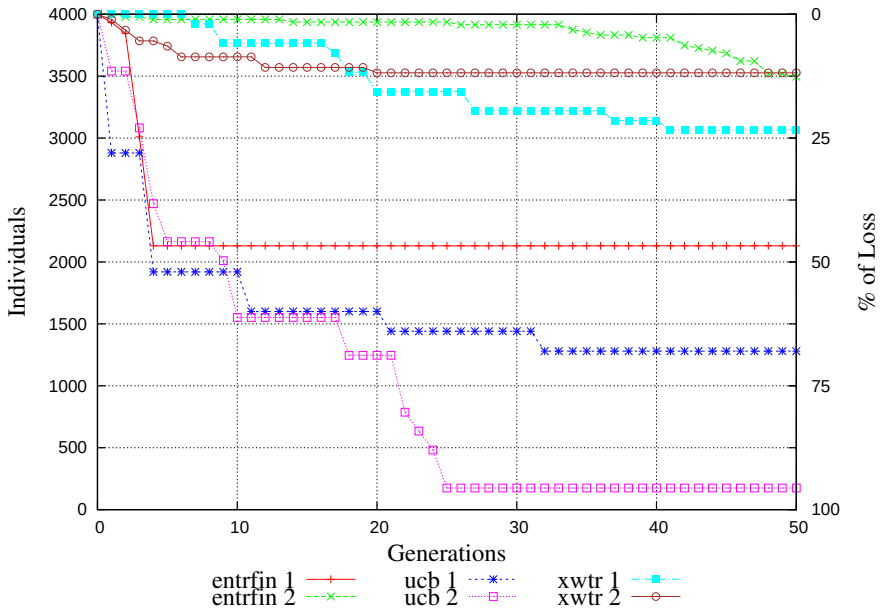


Fig. 2 Population size vs. generation.

5 Experimental results

5.1 GP: Even Parity 5

For the GP problem, fitness is measured as the error in the obtained solution, with zero meaning that a perfect solution has been found. All the GP parameters, including population sizes, are Koza-I/II standard [33]. See Table 2 for all details.

Table 2 Parameters of selected problems.

	EP5
Population	4000
Generations	51
Elitism	Yes
Crossover Probability	0.90
Reproduction Probability	0.10
Selection: Tournament	7
Max Depth in Cross	17
Max Depth in Mutation	17
ADFs	Yes

Even if the required time for fitness evaluation for the problems at hand is short, we simulate larger evaluation times representative of difficult real-world problems (so that 51 generations, the maximum, correspond to approximately 5 hours of computation in a platform without any failures).

5.1.1 EP5: Results without churn

In this section we consider the scenario in which hosts never become available again (no churn). Figure 2 shows the evolution of the number of individuals in each generation for the EP5 problem when simulated over two 24-hour periods, denoted by *Day 1* and *Day 2*, randomly selected out of each of three of our traces, *entrfin*, *ucb*, and *xwtr*, for a total of 6 experiments.

Table 3 shows a summary of the obtained fitness for the EP5 problems and of the fraction of lost individuals by the end of application execution. The first row of the table shows fitness values assuming a failure-free case. The fraction of lost individuals depends strongly on the trace and on the day. For instance, the *Day 1* period of the *entrfin* trace exhibits on its 10 first generations a severe loss of individuals (almost half); the *ucb* trace on its *Day 2* period loses almost the entire population after 25 generations (96.15% loss); and the *xwtr* exhibits more moderate losses, with overall 23.52% and 12.08% loss after 51 generations for *Day 1* and *Day 2*, respectively.

Table 3 Obtained fitness for EP5

		EP5
Trace	Loss(%)	Fitness
Error free	0.00	2.56
entrfin (Day 1)	48.02	3.58
entrfin (*Day 2*)	13.04	2.44
ucb (Day 1)	68.00	3.98
ucb (*Day 2*)	96.15	5.13
xwtr (Day 1)	23.52	2.78
xwtr (*Day 2*)	12.08	2.61

The obtained fitness in the failure-free case is 2.56, and it ranges from 2.44 to 5.13 for the failure-prone cases (see Table 5 for statistical significance of results). The quality of the fitness depends on host losses in each trace. The *entrfin* and *ucb* traces present the most severe losses. The *ucb* trace exhibits 68% losses for *Day 1* and 96.15% for *Day 2*. Therefore, the obtained fitness in these two cases are the worst ones relatively to the failure-free fitness. The *entrfin* trace exhibits 48.02% and 13.04% host losses for *Day 1* and *Day 2*, respectively. As with the previous trace, when losses are too high, as in *Day 1*, the quality of the solution is significantly worse than that in the failure-free case; when losses are lower, as in *Day 2*, the obtained fitness is not significantly far from the failure-free case. Similarly, the *xwtr*

trace with losses under 25% leads to a fitness that it is not significantly different from the failure-free case.

