Characterizing Fault Tolerance in Genetic Programming

Daniel Lombraña González[a], Francisco Fernández de Vega[a], Henri Casanova[b]

[a] *University of Extremadura, Santa Teresa Jornet 38, 06800 Mérida, Badajoz, Spain.*
[b] *University of Hawai'i at Manoa, PST #317, 1680 East-West Road, Honolulu, HI 96822, U.S.A.*

Abstract

Evolutionary Algorithms, including Genetic Programming (GP), are frequently employed to solve difficult real-life problems, which can require up to days or months of computation. An approach for reducing the time-to-solution is to use parallel computing on distributed platforms. Large such platforms are prone to failures, which can even be commonplace events rather than rare occurrences. Thus, fault tolerance and recovery techniques are typically necessary. The aim of this article is to show the inherent ability of Parallel GP to tolerate failures in distributed platforms without using any fault-tolerant technique. This ability is quantified via simulation experiments performed using failure traces from real-world distributed platforms, namely, desktop grids, for two well-known problems.

Key words: Fault-tolerance, Parallel Genetic Programming, Desktop Grids.

1. Introduction

Evolutionary Algorithms (EAs) are employed to solve real-world problems. However, when difficult problems are faced, large and often prohibitive times-to-solution ensue. A common approach is then to use parallel versions of the algorithm. Successful Parallel Evolutionary Algorithms (PEAs) have been proposed [1, 2, 3]. Additionally, development frameworks for PEAs have been implemented (e.g., Calypso [4]), some specifically for Parallel Genetic Programming (PGP) [5].

PEAs can be run on potentially large-scale parallel computing platforms. Two types of parallel platforms have achieved very high scales: *clusters* and *desktop grids*. In the last decade large clusters have become mainstream, and clusters account today for more than 80% of the Top500 list, which ranks the 500 fastest supercomputers based on the LINPACK benchmark [6]. The term "desktop grid" (DG) refers to distributed networks of heterogeneous individual systems that contribute computing resources when idle. A well-known example of DG is the Berkeley Open Infrastructure for Network Computing (BOINC) [7], which hosts the famous SETI@home [8] project. At the time this article is being written, BOINC enlists around 500,000 volunteer computers. Smaller, but still impressive, DG infrastructures, are deployed within enterprises or data centers [9]. Such deployments comprise more powerful, more available, and less heterogeneous computers than Internet-wide DGs. The main advantage of DGs is that they provide large-scale parallel computing capabilities at a very low cost for specific types of applications.

The aforementioned large-scale parallel computing platforms hold promises for running large PEAs. However, with large scale there is a higher risk that processors experience failures during the execution of an application (e.g., a crash). In this paper the

Email addresses: daniellg@unex.es (Daniel Lombraña González), fcofdez@unex.es (Francisco Fernández de Vega), heric@hawaii.edu (Henri Casanova)

terms "failure" and "fault" are used without making the subtle distinction between them, as it is not necessary for our purpose. Failures occur frequently in large-scale clusters [10]. In DGs, failures are the common case: a participating computer can be reclaimed by its owner at any time (e.g., when the owner launches an application, when the keyboard/-mouse is used). In this case, the DG application is abruptly suspended or terminated, which it is seen as a failure.

In order to circumvent and/or alleviate failures, many researchers have developed different techniques for an application to not be terminated when one or more of the participating processors experience a failure. This ability is known as *fault tolerance*, and ensures that the application behaves in a well-defined manner (e.g., with graceful degradation of performance) when a failure occurs [11]. Various fault tolerance techniques have been developed [12]. These techniques can be employed with parallel applications, and support many types of computational and communication failures [13]. In general, the employment of fault tolerance mechanisms requires the modification of the application, and sometimes the parallel algorithms themselves. The developer thus could face a sharp increase in software complexity. For this reason, generic fault tolerance solutions have been developed as libraries or software environments [14, 15, 16].

To the best of our knowledge, there has been little investigation of the behaviors of PEAs in general, and of PGP in particular, in the presence of failures. Nevertheless, there are different tools available that can be used to parallelize and run any EA, and thus GP, in volunteer computing environments [17], where failures are common.

In previous work [18, 19], we presented preliminary results about fault tolerance in PGP under several simplified assumptions. We showed that PGP applications exhibit inherent fault-tolerant behaviors. Therefore, it seems feasible to run them on large-scale computing infrastructures, which suffer from failures, but without the burden of implementing/using any kind of fault tolerance techniques, and without sacrificing overall application efficiency significantly. We then extended those preliminary results in two ways [20]. First, two different PGP problems were used for running simulations using host availability data collected from real-world DG deployments (instead of using simplistic, and ultimately unrealistic, processor availability models). Second, the simulations used availability data from different platforms, making it possible to study the impact of different host availability profiles on application execution. To the best of our knowledge, this was the first time that an attempt for characterizing PGP had been performed from the fault-tolerance point of view. The results showed that in some specific contexts, PGP can tolerate various failure rates.

