Characterizing Fault Tolerance in Genetic Programming

Daniel Lombraña González[*]
University of Extremadura
Avd. Santa Teresea de Jornet, 38
06800 Mérida, Badajoz, Spain
daniellg@unex.es

Francisco Fernández de Vega[†]
University of Extremadura
Avd. Santa Teresea de Jornet, 38
06800 Mérida, Badajoz, Spain
fcofdez@unex.es

Henri Casanova[‡]
University of Hawai'i at Manoa
POST # 317
1680 East-West Road,
Honolulu, HI 96822, U.S.A.
henric@hawaii.edu

ABSTRACT

Evolutionary Algorithms (EAs), and particularly Genetic Programming (GP), are techniques frequently employed to solve difficult real-life problems, which can require up to days or months of computation. One approach to reduce the time to solution is to use parallel computing on distributed platforms. Distributed platforms are prone to failures, and when these platforms are large and/or low-cost, failures are expected events rather than catastrophic exceptions. Therefore, fault tolerance and recovery techniques often become necessary. It turns out that Parallel GP (PGP) applications have an inherent ability to tolerate failures. This ability is quantified via simulation experiments performed using failure traces from real-world distributed platforms, namely, desktop grids (DGs), for two well-known GP problems. A simple technique is then proposed by which PGP applications can better tolerate the different, and often high, failures rates seen in different platforms.

Categories and Subject Descriptors

I.2.8 [**Artificial intelligence**]: Problem solving, control methods and search—*heuristic methods*.

General Terms

Algorithms, performance, reliability.

Keywords

Fault-tolerance, parallel genetic programming, desktop grids.

[*]D. Lombraña
[†]Dr. F. Fernández
[‡]Dr. H. Casanova

BADS'09, June 19, 2009, Barcelona, Spain.

1. INTRODUCTION

Evolutionary Algorithms (EAs) are a class of a well-known techniques to solve difficult problems. Unfortunately, for many of these problems, EAs require large execution times, leading to often prohibitively large times to solution. One known approach to address this problem is to use parallel computing so as to execute EAs on potentially large-scale parallel computing platforms. A number of such Parallel Evolutionary Algorithms (PEAs) have been successfully proposed and developed (see for instance [13, 30]). Software frameworks for PEAs have been proposed and implemented, such as Calypso [5], and others in the specific contexts of Parallel Genetic Programming (PGP) [15].

Parallel algorithms, and thus PEAs, must be executed on platforms that consist of multiple computing elements, or *processors*. Such parallel platforms include shared memory systems that are multi-socket and/or multi-core; tightly packaged and interconnected homogeneous clusters of potential large numbers of identical single systems; or distributed networks of single systems and/or clusters, for instance on the Internet.

Two types of parallel platforms that have reached very high scale and thus can provide enormous levels of performance are *clusters* and *desktop grids*. Clusters have become mainstream and are today the most common parallel computing platforms. Large clusters account for more than 80% of the Top500 list, which ranks the 500 fastest supercomputers based on the LINPACK Benchmark every 6 months [31]. Such clusters can reach enormous scale, from tens to hundreds of thousands of processors. The term "desktop grid" (DG) is used to refer to distributed networks of heterogeneous single systems that contribute compute cycles whenever idle. Perhaps the best known example of such a system is the Berkeley Open Infrastructure for Network Computing (BOINC) [1], which hosts, among other projects, SETI@home [28]. These "volunteer computing" systems aggregate the compute power of idle computers provided by individuals worldwide (e.g., home computers, office workstations). At the time this article is being written, the BOINC network consists of half a million of such donated computers, with many of them harboring multiple processor cores. DGs of smaller, but still impressive, scales are deployed within enterprises or data centers [8]. These deployments often comprises computers that are more powerful, more available, and less heterogeneous than their Internet-wide counterparts.

One of the crucial advantages of DG systems is that they provide large-scale parallel computing capabilities, admittedly for restricted classes of applications, at very low cost.

The aforementioned large-scale parallel computing platforms hold promises for executing large and demanding PEAs. But with large scale comes a higher probability that processors experience failures during application execution (e.g., a crash). (In this paper we use the terms "failure" and do not make the subtle distinction between "failure" and "fault", which is not necessary for our purpose.) Failures are known to occur frequently in large-scale clusters [3]. In DG platforms failures are in fact the common case. Indeed, participating computers are typically configured so that they can run a DG application task only when idle. A computer can then be reclaimed by its owner at any time (e.g., as soon as the owner starts an application, as soon as the keyboard/-mouse is used). A DG application task is thus often abruptly suspended or terminated, which is seen as a failure from the perspective of the application.