We conclude that, for the EP5 problem, it is possible to tolerate a gradual loss of up to 25% of the individuals without sacrificing solution quality. This is the case without using any fault tolerance technique. However, if the loss of individuals is too large, above 50%, then solution quality is significantly diminished. Since real-world DGSs do exhibit such high failure rates when running PGP applications, we attempt to remedy this problem. Our simple idea is to increase the initial population size (in our case by 10, 20, 30, 40, or 50%). The goal is to compensate for lost individuals by starting with a larger population.

Increasing population likely also affects the fitness in the failure-free case. We simulated the EP5 problem in the failure-free case with a population size increased by 10, 20, 30, 40 and 50%. Results are shown in Figure 5.1.1(a), which plots the evolution of fitness versus the "computing effort." The computing effort is defined as the total number of evaluated individuals nodes so far (we must bear in mind that GP individuals are variable size trees), i.e., from generation 1 to generation G, as described in [16]. We have fixed a maximum computing effort which corresponds to 51 generations and the population size introduced by Koza [33], which is employed in this work. Figure 5.1.1(a) shows that population sizes $M > 4,000$ for a similar effort obtain worse fitness values when compared with the original $M = 4,000$ population size. Thus, for static populations, increasing the population size is not a good option, provided a judicious population size is chosen to begin with. Nevertheless, we content that such population increase could be effective in a failure-prone case.

Table 4 EP5 fitness with increased population

Error Free fitness = 2.56						
Traces	*entrfin*	*ucb*	*xwtr*	*entrfin*	*ucb*	*xwtr*
+0%	3.58	3.98	2.78	2.44	5.13	2.61
+10%	3.52	3.75	2.40	2.65	5.21	2.66
+20%	3.01	3.61	2.32	2.29	4.68	2.42
+30%	3.13	3.33	2.46	2.36	4.50	2.33
+40%	2.80	3.35	2.15	2.01	4.71	1.96
+50%	2.85	3.17	2.13	1.92	4.47	2.24

Table 4 shows results for the increased initial population size, based on simulations for the *Day 1* and *Day 2* periods of all three traces. Overall, increasing the initial population size is an effective solution to tolerate failures while preserving (and even improving!) solution quality. For instance, for the *Day 1* period of the *entrfin* trace, with host losses at 48.02%, starting with 50% extra individuals ensures solution quality on par with the failure-free case. Similar results are obtained for the *entrfin* and *xwtr* two periods. Furthermore, for the *Day 2* period of traces *entrfin* and *xwtr*, adding 40% or 50% extra individuals results in obtaining solutions of better quality than in the failure-free case. However, the increase of the initial population is not enough for the *ucb* trace as its losses are as high as 96.15% and 68% for *Day 1*

and *Day 2*, respectively. Note that in these difficult cases the fault-tolerant approach does not succeed at all.

Table 5 EP5 fitness comparison between failure-prone and failure-free cases using Wilcoxon test (*Day 1 and 2*) – "not significantly different" means fitness quality comparable to the failure-free case.

Error Free fitness = 2.56

	Trace	Fitness	Wilcoxon Test	Significantly different?	Fitness	Wilcoxon Test	Significantly different?
Day 1	*entrfin*	3.58	W = 6726, p-value = 1.843e-05	yes	**2.44**	**W = 4778.5, p-value = 0.5815**	**no**
	entrfin 10%	3.52	W = 6685, p-value = 2.707e-05	yes	**2.65**	**W = 5201.5, p-value = 0.6167**	**no**
	entrfin 20%	3.01	W = 5760, p-value = 0.05956	yes	**2.29**	**W = 4571, p-value = 0.2863**	**no**
	entrfin 30%	3.13	W = 5942.5, p-value = 0.01941	yes	**2.36**	**W = 4732.5, p-value = 0.505**	**no**
	entrfin 40%	**2.80**	**W = 5355, p-value = 0.3773**	**no**	2.01	W = 4098, p-value = 0.02458	yes
	entrfin 50%	**2.85**	**W = 5620, p-value = 0.1233**	**no**	1.92	W = 3994.5, p-value = 0.01213	yes
	ucb	3.98	W = 7274, p-value = 1.789e-08	yes	5.13	W = 8735.5, p-value < 2.2e-16	yes
	ucb 10%	3.75	W = 6927.5, p-value = 1.799e-06	yes	5.21	W = 8735.5, p-value < 2.2e-16	yes
	ucb 20%	3.61	W = 6769, p-value = 1.123e-05	yes	4.68	W = 8266.5, p-value = 6.661e-16	yes
	ucb 30%	3.33	W = 6390, p-value = 0.0005542	yes	4.50	W = 8152, p-value = 6.439e-15	yes
	ucb 40%	3.35	W = 6408, p-value = 0.000464	yes	4.71	W = 8325.5, p-value = 2.220e-16	yes
	ucb 50%	3.17	W = 6080, p-value = 0.007298	yes	4.47	W = 8024.5, p-value = 6.95e-14	yes
	xwtr	**2.78**	**W = 5509, p-value = 0.2043**	**no**	**2.61**	**W = 5238.5, p-value = 0.5524**	**no**
	xwtr 10%	**2.40**	**W = 4762, p-value = 0.5532**	**no**	**2.66**	**W = 5215.5, p-value = 0.5927**	**no**
	xwtr 20%	**2.32**	**W = 4643.5, p-value = 0.3753**	**no**	**2.42**	**W = 4686.5, p-value = 0.4364**	**no**
	xwtr 30%	**2.46**	**W = 4802, p-value = 0.6221**	**no**	**2.33**	**W = 4611.5, p-value = 0.3336**	**no**
	xwtr 40%	**2.15**	**W = 4363, p-value = 0.1121**	**no**	1.96	W = 4033.5, p-value = 0.01574	yes
	xwtr 50%	**2.13**	**W = 4296.5, p-value = 0.08027**	**no**	2.24	**W = 4511, p-value = 0.2226**	**no**