All previous results were obtained using a stringent assumption: once unavailable, a resource never becomes available again. This is, however, not the case in real-world DGs. In this paper we extend the work in [20] and run simulations in which resources can become available again. This should further improve the *graceful degradation* feature of PGP as the number of resources fluctuates throughout application execution instead of continuously decreasing.

This paper is organized as follows. Section 2 reviews related works beyond the ones described earlier. Section 3 provides an overview of the different types of failures that may arise as well as the relevant fault tolerance techniques. Section 4 describes our experimental methodology, and results are discussed in Section 5. Section 6 concludes the paper with a summary of our results and future directions.

2. Background and Related Work

When using EAs, and especially GP, 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 [21], 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 [1, 2, 3]. Such PEAs have been used for decades, for instance, on the Transputer platform [22], or, more recently, via software frameworks such as Beagle [23], grid based

tools like Paradiseo [24], or BOINC-based EA frameworks for execution on DGs [17].

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 mandating that the application 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 [25, 26, 27]. Failures are, in fact, the common case in DGs. 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 [28], redundancy [29], long-term-memory [30], specific solutions to message-passing [31] or rejuvenation frameworks [32]. 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 and restart from scratch), the more involved 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 like in DGs. For instance ECJ [33], ParadisEO [34], DREAM [35] or Distributed Beagle [23]. These frameworks have distinct features (programming language, parallelism models, etc.) that may be considered in combination with DGs, and provide different techniques to cope with failures:

- ECJ [33] 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 [34] is a C++ framework for running a master-worker model using MPI [36], PVM [37], or POSIX threads. Initially, ParadisEO did not provide any fault-tolerance. The developers later implemented a new version on top of the Condor-PVM resource manager [38] in order to provide a checkpointing feature [28]. This framework, however, is not the best choice for DGs 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 [35] is a Java peer-to-peer (P2P) framework for PEAs that provides a fault-tolerance mechanism called *long-term-memory* [30]. 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 DGs.

- Distributed BEAGLE [23] 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.

An interesting and relatively recent development is the use of *dynamic populations* [39, 40, 41, 42], i.e., population size is reduced and/or increased in order to minimize fitness stagnation. In 2003, Fernández *et al.* [40] introduced a new operator named *plague*, which reduces the population size at a linear rate. The same year, Luke *et al.* [41] introduced a similar idea: reducing population size according to different *layouts*. In 2004, Tomassini *et al.* [39] presented a study on dynamic populations by removing/adding individuals to avoid fitness stagnation. More recently, Kouchakpour *et al.* [43] introduced a population variation scheme to add/remove individuals from the population, and in [42] improved the work presented by Tomassini *et al.* by proposing a

new pivot function and four new measures for characterizing the stagnation phase.

Our key observation is that the loss of individuals due to worker failures inherently leads to dynamic populations, albeit in a random and thus less controlled fashion. Based on this observation, the intriguing question is whether one could simply allow individuals to be lost without taking any particular fault-tolerant measures, thereby achieving dynamic population and fault tolerance "for free". We first explored this idea in [44, 45], and, as explained in Section 1, attempted a preliminary quantification of the fault tolerance characteristics of PGP in [18, 19]. These results were extended in our more recent work [20], using more realistic failure models based on real-world observations of DG platforms. In this paper we further extend the realism of the results in [20] by allowing workers to become available again after they have experienced a failure, as it is the case in real-world DGs. Our objective is to quantify the *graceful degradation* feature of PGP as the number of resources fluctuates throughout application execution.

3. Fault Tolerance

3.1. Failure Models

Fault tolerance can be defined as the ability of a system to behave in a well-defined manner once a failure occurs. In this paper 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. According to Ghosh [13], 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 [13]: (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 [13].

To discuss fault-tolerance in the context of PGP, we first need to specify the way in which the GP application is parallelized. Parallelism has been traditionally applied to GP at two possible levels: the individual level or the population level [2, 3, 46, 47]. 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 [2, 46] and taking into account the specific parallel features of DGs [48, 49], we focus on parallelization at the individual level. Indeed, DGs 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 [3, 46, 47].

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 [29]) 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 PGP 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 DGs, namely ECJ and Distributed Beagle, the common technique to cope with failures is the first one: resend 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 DGs due to high levels of host heterogeneity [7, 13, 50]. 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 [39, 40, 41, 42]. 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

We rely on simulation experiments. Simulation allows us to perform a statistically significant number of experiments in a wide range of realistic scenarios. Most importantly, our experiments are repeatable, via "replaying" host availability trace data collected from real-world DG platforms [49], so that fair comparison between simulated application executions is possible.

4.1. Application and Failure Model

We perform experiments for two well-known GP problems, even parity 5 (EP5) and 11-multiplexer (11M) [51]. The EP5 problem tries to build a program capable of calculating the parity of a set of 5 bits, while the 11M problem consists in learning the boolean 11-multiplexer function. All the GP parameters, including population sizes, are Koza-I/II standard [51]. See Table 1 for all details. For both problems, fitness is measured as the error in the obtained solution, with zero meaning that a perfect solution has been found. 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).