To cope with failures, many researchers have developed techniques for an application to not be terminated when one or more of the participating processors experiences a failure. This ability is called *fault tolerance*, and enforces that the application behave in a well-defined manner (e.g., with graceful degradation of performance) when a failure occurs [18]. A number of well-known fault tolerance techniques have been developed [10]. These techniques can be used with parallel applications, and make it possible to tolerate many types of computation and communication failures [19]. In general, the use of fault tolerance techniques mandates that the application implementation, and even the parallel algorithms themselves, be modified. This modification can represent a heavy burden for the application developer due to heightened complexity of both the algorithms and their implementations. A number of generic fault tolerance solutions have thus been developed as libraries or software environments [6, 16, 26].

To the best of our knowledge, there has been little scientific study of the fault tolerance features of PEAs in general, and of PGP applications in particular. Nevertheless, there are tools available to easily parallelize and run any EA, and thus GP applications, in volunteer computing environments [24], in which failures are common.

In previous work [23, 20] we have provided preliminary answers to the question of fault tolerance for PGP applications under several simplified assumptions. The results therein showed that PGP applications expectedly exhibit inherent fault-tolerant behaviors. Therefore, there should be an opportunity to run these application on large-scale computing infrastructures that are subject to failures without the burden of implementing/using specific fault tolerance techniques, and without sacrificing overall application efficiency significantly.

In this work we extend and strengthen these preliminary results in two ways. First, we conduct simulations of two PGP applications using processor availability data collected from real-world DG deployments (rather than using naïve, and ultimately unrealistic, processor availability models). Our results are thus representative of what one would expect in real DG platforms. Second, due to our use of availability data from several real platforms, we are able to study the effects of different failure rates on application execution. This leads us to a simple technique via which a PGP application can tolerate various failure rates, and thus easily exploit different DG systems to the best of their potentials. Besides the specific interest of this study to the GP community, we contend that our conclusion could be easily extended and generalized to other EAs.

This paper is organized as follows. Section 2 reviews related work. Section 3 describes the different types of failures that may arise as well as the different fault tolerance techniques that can be employed in a distributed system. Section 4 discusses our experiments and experimental results. Section 5 concludes the paper with a discussion of our results and future directions.

2. BACKGROUND AND RELATED WORK

Since the early days of applying EAs, and especially GP, to real-world problems, practitioners have often found that the time to solution on a single computer is prohibitively long. For instance, Trujillo et al. recently required more than 24 hours to obtain a solution to a computer vision problem [32]. Times to solution on a single computer can in fact be much longer, measured in weeks or even months. Consequently, several researchers have studied the application of parallel computing techniques and distributed computing platforms in conjunction with Spatially Structured EAs in order to shorten times to solution [13, 30].

A distributed system, in the sense of a parallel application running on a distributed platform, can be defined based on its logical or functional distribution of processing capabilities [19]. The logical distribution is typically based on the following set of criteria:

- *Multiple processes* – The system consists of one or more sequential processes, each with an independent thread of control.
- *Interprocess communication* – Processes can communicate via messages.
- *Disjoint address spaces* – Processes have disjoint address spaces.
- *Collective goal* – Processes interact among each other in order to meet a common goal.

PEAs structured according to the above distributed system paradigm have been developed and used for decades, going back to work on the now defunct Transputer platform [2], including more recent software frameworks such as Beagle [17], grid based tools like Paradiseo [7], or DG infrastructures running BOINC-based EAs [24]. Yet, as distributed computing platforms become large and/or low-cost, failures can and do occur with higher probability. These failures can be local failures, affecting a single processor, or communication failures, affecting a large set of the participating processors. In general, any such failure can completely disrupt a running application and prevent final results from being obtained, mandating that the application be restarted from scratch. As seen in Section 1 failures are common in large-scale distributed computing platforms, and even routine in the case of DG platforms. For this reason, techniques for fault tolerance are needed for parallel applications in general, and in our case PEAs, to be able to benefit from large-scale distributed computing platforms.

Failures can be classified in different levels. In this paper we only take into account the failures at the process

level. A complete description of failures in distributed systems is beyond the scope of our discussion. The possible failures that can occur at the process level are: crash, omission, transient, Byzantine, software, temporal and security failures [19], which we explain further in Section 3.

All these failures can be alleviated, and sometimes completely circumvented. Three examples of fault tolerance techniques developed are checkpointing [11], redundancy [27], and rejuvenation frameworks [29]. These techniques typically require extra computing resources/time. Additionally, they often need to be embedded in the application, often leading to an increase in complexity of the implementation and of the algorithms.