Results with Host Churn

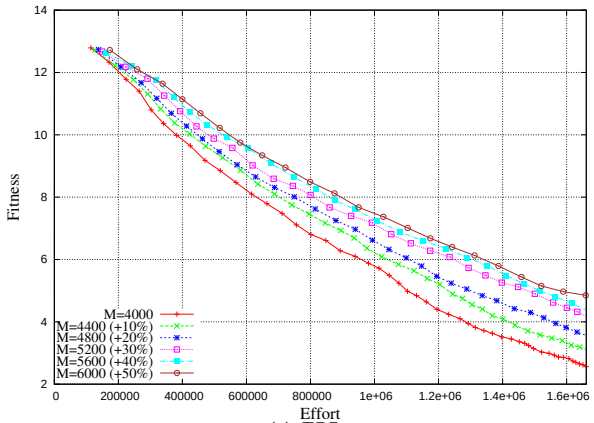
Day 1	*entrfin*	**2.86**	W = **5513.5, p-value=0.2012**	**no**	**2.75**	**W = 5404.5, p-value = 0.3142**	**no**
	ucb	8.87	W = 9997, p-value = 2.2e-16	yes	5.89	W = 9645, p-value < 2.2e-16	yes
	xwtr	**2.56**	**W = 4940, p-value = 08823**	**no**	**2.52**	**W = 5035, p-value = 0.9315**	**no**

From these results we conclude that increasing the initial population size is effective to maintain fitness quality at the level of that in the failure-free case. The fraction by which the population is increased is directly correlated to the host loss rate. If an estimation of this rate is known, for instance based on historical trends, then the initial population size can be chosen accordingly. Also, one must keep in mind that an increased population size implies longer execution time for each generation since more individuals must be evaluated.

5.1.2 EP5: Results with churn

In this section we present results for the case in which hosts can become available again after becoming unavailable, leading to churn. Recall from the discussion at the beginning of Section 4.2 that the population size is capped at 4,000 individuals (according to Table 2) and that each worker is assigned I individuals. Such individuals are randomly generated by the master when assigned to a newly available worker.

Table 6 shows the obtained fitness for the EP5 problem on all traces. It also shows the host churn represented by the minimum, median, mean, maximum, and variance of the number of available hosts during application execution. Among all the traces, the *ucb* trace is the worst possible scenario as it has very few available hosts. This prevents the master from sending individuals to workers, both in *Day 1* and *Day 2*,



(a) EP5

Fig. 3 Fitness vs. Effort with increased population for failure-free experiments

leading to poor fitness values. For the *entrfin* and *xwtr* traces, both for *Day 1* and *Day 2*, the obtained fitness value is comparable to that in the failure-free case (see Table 5 for statistical significance).

Table 6 Obtained fitness for EP5 with host churn

Trace	Hosts					Fitness
	Min.	Median	Mean	Max.	Var. (s^2)	EP5
Error free	-	-	-	-	-	2.56
entrfin (*Day 1*)	92	160	157.50	177	179.33	2.86
entrfin (*Day 2*)	180	181	181.30	183	0.75	2.75
ucb (*Day 1*)	0	1	1.51	9	2.21	8.87
ucb (*Day 2*)	0	2	2.57	7	4.29	5.89
xwtr (*Day 1*)	28	29	28.92	29	0.07	2.56
xwtr (*Day 2*)	86	86	86	86	0	2.52

If the variance of the number of available hosts for a trace is zero, then the trace is equivalent to the failure-free case, as the hosts do not experience any failure. The obtained fitness should then be similar to that in the failure-free case.