Table 1: Parameters of selected problems.

	EP5	11M
Population	4000	4000
Generations	51	51
Elitism	Yes	Yes
Crossover Probability	0.90	0.90
Reproduction Probability	0.10	0.10
Selection: Tournament	7	7
Max Depth in Cross	17	17
Max Depth in Mutation	17	17
ADFs	Yes	-

Two kind of experiments are performed: (i) for the failure-free case, i.e., assuming that no worker failures occur; and (ii) replaying and simulating failures traces from real-world DGs. In the failure-free case the amount of available computing power is steady throughput execution, while in a real-world environment it varies between 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 [49]: *ucb*, *entrfin*, and *xwtr*. These traces are time-stamped observations of the host availability in three DGs. 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 [49] for full details on the measurement methods and the traces, and Tab. 2 for a summary of its main features.

Table 2: 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.

4.2. Determining T

Recall that in the approaches described in Section 3.2, the master waits a time T before declaring that individuals have been lost. T is a common parameter of DGs middlewares like BOINC [7]. The T parameter is used both to distinguish between slow and fast hosts and to detect host failures. Thus, it is recommended to estimate how much time would be needed on average to run an application on a typical host. This estimation is commonly carried out by GP researchers, when they are faced with complex and open problems. Furthermore, as several GP parameters (population size, generations, probability of crossover, probability of mutation, selection method, etc.) must be tuned, a common practice is to make some preliminary runs of the algorithm trying different values, in order to find out good values for the parameter set [51, 52]. As a result, the GP researcher will have an estimation of individual evaluation times, through which T can be determined. Additionally, DG systems are often designed to send jobs to hosts that are likely to finish them before a specified deadline, in this case T, like in BOINC [50].

4.3. No Churn vs. Churn

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

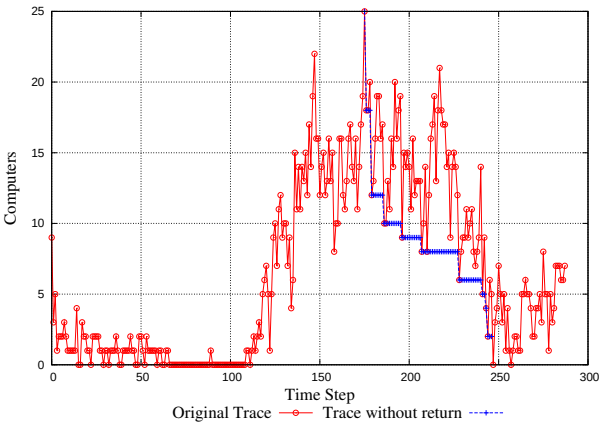


Figure 1: Host availability for 1 day of the *ucb* trace.

"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."

4.4. 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 (see Table 1). 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 (see Table 1). The master then proceeds to the next generation by waiting the specified time T for newly available workers.

In the churn scenario, or in the non-churn scenario using a non-fault-tolerant approach, population size can change dynamically as individuals are added or lost. Recall that dynamic population variation has been applied to sequential versions of EAs [39, 41, 42]. In those studies (except [41] where Luke *et al.* only reduce the population size like in our non-churn scenario), a *pivot function* is applied to control the stagnation of fitness and act accordingly by removing/adding individuals from/to the population. By contrast, in our case population size changes without any consideration for fitness quality.

4.5. 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 6 and 7 present the Wilcoxon analysis of the data. The following sections discuss these results in detail.

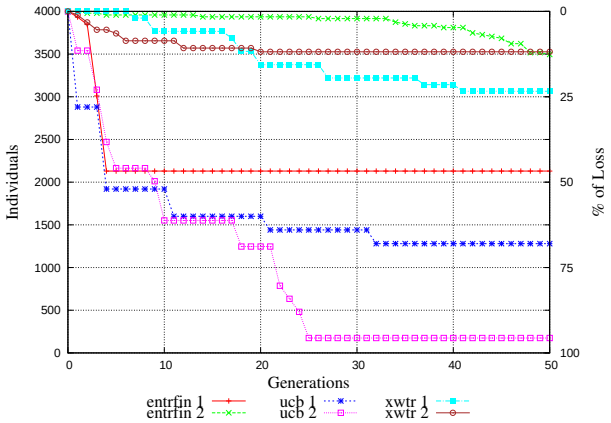


Figure 2: Population size vs. generation.