While currently available PEAs software frameworks may use some fault tolerant techniques, none of them, nor researchers using them, have previously considered the specific problem of fault tolerance given the very specific features of PEAs. We have ourselves started tackling this problem, first Fernandez [9] and later Lombraña et. al. [14], in the context of PGP. An approach for fault tolerance for PGP application was described based on the previous study of the *plague* operator [12]. This operator was initially proposed for controlling the bloat phenomenon that hinders GP applications [4]. "Bloat" refers to the fact that GP individuals tend to increase both in size and complexity from one generation to the next. One of the consequences of bloat is that the evaluation phase of the individuals becomes progressively more expensive. This is a well-known problem for which different solutions have been proposed [25].

The plague operator eliminates the worst individuals from the population to control bloat while maintaining the quality of the obtained solutions. A key insight here is that this elimination step is akin to the loss of individuals due to losing a process due to a failure in master-worker execution of a parallel application running on a distributed computing platform. Building on this insight, Lombraña et al. [14] have proposed that PGP be considered inherently fault-tolerant if one considers that the plague operator in use is simply the removal of random individuals (due to failures) rather than the removal of the worst individuals (as in the original plague idea). The work of Lombraña et al. was done by simulating the loss of individuals with different ratios. The results were encouraging in the sense that, indeed, PGP are likely inherently fault-tolerant due to platform failures providing an effective plague operator. However, Lombraña et al. did not characterize the fault tolerance of PGP and, more importantly, used a naïve failure model rather than one inspired by what is observed in real-world systems.

In this paper we build on the work in [14] to characterize the fault tolerance of PGP more precisely based on models that come from real-world distributed computing platforms. More specifically, we focus on Desktop Grid (DG) platforms. We content that DGs are ideally suited and relevant to our study. They are known to exhibit large numbers of failures, and their failure behavior have been studied in the literature [21]. Very large DG platforms are available [1] and are very low-cost when compared to clusters of comparable scale. While clusters can accommodate a much wider range of applications (e.g., communication-intensive tightly coupled parallel computations), PGP applications are loosely coupled and thus well-suited to DG platforms. In summary, DGs are very promising platforms for PGP applications, and their high failure rate make them a great test case for studying and characterizing the fault tolerance abilities of PGP applications. If an application can tolerate the high failure rates of DG platforms, it should be able to tolerate those of most other parallel computing platforms.

3. FAULT TOLERANCE

Fault tolerance is the ability of a system to behave in a well-defined manner once a failure occurs. Our view of the distributed system is a process-level view, so failures are related only to the process level. According to Ghosh [19], failures can be classified as follows:

- *Crash failure* – A process undergoes a crash failure when it permanently ceases to execute its actions. This is an irreversible change. A variation of this failure is the *Napping failure*. In this case, the failure is reversible, that is, a process may crash and after a period of time can be restored or repaired.
- *Omission failure* – An omission failure occurs when a receiver process does not receive one or more messages sent by a transmitter process.
- *Transient failure* – This type of failure can perturb the state of processes in an arbitrary way (like a power surge, lightning, etc.).
- *Byzantine failure* – Considering a process i that sends the value of a local variable to all its neighbors. Then, the following are examples of Byzantine failures:
  - Two different neighbors j and k receive values x and y, where $x \neq y$.
  - Every neighbor receives a value z where $z \neq x$.
  - One or more neighbors do not receive any data from process i.
- *Software failure* – This failure is commonly caused by coding or human errors, software design errors, memory leaks and/or problems with inadequacy specifications.
- *Temporal failure* – Considering a real system in which some actions require to be done by a specific deadline, if this deadline is not met then a temporal failure occurs.
- *Security failure* – Virus or other kind of malicious software can lead to unexpected behaviors, which can manifest themselves as failures.

The set of all possible configurations or behaviors of a distributed system can be divided into two classes: *legal* and *illegal* configurations. Well-behaved systems are always in a legal configuration, because of: (i) a proper initialization makes the initial configuration legal, (ii) a closure property of normal actions that transforms one legal configuration into another; and (iii) the absence of failures or perturbations.

However, in practice, a well-behaved system can go from a legal configuration to an illegal one due to the following reasons [19]: (i) *Transient failures*: the system state can be corrupted in an unpredictable way; (ii) *Topology changes*: the system topology changes at runtime when a node crashes, or a new node is added; and (iii) *Environmental changes*: the

environment – external variables that should only be read – may change without notice.