The *xwtr* trace, *Day 2*, exhibits such zero variance, and indeed the obtained fitness value is similar to that in the failure-free case (see Table 6). The variance of the *xwtr* trace, *Day 1*, is low at 0.07, and the obtained fitness is again on par with that in the failure-free case. The *entrfin* trace, *Day 1*, exhibits the largest variance. Nevertheless, the obtained fitness is better than that of its counterpart in the non-churn scenario, and similar to that in the failure-free case. This shows that re-acquiring hosts is, expectedly, beneficial. Finally the *ucb* trace leads to the worst fitness values despite its low variability (see Table 6). The reason is a low maximum number of

available hosts (9 and 7 for *Day 1* and *Day 2*, respectively), and many periods during which no hosts were available at all (in which case the master loses the entire population except for the best individual). As a result, it is very difficult to obtain solutions comparable to those in the failure-free case.

5.2 GA: 3-trap function

According to [12], 3-trap lies on the region between the deceptive 4-trap and the non-deceptive 2-trap having, therefore, intermediate population size requirements that Thierens estimates in 3000 for the instance under study in [49]. A trap function is a piecewise-linear function defined on unitation (the number of ones in a binary string). There are two distinct regions in the search space, one leading to a global optimum and the other one to the local optimum (see Eq. 1). In general, a trap function is defined by the following equation:

$$trap(u(\overrightarrow{x})) = \begin{cases} \frac{a}{z}(z - u(\overrightarrow{x})), & \text{if} \quad u(\overrightarrow{x}) \leq z \\ \frac{b}{l-z}(u(\overrightarrow{x}) - z), & \text{otherwise} \end{cases} \tag{1}$$

where $u(\overrightarrow{x})$ is the unitation function, a is the local optimum, b is the global optimum, l is the problem size and z is a slope-change location separating the attraction basin of the two optima.

For the following experiments, 3-trap was designed with the following parameter values: $a = l - 1$, $b = l$, and $z = l - 1$. Tests were performed by juxtaposing $m = 10$ trap functions in binary strings of length $L = 30$ and summing the fitness of each sub-function to obtain the total fitness. All settings are summarized in Table 7.

Table 7 Parameters of the experiments

Trap instance	
Size of sub-function (k)	3
Number of sub-functions (m)	10
Individual length (L)	30

GA settings	
GA	GGA
Population size	3000
Selection of Parents	Binary Tournament
Recombination	Uniform crossover, $p_c = 1.0$
Mutation	Bit-Flip mutation, $p_m = \frac{1}{L}$

In order to analyze the results with confidence, data has been statistically analyzed (each experiment has been run 100 times). Firstly, we analyzed the normality of the data using the Kolgomorov-Smirnov and Shapiro-Wilk tests [10], obtaining as a result that all data are non-normal. Thus, to compare two samples, the error-free

case with each trace, we used the Wilcoxon test (Tab. 8 shows the Wilcoxon analysis of the data).

Error Free fitness = 23.56

	Trace	Fitness	Wilcoxon Test	Significantly different?		Fitness	Wilcoxon Test	Significantly different?
Day 1	Entrfin	23.3	W = 6093, p-value = 0.002688	yes	Day 2	**23.57**	**W = 4979.5, p-value = 0.9546**	**no**
	Entrfin 10%	**23.47**	**W = 5408.5, p-value = 0.2535**	**no**		**23.69**	**W = 4397.5, p-value = 0.07682**	**no**
	Entrfin 20%	**23.48**	**W = 5360, p-value = 0.3137**	**no**		**23.67**	**W = 4522.5, p-value = 0.1645**	**no**
	Entrfin 30%	**23.49**	**W = 5283.5, p-value = 0.4271**	**no**		**23.70**	**W = 4405, p-value = 0.08086**	**no**
	Entrfin 40%	**23.57**	**W = 4923.5, p-value = 0.8286**	**no**		**23.69**	**W = 4453.5, p-value = 0.11**	**no**
	Entrfin 50%	**23.59**	**W = 4910.5, p-value = 0.7994**	**no**		23.75	W = 4162.5, p-value = 0.01234	yes
	Ucb	23.22	W = 6453, p-value = 6.877e-05	yes		23.09	W = 6672.5, p-value = 7.486e-06	yes
	Ucb 10%	23.27	W = 6098.5, p-value = 0.002753	yes		23.12	W = 6826, p-value = 6.647e-07	yes
	Ucb 20%	23.37	W = 5837.5, p-value = 0.02051	yes		23.14	W = 6654, p-value = 7.223e-06	yes
	Ucb 30%	**23.40**	**W = 5664, p-value = 0.06588**	**no**		23.26	W = 6371, p-value = 0.0001507	yes
	Ucb 40%	**23.51**	**W = 5186.5, p-value = 0.6004**	**no**		23.37	W = 5893.5, p-value = 0.01316	yes
	Ucb 50%	**23.42**	**W = 5623, p-value = 0.08335**	**no**		23.32	W = 6108, p-value = 0.002166	yes
	Xwtr	**23.56**	**W = 5056, p-value = 0.8748**	**no**		**23.60**	**W = 4806, p-value = 0.5791**	**no**
	Xwtr 10%	**23.57**	**W = 4923.5, p-value = 0.8286**	**no**		**23.62**	**W = 4765, p-value = 0.5002**	**no**
	Xwtr 20%	**23.68**	**W = 4474, p-value = 0.1245**	**no**		**23.69**	**W = 4453.5, p-value = 0.11**	**no**
	Xwtr 30%	23.73	W = 4259.5, p-value = 0.02812	yes		**23.60**	**W = 4806, p-value = 0.5791**	**no**
	Xwtr 40%	**23.68**	**W = 4502, p-value = 0.1466**	**no**		**23.63**	**W = 4688.5, p-value = 0.3695**	**no**
	Xwtr 50%	**23.71**	**W = 4356.5, p-value = 0.05817**	**no**		23.77	W = 4065.5, p-value = 0.004877	yes