5. Experimental Results

5.1. Results without Churn

In this section we consider the scenario in which hosts never become available again (no churn): the number of individuals per generation is non-increasing. Figure 2 shows the evolution of the number of individuals in each generation for the EP5 and 11M problems 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 and 11M problems (two last columns) 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 and 11M

		EP5	11M
Trace	Loss(%)	Fitness	Fitness
Error free	0.00	2.56	14.4
entrfin (Day 1)	48.02	3.58	30.36
entrfin (*Day 2*)	13.04	2.44	18.84
ucb (Day 1)	68.00	3.98	49.04
ucb (*Day 2*)	96.15	5.13	67.07
xwtr (Day 1)	23.52	2.78	20.92
xwtr (*Day 2*)	12.08	2.61	16.16

The broad conclusion from this table is that the fitness values achieved by our non-fault-tolerant approach is not necessarily much worse than that in the failure-free case. In some cases, however, it leads to significantly worse fitness due to severe losses (e.g., for 11M and *Day 2* of *ucb*).

An interesting question is the additional time that would be required in case the fault-tolerant approach described in Section 3 is used. While this approach leads to the same fitness as the failure-free case, it leads to longer execution times due to individuals being resent upon a host failure. Therefore, the execution time depends on host failures and on host availability. By contrast, in our non-fault-tolerant approach, the server waits T seconds for each generation, independently of failures and host availability. Therefore, in all cases, the execution time is equal to the number of generations multiplied by T.

Table 4 shows simulated execution times for both approaches, as multiples of T, for all our traces. Our non-fault-tolerant approach leads to an execution time equal to $51 \times T$. This fault-tolerant approach fails for three traces, for which the execution time is shown as infinite in Table 4. In these cases, no more hosts are available (recall that we are in the non-churn scenario) and individuals still need to be evaluated.

Given that the non-fault-tolerant approach achieves acceptable fitness values in many cases, these results provide a strong indication that using a fault-tolerant approach is not worthwhile given the dramatic increase in execution time. Nevertheless, the cases in which our non-fault-tolerant approach achieves fitness values significantly lower than those in the failure-free case are a concern. In the next section we discuss the results in Table 3 in more depth for both EP5 and 11M, and propose a simple technique to improve the fitness achieved by our approach.

Table 4: Execution time of the fault-tolerant approach in terms of T for the non-churn scenario. The non-fault-tolerant approach has an execution time equal to $51 \times T$.

Period	*entrfin*	*ucb*	*xwtr*
Day 1	$101 \times T$	∞	$95 \times T$
Day 2	$100 \times T$	∞	∞

5.1.1. Even Parity 5 (EP5)

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 6 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 DGs 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 [40]. We have fixed a maximum computing effort which corresponds to 51 generations and the population size introduced by Koza [51], 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 5: 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 5 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 (see Table 4).

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. 11 Bits Multiplexer (11M)

For the 11M problem the failure-free fitness is 14.40, while it ranges from 16.16 to 67.07 for the failure-prone cases (see Table 3). As for the EP5 problem, and for the same reason, the worst fitness is obtained by the *ucb* trace in both periods. The traces with host losses under 25% obtain solutions not significantly different from the failure-free case. Table 7 shows a summary of the results and of their statistical significance.

Figure 5.1.1(b) is similar to Figure 5.1.1(a), but shows fitness versus computational effort for the 11M problem. The conclusion is the same as for the EP5 problem: increasing the population size beyond a judiciously chosen population size is not a good option in the failure-free case. Table 8 shows fitness values for the *entrfin*, *ucb* and *xwtr* traces with increased population sizes. The conclusion is the same as for the EP5 problem, namely that increasing population size is an effective way to cope with lost individuals.

Table 6: 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.

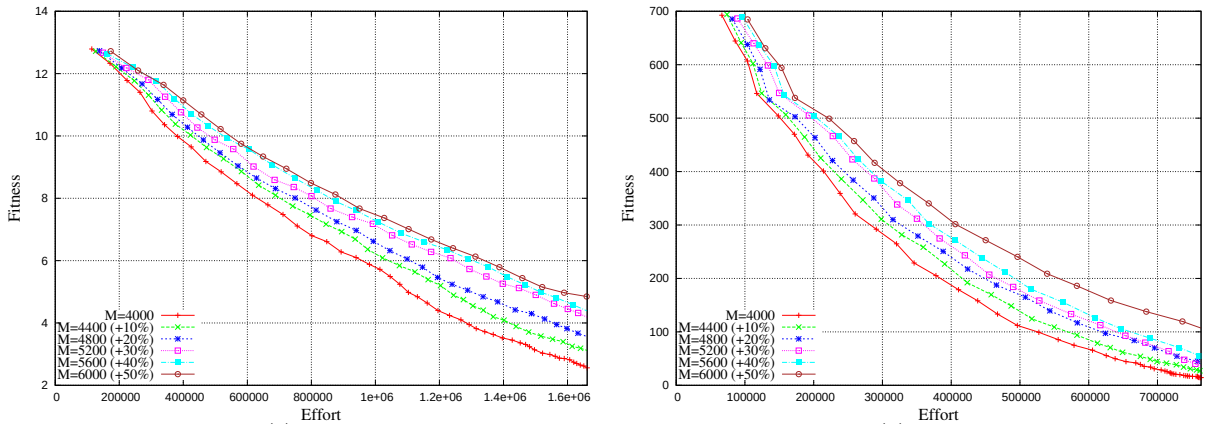
Error Free fitness = 2.56

	Trace	Fitness	Wilcoxon Test	Significantly different?		Fitness	Wilcoxon Test	Significantly different?
Day 1	*entrfin*	3.58	W = 6726, p-value = 1.843e-05	yes	Day 2	**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							
	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**