Once a system configuration becomes illegal, a mechanism is required to go from the illegal configuration to a legal one. There are four major types of such fault tolerance:

- *Masking tolerance* – When a fault is masked, its occurrence has no impact on the application. Masking techniques are very useful in safety-critical systems and real-time systems.
- *Non masking tolerance* – Faults may temporarily affect the application but the system can eventually be restored to a legal configuration.
- *Fail-safe tolerance* – Certain faulty configurations are considered harmless because they do not affect the application severely. Fail-safe tolerance relaxes its requirements by only avoiding those faulty configurations that are not harmless (even when failures occur).
- *Graceful degradation* – One may neither mask, nor fully recover from failures, but exhibit a degraded behavior which is considered acceptable. The notion of acceptability is highly subjective, and is typically user-defined.

This work presents an analysis of PGP based on the above description of distributed systems, their failures and the fault tolerance techniques. To the best of our knowledge, this is the first time that a fault tolerance characterization of PGP in obtained that is informed by real-world distributed computing platforms.

For our purpose we first need to choose the kind of parallelization employed in the PGP application. Parallelism has been traditionally applied to GP at one of two levels: the individual level or the population level [30]. 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 used (ring, multi-dimensional grids, etc.). In line with previous studies on this topic [30], we consider master-worker parallelization at the individual level. A server, the "master", is in charge of the main algorithm and the whole population. It sends non evaluated individuals to different processes, the "workers", in the distributed system. This is effective because the most time-consuming portion 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 server program 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 one or more 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 assumption is justified because the master is under the control of a single organization/person, and, besides, known fault tolerance techniques (e.g., primary backup [27]) could be easily used to tolerate master failures.

Our system thus suffers only from omission failures: (i) 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 (ii) the master sends N individuals with $N > 0$ to a worker, the worker receives them but never returns them. This can happen because the worker crashes or because the returned individuals are lost during the transmission.

4. EXPERIMENTS AND RESULTS

We study the fault tolerance characteristics of PGP via simulation experiments. We use simulation because it 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 [21], so that fair comparison between simulated application executions is possible.

We perform experiments for two well-known GP problems, even parity 5 (EP5) and 11-multiplexer (11M) [22]. 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 are Koza-I/II standard [22] (including optimal population size for both problems: choosing smaller or larger sizes will lead to worse results). See Tab. 1 for all details. For both problems, fitness is measured as the error in the obtained solution: A fitness of zero means that a perfect solution has been found. Even when the required time for fitness evaluation for the benchmark problem is short, we simulate a larger evaluation time so as to simulate real-life hard problems rather than mere benchmark problems.

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)	Tournament (7)
Max Depth in Cross	17	17
Max Depth in Mutation	17	17
ADFs	Yes	-

To achieve our goal of characterizing the fault-tolerant nature of PGP we run two kinds of experiments: (i) using a failure-free environment, i.e., assuming that no failures occur during evaluation steps at the workers; and (ii) replaying and simulating failures traces from real-world DG platforms

so that workers can fail. In a failure-free environment, the amount of available computing power is steady throughput execution, while in a real-world environment it varies between generations. If the results obtained for both problems are of similar quality in both scenarios then this is a strong indication that PGP is inherently fault-tolerant in practical settings.

We perform simulations of DG platforms and of host availability in these platforms based on three real-world traces of host availability [21]: *ucb*, *entrfin*, and *xwtr*. These traces are time-stamped observation of the host availability in three DG platforms. 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 [21] for full details on the measurement methods and the traces.

Fig. 1 shows example availability data from the ucb trace: the number of available hosts in the platform versus time over 24 hours. The figures show 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 fixed the maximum number of generations to 51 generations (which corresponds to 5 hours of execution in a failure-free environment).

We arbitrarily select the time within each 24-hour trace segment with the largest number of available hosts as the beginning of the problem execution. In our experiments we consider the worst-case scenario in which hosts that become unavailable never become available again and thus cannot be used again. The number of available hosts with this assumption, starting from the time at which the number of available hosts is the largest, is shown in Fig. 1 by the "trace without return" curve. We demonstrate that good results can be achieved even with this worst-case assumption. Even better results could be achieved in practice if the master is implemented so that worker hosts can be re-acquired.