Results with Host Churn

	Entrfin	**23.52**	W = **W = 5222, p-value = 0.5322**	**no**		**23.58**	**W = 4931, p-value = 0.8452**	**no**
	Ucb	21.31	W = 9708.5, p-value < 2.2e-16	yes		23.03	W = 7038.5, p-value = 4.588e-08	yes
	Xwtr	**23.64**	**W = 4640, p-value = 0.2982**	**no**		**23.7**	**W = 4405, p-value = 0.08086**	**no**

Table 8 3-Trap fitness comparison between error-prone and error-free cases using Wilcoxon test (*Day 1 and 2*) – "not significantly different" means fitness quality comparable to the error-free case.

Fig. 2 shows, for the worst-case scenario, how the population decreases as failures occur in the system. As explained before, two different 24-hours periods randomly selected are shown, denoted by Day 1 and Day 2, for the three employed traces of the experiments. Thus, a total of 6 different experiments, one per trace and day period, were run with the 3-Trap function problem.

Tab. 8 shows a summary of the obtained results for the experiments. From all the traces, the *ucb* has obtained the worst fitness 23.22 and 23.09 (respectively for both periods Day 1 and Day 2). The reason is that this trace in the first day loses a 64% of the population and in the second day it loses more or less the whole population 95.83% (see Fig. 2). Consequently it is very difficult for the algorithm to obtain a solution with a similar quality to the error-free scenario.

The second worst case of all the experiments is the *entrfin* trace for the first period (Day 1). This trace loses more or less half of the population in the first 5 generations (see Fig. 2), making really difficult to obtain a good solution even though the population size is steady the rest of the generations. Thus, the obtained fitness for this period is not comparable to the error-free case.

Finally, the *xwtr* trace in both periods obtains solutions with similar quality to the error-free environment (23.56 and 23.6 respectively for each day). In both periods, the *xwtr* trace does not lose more than a 20% for Day 1 and 12% for the second day. Consequently, we conclude that for the 3-Trap function problem, it is possible to tolerate a gradual loss of up to 20% of the individuals without sacrificing solution quality and more importantly without using any fault-tolerance mechanism. Never-

theless, if the loss of individuals is too high, above the 45%, the solution quality is significantly diminished. Since real-world DGSs experience such large amount of failures, we attempt to address this problem. Our simple idea is to increase the initial population size (a 10%, 20%, 30%, 40% and 50%) and run the same simulations using the same traces. The aim is to compensate the loss of the system by providing more individuals at the first generation.

Tab. 8 shows the obtained results for Day 1 and Day 2 periods of the three traces with the increased population. For the *entrfin* trace, the first period (Day 1) with a loss rate of 45.3%, a 10% extra individuals is enough to obtain solutions of similar quality to the error-free case. In the second period, Day 2, the trace obtains similar solutions to the error-free case and when adding an extra 50% the obtained solution is even better than in the error-free case.

For the *ucb* trace, the first period (Day 1) increasing a 30% the size of the population is sufficient to obtain solutions with similar quality to the error-free case. The second period, Day 2, even though an extra 50% of individuals is added at the first generation it is not enough to cope with the high loss rate of this period: 95.83%.

Finally, the *xwtr* trace for both periods obtains solutions with similar quality to the error-free case and in some cases it improves it. For this trace, the increased population would have not been necessary because the PGA tolerates, without any extra individual, the rate loss of both periods.

It is important to remark that by adding more individuals to the initial population, we are increasing the computation time since more individuals have to be evaluated per generation. Nevertheless, this extra time is similar to the extra time that would be required by standard fault tolerance mechanisms (e.g. failure detection and re-send lost individuals for fitness evaluation). Thus, we conclude that increasing the population size, accordingly to the failure rate, is enough to improve the PGA quality of solutions when the failure rate is known.