Table 7: 11M 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 = 14.40

	Trace	Fitness	Wilcoxon Test	Significantly different?		Fitness	Wilcoxon Test	Significantly different?
Day 1	*entrfin*	30.36	W = 5858, p-value = 0.003781	yes	Day 2	**18.84**	**W = 5282.5, p-value = 0.2797**	**no**
	entrfin 10%	28.23	W = 5558.5, p-value = 0.04374	yes		**14.36**	**W = 5077.5, p-value = 0.755**	**no**
	entrfin 20%	26.20	W = 5592, p-value = 0.03438	yes		**10.40**	**W = 4881.5, p-value = 0.6093**	**no**
	entrfin 30%	**22.98**	**W = 5470, p-value = 0.08639**	**no**		**10.40**	**W = 4923.5, p-value = 0.7464**	**no**
	entrfin 40%	**8.88**	**W = 4866, p-value = 0.5631**	**no**		1.92	W = 4485, p-value = 0.00752	yes
	entrfin 50%	**10.08**	**W = 4761.5, p-value = 0.2740**	**no**		1.92	W = 4447, p-value = 0.003106	yes
	ucb	49.04	W = 6413.5, p-value = 9.58e-06	yes		67.07	W = 7182, p-value = 2.870e-10	yes
	ucb 10%	47.44	W = 6288, p-value = 4.426e-05	yes		73.76	W = 7516.5, p-value = 1.436e-12	yes
	ucb 20%	35.24	W = 5934, p-value = 0.001779	yes		49.44	W = 6805.5, p-value = 7.762e-08	yes
	ucb 30%	39.20	W = 6306.5, p-value = 3.755e-05	yes		54.98	W = 7009, p-value = 4.39e-09	yes
	ucb 40%	**23.09**	**W = 5483, p-value = 0.07804**	**no**		46.80	W = 6478.5, p-value = 4.842e-06	yes
	ucb 50%	25.20	W = 5696.5, p-value = 0.01573	yes		50.78	W = 6758, p-value = 1.449e-07	yes
	xwtr	**20.92**	**W = 5419, p-value = 0.1222**	**no**		**16.16**	**W = 5098, p-value = 0.6926**	**no**
	xwtr 10%	**15.36**	**W = 5120.5, p-value = 0.6318**	**no**		**10.56**	**W = 4923.5, p-value = 0.7464**	**no**
	xwtr 20%	**8.64**	**W = 4752.5, p-value = 0.2565**	**no**		**8.62**	**W = 4788.5, p-value = 0.3423**	**no**
	xwtr 30%	**6.8**	**W = 4730.5, p-value = 0.2165**	**no**		**5.76**	**W = 4645, p-value = 0.08845**	**no**
	xwtr 40%	**5.12**	**W = 4676, p-value = 0.1288**	**no**		1.2	W = 4438.5, p-value = 0.002680	yes
	xwtr 50%	**10.32**	**W = 4844.5, p-value = 0.4939**	**no**		1.6	W = 4483.5, p-value = 0.007348	yes
	Results with Host Churn							
	entrfin	**21.86**	**W = 5473, p-value = 0.08428**	**no**		**13.2**	**W = 5030.5, p-value = 0.9017**	**no**
	ucb	341.74	W = 10000, p-value < 2.2e-16	yes		85.16	W = 7985.5, p-value = 4.441e-16	yes
	xwtr	**11.98**	**W = 4970, p-value = 0.9018**	**no**		**11.36**	**W = 4898, p-value = 0.66**	**no**



(a) EP5 (b) 11M

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

Here again, 50% extra individuals for the *ucb* trace is not enough to fully compensate for its dramatic host failure rate. In some cases the achieve fitness is better than that in the failure-free case (e.g., *Day 2* of the *xwtr* trace with 40% or 50% extra individuals).

Table 8: 11M fitness with increased population

Error Free fitness = 14.40						
Traces	*entrfin*	*ucb*	*xwtr*	*entrfin*	*ucb*	*xwtr*
+0%	30.36	49.04	20.92	18.84	67.07	16.16
+10%	28.23	47.44	15.36	14.36	73.76	10.56
+20%	26.20	35.24	8.64	10.40	49.44	8.62
+30%	22.98	39.20	6.8	10.40	54.98	5.76
+40%	8.88	23.09	5.12	1.92	46.80	1.2
+50%	10.08	25.20	10.32	1.92	50.78	1.6

In summary, for both the EP5 and the 11M problem, we have demonstrated that our non-fault-tolerant approach is preferable to a fault-tolerant approach. Further, unless host failure rates are extremely high, it can achieved good fitness thanks to a simple initial population increase. These results were obtained in the non-churn scenario, and in the next section we turn our attention to the, arguably, more realistic scenario in which there is churn.