Up to now, we have only considered the worst-case scenario: lost resources never become available again. Nevertheless, real-world DG systems does not behave like this assumption, and thus we are going to use the traces with the possibility of re-acquiring the lost resources (see Fig. 1). Next section analyzes the results obtained when re-acquiring lost resources is a possibility.

5.2.1 3-trap:Results with churn

When using the full churn traces of the three DGSs (*entrfin*, *ucb* and *xwtr*) an important question arises: what work is assigned to the new available workers? We have assumed that when workers become available again the master node creates I new random individuals and increases the size of the population accordingly. Thus, the size of the population can be changed dynamically as individuals are added and removed along generations. In this scenario it could happen that new workers nodes appear during the execution of the algorithm increasing the population over its optimum size. Hence the master node is not allowed to create more individuals than the optimum population size leaving several workers idle. In order to avoid idle workers,

it would be interesting to adjust the number of I individuals to evaluate accordingly to the number of available hosts. Nevertheless, we leave such load-balancing study for a future work.

On the other hand, due to the loss of resources, the population can be emptied because all the workers have disappeared. If this situation occurs, the server node proceeds to the next generation by waiting the specified time T (based on the required time per generation in the failure-free environment) for new workers.

Tab. 8 shows the obtained results for the three traces with the host-churn phenomena (*entrfin*, *ucb* and *xwtr*) and the previous corresponding two periods: Day 1 and Day 2. We used the same periods as in the worst-case scenario, but now choosing a random point in the 24-hours period as the starting point for the algorithm. Tab. 9 shows the obtained fitness of the 3-Trap function problem and the host churn of each trace represented by the minimum, median, mean, maximum, and variance of the number of available worker nodes.

Table 9 Obtained fitness for 3-Trap function with host churn

Trace	Hosts					Fitness
	Min.	Median	Mean	Max.	Var. (s^2)	3-Trap
Error free	-	-	-	-	-	23.56
entrfin (*Day 1*)	92	161.5	156.8	177	305.59	23.52
entrfin (*Day 2*)	180	181	180.9	182	0.6	23.58
ucb (*Day 1*)	0	2	1.9	9	3.12	21.31
ucb (*Day 2*)	0	4	3.7	7	2.7	23.03
xwtr (*Day 1*)	28	29	28.87	29	0.11	23.64
xwtr (*Day 2*)	86	86	86	86	0	23.70

If the variance of the number of available hosts is zero, then the execution is obviously the same as in the error-free case because the number of hosts is steady along generations. In this case, the obtained fitness should be similar to the error-free case. This situation is present within the second period (Day 2) of the *xwtr* trace (variance equal to zero) and thus the obtained fitness is similar to the error-free case (see Tab. 8). The other period of the *xwtr* trace has also a very small variance, 0.11, resulting in a similar solution quality in comparison with the error-free scenario. The *entrfin* trace for both periods obtains solutions of similar quality to the error-free environment, even though the large variance observed in the Day 1 period ($s^2 = 305.59$). Despite the large variance, the number of available hosts is high in comparison with the other traces, so the PGA tolerates better the failures and provides solutions of similar quality to the error-free case. Finally, the *ucb* trace obtains the worst results due to in both periods the minimum number of available hosts is zero. Consequently, the population is emptied, making very difficult to obtain solutions of similar quality to the error-free environment.

5.3 Summary of Results

Based on two standard applications, EP5 and 3-trap, we have shown that PGP and PGA applications based on the master-worker model running on DGSs that exhibit host failures can achieve solution qualities close to those in the failure-free case, without resorting to any fault tolerance technique. Two scenarios were tested: (i) the scenario in which lost hosts never come back but in which one starts with a large number of hosts; and (ii) the scenario in which hosts can re-appear during application execution. For scenario (i) we found that there is an approximately linear degradation of solution quality as host losses increase. This degradation can be alleviated by increasing initial population size. For scenario (ii) the degradation varies during application execution as the number of workers fluctuates. The main observation is that in both cases we have *graceful degradation*.

6 Conclusions

In this chapter we have analyzed the behavior of Parallel Genetic Programming (PGP) and Parallel Genetic Algorithms executing in distributed platforms with high failure rates, with the goal of characterizing the inherent fault tolerance capabilities of the PGP paradigm. We have used two well-known problems and, to the best of our knowledge, for the first time in this context we have used host availability traces collected on real-world Desktop Grid (DG) platforms.

Our main conclusion is that PGP and PGA inherently provides *graceful degradation* without the need for fault tolerance techniques.

We have presented a simple method for tolerating high host losses, which consists of increasing the initial population size.

To the best of our knowledge, this is the first time that PGP and PGAs are characterized from a fault-tolerance perspective. We contend that our conclusions can be extended to Parallel Evolutionary Algorithms (PEAs) via similar experimental validation.

References

1. David H. Ackley. *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.
2. Adnan Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 31, Washington, DC, USA, 1999. IEEE Computer Society.
3. E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 6(5):443–462, Oct 2002.
4. D.P. Anderson. Boinc: a system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10, 2004.

5. D.P. Anderson and G. Fedak. The Computational and Storage Potential of Volunteer Computing. *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, 2006.
6. David Andre and John R. Koza. Parallel genetic programming: a scalable implementation using the transputer network architecture. pages 317–337, 1996.
7. M.G. Arenas, P. Collet, A.E. Eiben, M. Jelasity, J.J. Merelo, B. Paechter, M. Preuß, and M. Schoenauer. A framework for distributed evolutionary algorithms. *Lecture Notes in Computer Science*, pages 665–675, 2003.
8. S. Cahon, N. Melab, and E.G. Talbi. Building with paradisEO reusable parallel and distributed evolutionary algorithms. *Parallel Computing*, 30(5-6):677–697, 2004.
9. E. Cantu-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.
10. Michael J. Crawley. *Statistics, An Introduction using R*. Wiley, 2007.
11. Francisco Chávez de la O, Jose Luis Guisado, Daniel Lombraña, and Francisco Fernández. Una herramienta de programación genética paralela que aprovecha recursos públicos de computación. In *V Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados*, volume 1, pages 167–173, Tenerife, Spain, February 2007.
12. Kalyanmoy Deb and David E. Goldberg. Analyzing deception in trap functions. In L. Darrell Whitley, editor, *FOGA*, pages 93–108. Morgan Kaufmann, 1992.
13. T. Desell, B. Szymanski, and C. Varela. An asynchronous hybrid genetic-simplex search for modeling the Milky Way galaxy using volunteer computing. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 921–928. ACM, 2008.
14. Douglas Zongker Dr. Bill Punch. Lil-gp. `http://garage.cse.msu.edu/software/lil-gp/index.html`.
15. E.N.M. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.
16. L. Vanneschi F. Fernández, M. Tomassini. Saving computational effort in genetic programming by means of plagues. *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, 2003.
17. Francisco Fernandez, Giandomenico Spezzano, Marco Tomassini, and Leonardo Vanneschi. Parallel genetic programming. In Enrique Alba, editor, *Parallel Metaheuristics*, Parallel and Distributed Computing, chapter 6, pages 127–153. Wiley-Interscience, Hoboken, New Jersey, USA, 2005.
18. Message Passing Interface Forum. Mpi: a message-passing interface standard. *International Journal Supercomput. Applic.*, 8(3–4):165–414, 1994.
19. Christian Gagné, Marc Parizeau, and Marc Dubreuil. Distributed beagle: An environment for parallel and distributed evolutionary computations. In *Proc. of the 17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS) 2003*, pages 201–208, May 11-14 2003.
20. Felix C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, 1999.
21. Sukumar Ghosh. *Distributed systems: an algorithmic approach*. Chapman & Hall/CRC, 2006.
22. Daniel Lombraña González, Francisco Fernández de Vega, and Henri Casanova. Characterizing fault tolerance in genetic programming. In *Workshop on Bio-Inspired Algorithms for Distributed Systems*, pages 1–10, Barcelona, Spain, June 2009.
23. Daniel Lombraña González, Francisco Fernández de Vega, and Henri Casanova. Characterizing fault tolerance in genetic programming. *Future Generation Computer Systems*, 26(6):847–856, 2010.
24. Daniel Lombraña González, Francisco Fernández de Vega, Leonardo Trujillo, Gustavo Olague, Lourdes Araujo, Pedro Castillo, Juan Julián Merelo, and Ken Sharman. Increasing gp computing power for free via desktop grid computing and virtualization. In *Proceedings of the 17th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 419–423, Weimar, Germany, February 2009.