5.2. 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 4.4 that the population size is capped at 4,000 individuals (according to Table 1) and that each worker is assigned I individuals. Such individuals are randomly generated by the master when assigned to a newly available worker.

Table 9: Execution time of the fault-tolerant approach in terms of T for the churn scenario. The non-fault-tolerant approach has an execution time equal to $51 \times T$.

Period	*entrfin*	*ucb*	*xwtr*
Day 1	$101 \times T$	$165 \times T$	$53 \times T$
Day 2	$67 \times T$	$154 \times T$	$51 \times T$

Let us first compare the execution time of our non-fault-tolerant approach to that of the fault-tolerant one presented in Section 3. Table 9 shows execution times as multiples of T. Like in the non-churn case, and for the same reasons, the fault-tolerant approach leads to larger execution times than the non-fault-tolerant one. Unlike in the non-churn case, however, the fault-tolerant approach always succeeds since hosts come back after a failure. For the *entrfin* and *ucb* traces, the fault-tolerant approach takes significantly (up to a factor 3) longer to execute than our

non-fault-tolerant one. For the *xwtr* trace, the difference is insignificant because the host failure rate is low. Recall that the estimation of the execution time for the fault-tolerant approach is optimistic as we have assumed that an individual experiences at most one host failure. In practice, this is clearly not true and the execution times would likely be larger. In the next sections we discuss fitness results for the EP5 and the 11M problems.

5.2.1. Even Parity 5 (EP5)

Table 10 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*, 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 6 for statistical significance).

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 10). 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 10). 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.

5.2.2. 11 Bits Multiplexer (11M)

Results for 11M are shown in Table 10 and Table 7 and are essentially the same as those for EP5 for the same reasons. The obtained fitness for the *entrfin* and *xwtr* traces is similar to that in the failure-free case, while the *ucb* trace has the worst fitness due to low host availability. In the case of the *entrfin* trace, *Day 1* period, the obtained fitness is better than in the non-churn case. This suggests that reusing hosts that became available again is effective when compared to the case in which one starts with a large number of hosts but loses hosts forever once they become unavailable, even if the trace exhibits a large variance (see *entrfin*, *Day 1*, in Table 10). The *entrfin*, *Day 2*, trace and *xwtr*, *Day 1* and *Day 2*, traces have the lowest variance and the number of available hosts is always greater than zero along the generations. Therefore, the obtained fitness for these cases is comparable to that in the failure-free case (see Table 7 for statistical significance).

5.3. Summary of Results

Based on two standard applications, EP5 and 11M, we have shown that PGP applications based on the master-worker model running on DGs 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*.

This result is coherent with previous theoretical studies in GP: the replication and diffusion of components of the solution – schemas – in the GP population [53, 54, 55], as well as the convergence towards

Table 10: Obtained fitness for EP5 and 11M with host churn

Trace	Hosts					Fitness	
	Min.	Median	Mean	Max.	Var. (s^2)	EP5	11M
Error free	-	-	-	-	-	2.56	14.4
entrfin (Day 1)	92	160	157.50	177	179.33	2.86	21.86
entrfin (Day 2)	180	181	181.30	183	0.75	2.75	13.20
ucb (Day 1)	0	1	1.51	9	2.21	8.87	341.74
ucb (Day 2)	0	2	2.57	7	4.29	5.89	85.16
xwtr (Day 1)	28	29	28.92	29	0.07	2.56	11.98
xwtr (Day 2)	86	86	86	86	0	2.52	11.36

a global solution. Thus, the inherent fault tolerance features of distributed GP systems may have its origins in how schemas are propagated from one generation to the next one as described by Langdon *et al.* in [53] and Poli *et al.* in [54, 55]: even when some individuals are lost, the population includes duplicated information that may be used along generations.

Finally, our results show the similarity between the inherent volatility of hosts in DGs and *dynamic population variation* techniques. The main difference is that in DGs it is not necessary to use a *pivot function* [39, 42] to achieve dynamic populations. In fact, combining such dynamic variations techniques with the host volatility of DGs could be interesting.

6. Conclusions

In this paper we have analyzed the behavior of Parallel Genetic Programming (PGP) applications 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 GP 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 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. Additionally, results have shown that DGs are the kind of platforms where dynamic population variation techniques should be employed; and that GP schema duplication and propagation in the population –studied by the GP schema theory– may be the origin of the inherent fault-tolerance nature of GP: individuals can be lost, but their genetic material may be still be present in the remaining individuals in the population.