25. Daniel Lombraña González, Juan Luís Jiménez Laredo, Francisco Fernández de Vega, and Juan Julián Merelo Guervós. Characterizing fault-tolerance of genetic algorithms in desktop grid systems. In *10th European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 131–142, Istanbul, Turkey, April 2010.
26. R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, 1997.
27. Ignacio Hidalgo, Francisco Fernández, Juan Lanchares, and Daniel Lombraña. Is the island model fault tolerant? In *Genetic and Evolutionary Computation Conference*, volume 2, page 1519, London, England, July 2007.
28. Márk Jelasity, Mike Preuß, Maarten van Steen, and Ben Paechter. Maintaining connectivity in a scalable and robust distributed environment. In Henri E. Bal, Klaus-Peter Löhr, and Alexander Reinefeld, editors, *Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 389–394, Berlin, Germany, 2002. IEEE Computer Society. 2nd GP2PC workshop.
29. B. Knispel, B. Allen, JM Cordes, JS Deneva, D. Anderson, C. Aulbert, NDR Bhat, O. Bock, S. Bogdanov, A. Brazier, et al. Pulsar Discovery by Global Volunteer Computing. *Science*, 329(5997):1305, 2010.
30. D. Kondo, G. Fedak, F. Cappello, A.A. Chien, and H. Casanova. Characterizing resource availability in enterprise desktop grids. volume 23, pages 888–903. Elsevier, 2007.
31. D. Kondo, M. Taufer, C. Brooks, H. Casanova, and A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'04)*. Citeseer, 2004.
32. P. Kouchakpour, A. Zaknich, and T. Br
"aunl. Dynamic population variation in genetic programming. *Information Sciences*, 179(8):1078–1091, 2009.
33. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
34. Juan L. J. Laredo, Pedro A. Castillo, Antonio M. Mora, Juan J. Merelo, and Carlos Fernandes. Resilience to churn of a peer-to-peer evolutionary algorithm. *Int. J. High Performance Systems Architecture*, 1(4):260–268, 2008.
35. Juan L. J. Laredo, Agoston E. Eiben, Maarten van Steen, and Juan J. Merelo. On the run-time dynamics of a peer-to-peer evolutionary algorithm. In Gunter Rudolph, Thomas Jansen, Simon Lucas, Carlo Poloni, and Nicola Beume, editors, *Parallel Problem Solving from Nature - PPSN X*, volume 5199 of *LNCS*, pages 236–245, Dortmund, 13-17 September 2008. Springer.
36. Daniel Lombraña and Francisco Fernández. Analyzing fault tolerance on parallel genetic programming by means of dynamic-size populations. In *Congress on Evolutionary Computation*, volume 1, pages 4392 – 4398, Singapore, September 2007.
37. Daniel Lombraña, Francisco Fernández, L. Trujillo, G. Olague, M. Cárdenas, L. Araujo, P. Castillo, K. Sharman, and A. Silva. Interpreted applications within boinc infrastructure. In *Ibergrid 2008*, pages 261–272, Porto, Portugal, May 2008.
38. Daniel Lombraña, Francisco Fernández, Leonardo Trujillo, Gustavo Olague, and Ben Segal. Customizable execution environments with virtual desktop grid computing. *Parallel and Distributed Computing and Systems, PDCS*, 2007.
39. Sean Luke, Gabriel Catalin Balan, and Liviu Panait. Population implosion in genetic programming. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1729–1739, Chicago, 12-16 July 2003. Springer-Verlag.
40. Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Elena Popovici, Joseph Harrison, Jeff Bassett, Robert Hubley, and Alexander Chircop. Ecj a java-based evolutionary computation research system, 2007. http://cs.gmu.edu/ eclab/projects/ecj/.
41. N. Melab, S. Cahon, and E.-G. Talbi. Grid computing for parallel bioinspired algorithms. *J. Parallel Distrib. Comput.*, 66(8):1052–1061, 2006.

42. J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *FGCS. Future generations computer systems*, 12(1):67–85, 1996.
43. D.A. Reed, C. Lu, and C.L. Mendes. Reliability challenges in large systems. *Future Generation Computer Systems*, 22(3):293–302, 2006.
44. B. Schroeder and G. A. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *Proc. of the International Conference on Dependable Systems*, pages 249–258, 2006.
45. M.L. Shooman. *Reliability of computer systems and networks: fault tolerance, analysis and design*. Wiley-Interscience, 2002.
46. Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM on Internet measurement (IMC 2006)*, pages 189–202, New York, NY, USA, 2006. ACM Press.
47. V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.
48. Ann T. Tai and Kam S. Tso. A performability-oriented software rejuvenation framework for distributed applications. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 570–579, Washington, DC, USA, 2005. IEEE Computer Society.
49. Dirk Thierens. Scalability problems of simple genetic algorithms. *Evolutionary Computation*, 7(4):331–352, 1999.
50. M. Tomassini. Parallel and distributed evolutionary algorithms: A review. In P. Neittaanmäki K. Miettinen, M.Mäkelä and J. Periaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*, pages 113,133. J. Wiley and Sons, Chichester, 1999.
51. M. Tomassini. *Spatially Structured Evolutionary Algorithms*. Springer, 2005.
52. M. Tomassini, L. Vanneschi, J. Cuendet, and F. Fernandez. A new technique for dynamic size populations in genetic programming. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, 2004.
53. L. Trujillo and G. Olague. Automated Design of Image Operators that Detect Interest Points. volume 16, pages 483–507. MIT Press, 2008.
54. E. Vargas. High availability fundamentals. *Sun Blueprints series*, 2000.