To the best of our knowledge, this is the first time that PGP is characterized from a fault-tolerance perspective. We contend that our conclusions can be extended to Parallel Evolutionary Algorithms (PEAs) via similar experimental validation. An extension of this work would be to plan to conduct such validation. Another promising direction is a study of load-balancing techniques for exploiting all available hosts, and of dynamic population variation methods applied in addition to the dynamics of host failures.

7. Acknowledgments

This work was supported by University of Extremadura, regional government Junta de Extremadura, National Nohnes project TIN2007-68083-C02-01 Spanish Ministry of Science and Education and by the U.S. National Science Foundation under Award #0546688.

References

[1] F. Fernandez, G. Spezzano, M. Tomassini, L. Vanneschi, Parallel Genetic Programming, in: E. Alba (Ed.), Parallel Metaheuristics, Parallel and Distributed Computing, Wiley-Interscience, Hoboken, New Jersey, USA, 2005, Ch. 6, pp. 127–153.

[2] M. Tomassini, Spatially Structured Evolutionary Algorithms, Springer, 2005.

[3] E. Cantu-Paz, A survey of parallel genetic algorithms, Calculateurs Paralleles, Reseaux et Systems Repartis 10 (2) (1998) 141–171.

[4] A. Baratloo, P. Dasgupta, Z. Kedem, CALYPSO: a novel software system for fault-tolerant parallel processing on distributed platforms, in: Proc. of the Fourth IEEE International Symposium on High Performance Distributed Computing, 1995.

[5] G. Folino, C. Pizzuti, G. Spezzano, CAGE: A tool for parallel genetic programming applications, in: J. F. M. et. al. (Ed.), Genetic Programming, Proceedings of EuroGP'2001, Vol. 2038 of LNCS, Springer-Verlag, Lake Como, Italy, 2001, pp. 64–73.

[6] Top 500 Supercomputer Sites, `http://www.top500.org/` (2009).

[7] D. Anderson, Boinc: a system for public-resource computing and storage, in: Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on, 2004, pp. 4–10.

[8] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, D. Werthimer, Seti@home: an experiment in public-resource computing, Commun. ACM 45 (11) (2002) 56–61.

[9] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, H. Casanova, Resource Availability in Enterprise Desktop Grids, Journal of Future Generation Computer Systems 23 (7) (2007) 888–903.

[10] B. Schroeder, G. A. Gibson, A Large-Scale Study of Failures in High-Performance Computing Systems, in: Proc. of the International Conference on Dependable Systems, 2006, pp. 249–258.

[11] F. C. Gartner, Fundamentals of fault-tolerant distributed computing in asynchronous environments, ACM Computing Surveys 31 (1) (1999) 1–26.

[12] M. L. Douglas Thain, The Grid 2, Morgan Kaufmann, 2004, Ch. 19, pp. 285–318.

[13] S. Ghosh, Distributed systems: an algorithmic approach, Chapman & Hall/CRC, 2006.

[14] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, F. Cappello, Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI, in: Proc. of the ACM/IEEE SC Conference, 2006.

[15] G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, J. Dongarra, Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems, in: Proc. of the International Supercomputer Conference, 2004.

[16] J. Pruyne, M. Livny, Managing checkpoints for parallel programs, in: Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '96), Honolulu, HI, 1996.

[17] D. Lombraña, F. Fernández, L. Trujillo, G. Olague, B. Segal, Customizable execution environments with virtual desktop grid computing, Parallel and Distributed Computing and Systems, PDCS.

[18] D. Lombraña, F. Fernández, Analyzing fault tolerance on parallel genetic programming by means of dynamic-size populations, in: Congress on Evolutionary Computation, Vol. 1, Singapore, 2007, pp. 4392 – 4398.

[19] I. Hidalgo, F. Fernández, J. Lanchares, D. Lombraña, Is the island model fault tolerant?, in: Genetic and Evolutionary Computation Conference, Vol. 2, London, England, 2007, p. 1519.

[20] D. L. González, F. F. de Vega, H. Casanova, Characterizing fault tolerance in genetic programming, in: Workshop on Bio-Inspired Algorithms for Distributed Systems, Barcelona, Spain, 2009, pp. 1–10.

[21] L. Trujillo, G. Olague, Automated Design of Image Operators that Detect Interest Points, Vol. 16, MIT Press, 2008, pp. 483–507.

[22] D. Andre, J. R. Koza, Parallel genetic programming: a scalable implementation using the transputer network architecture (1996) 317–337.

[23] C. Gagné, M. Parizeau, M. 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, 2003, pp. 201–208.

[24] N. Melab, S. Cahon, E.-G. Talbi, Grid computing for parallel bioinspired algorithms, J. Parallel Distrib. Comput. 66 (8) (2006) 1052–1061.

[25] D. Reed, C. Lu, C. Mendes, Reliability challenges in large systems, Future Generation Computer Systems 22 (3) (2006) 293–302.

[26] M. Shooman, Reliability of computer systems and networks: fault tolerance, analysis and design, Wiley-Interscience, 2002.

[27] E. Vargas, High availability fundamentals, Sun Blueprints series.

[28] E. Elnozahy, L. Alvisi, Y. Wang, D. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Computing Surveys (CSUR) 34 (3) (2002) 375–408.

[29] R. Guerraoui, A. Schiper, Software-Based Replication for Fault Tolerance, IEEE Computer 30 (4) (1997) 68–74.

[30] M. Jelasity, M. Preuß, M. van Steen, B. Paechter, Maintaining connectivity in a scalable and robust distributed environment, in: H. E. Bal, K.-P. Löhr, A. Reinefeld (Eds.), Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002), IEEE Computer Society, Berlin, Germany, 2002, pp. 389–394, 2nd GP2PC workshop.

[31] A. Agbaria, R. 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, IEEE Computer Society, Washington, DC, USA, 1999, p. 31.

[32] A. T. Tai, K. 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), IEEE Computer Society, Washington, DC, USA, 2005, pp. 570–579.

[33] S. L. et al., Ecj a java-based evolutionary computation research system, http://cs.gmu.edu/ eclab/projects/ecj/ (2007).

[34] S. Cahon, N. Melab, E. Talbi, Building with paradisEO reusable parallel and distributed evolutionary algorithms, Parallel Computing 30 (5-6) (2004) 677–697.

[35] M. Arenas, P. Collet, A. Eiben, M. Jelasity, J. Merelo, B. Paechter, M. Preuß, M. Schoenauer, A framework for distributed evolutionary algorithms, Lecture Notes in Computer Science (2003) 665–675.

[36] M. P. I. Forum, Mpi: a message-passing interface standard, International Journal Supercomput. Applic. 8 (3–4) (1994) 165–414.

[37] V. S. Sunderam, Pvm: A framework for parallel distributed computing, Concurrency: Practice and Experience 2 (1990) 315–339.

[38] J. Pruyne, M. Livny, Interfacing Condor and PVM to harness the cycles of workstation clusters, FGCS. Future generations computer systems 12 (1) (1996) 67–85.

[39] M. Tomassini, L. Vanneschi, J. Cuendet, F. Fernandez, A new technique for dynamic size populations in genetic programming, in: Evolutionary Computation, 2004. CEC2004. Congress on, Vol. 1, 2004.

[40] L. V. F. Fernández, M. Tomassini, Saving computational effort in genetic programming by means of plagues, Evolutionary Computation, 2003. CEC '03. The 2003 Congress on.

[41] S. Luke, G. Balan, L. Panait, Population implosion in genetic programming, Lecture Notes in Computer Science (2003) 1729–1739.

[42] P. Kouchakpour, A. Zaknich, T. Bräunl, Dynamic population variation in genetic programming, Information Sciences 179 (8) (2009) 1078–1091.

[43] P. Kouchakpour, A. Zaknich, T. Bräunl, Population variation in genetic programming, Information Sciences 177 (17) (2007) 3438–3452.

[44] F. F. de Vega, A fault tolerant optimization algorithm based on evolutionary computation, in: Proceedings of the International Conference on Dependability of Computer Systems, 2006.

[45] F. Fernández, D. Lombraña, Algoritmos evolutivos tolerantes a fallos en entornos de computación distribuida, in: XVII Jornadas de Paralelismo, Vol. 1, Albacete, Spain, 2006, pp. 401–406.

[46] M. Tomassini, Parallel and distributed evolutionary algorithms: A review, in: P. N. K. Miettinen, M.Mäkelä, J. Periaux (Eds.), Evolutionary Algorithms in Engineering and Computer Science, J. Wiley and Sons, Chichester, 1999, pp. 113,133.

[47] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, Evolutionary Computation, IEEE Transactions on 6 (5) (2002) 443–462.

[48] D. Kondo, M. Taufer, C. Brooks, H. Casanova, A. Chien, Characterizing and evaluating desktop grids: An empirical study, in: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'04), Citeseer, 2004.

[49] D. Kondo, G. Fedak, F. Cappello, A. Chien, H. Casanova, Characterizing resource availability in enterprise desktop grids, Vol. 23, Elsevier, 2007, pp. 888–903.

[50] D. Anderson, G. Fedak, The Computational and Storage Potential of Volunteer Computing, Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06).

[51] J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, USA, 1992.

[52] W. Banzhaf, F. D. Francone, R. E. Keller, P. Nordin, Genetic programming: an introduction: on the automatic evolution of computer programs and its applications, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[53] W. Langdon, R. Poli, Foundations of genetic programming, Springer Verlag, 2002.

[54] R. Poli, N. McPhee, General schema theory for genetic programming with subtree-swapping crossover: Part I, Evolutionary Computation 11 (1) (2003) 53–66.

[55] R. Poli, N. McPhee, General schema theory for genetic programming with subtree-swapping crossover: Part II, Evolutionary Computation 11 (2) (2003) 169–206.