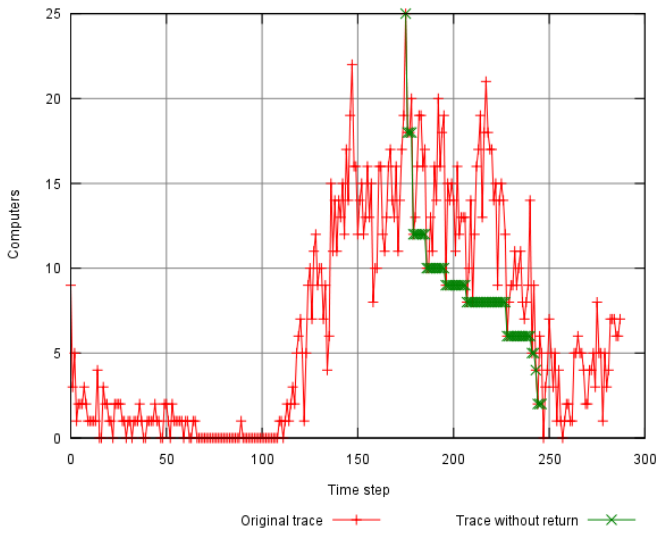

Figure 1: Host availability for 1 day of the ucb trace.

The master does no attempt to detect failures and no fault tolerance technique is used (e.g., no replication of individuals over multiple workers). The master waits a time T per generation, defined based on the time needed for a generation in the failure-free case, and proceeds to the next generation with the available individuals at that time. Hence, if a worker disappears, I individuals will be lost and will not participate in the following generations of the algorithm (due to our worst-case assumption). Note that the execution, which could lead to worse results compared to an execution in a failure-free environment due to lost individuals, becomes progressively less computation-demanding as generations become progressively smaller. At the onset of each generation the master sends an as equal as possible number of individuals to each worker. This is because the master assumes homogeneous workers and thus strives for perfect load-balancing.

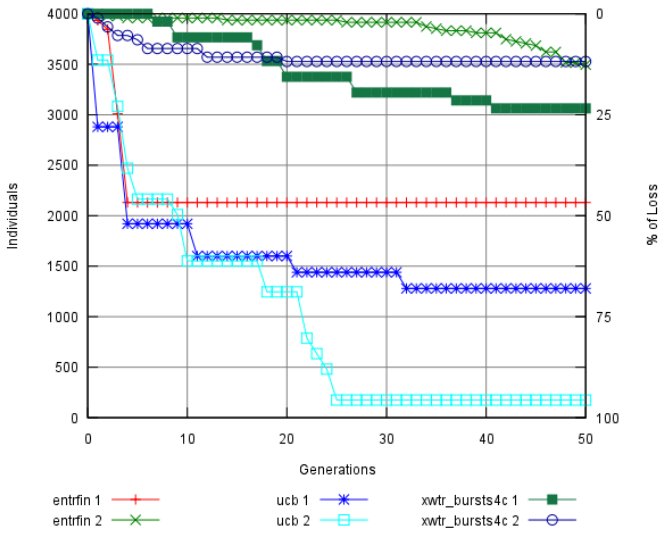

Figure 2: Population size vs. generation.

We have to bear in mind that the execution times per generation in the failure-free and the failure-prone environments are identical: with P individuals to be evaluated at a given generation and W workers, we send $\frac{P}{W} = I$ individuals to each worker; when a worker fails $\frac{P}{W}$ individuals are lost; given that those individuals are discarded for the next generation, the remaining workers will continue evaluating $\frac{P}{W}$ individuals each, regardless of the number of failures.

In order to provide conclusions with confidence, the data obtained by running 100 times each of the experiments have been statistically analyzed. Firstly, we analyzed the normality of the data using the Kolgomorov-Smirnov and Shapiro-Wilk tests, 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. Tables 4, 5, 6 and 7 present the Wilcoxon analysis of the data. The following sections discuss our results in depth for each problem.

4.1 Even Parity 5 (EP5)

As explained in the previous section, we use three traces collected on three DG platforms: entrfin, ucb and xwtr. Based on these traces we simulate the loss of hosts based on

the real-world traces. Recall that we assume that once hosts become unavailable they never become available again. The implication is that the number of individuals per generation can decrease at each generation due to failures.

Fig. 2 shows decreases in the number of individuals at each generation for executions of the EP5 problem when simulated over two 24-hour periods, denoted by Day1 and Day2, randomly selected out of each of our traces, for a total of 6 experiments. Results from these simulation experiments is summarized in Tab. 2, and we see in the second column of the table that the fraction of lost individuals depends strongly on the trace and on the day. The Day1 period of the entrfin trace exhibits a severe loss of individuals, with almost half of the individuals lost during the first 10 generations. For the Day2 period of the ucb trace almost all the population is lost after 25 generations (96.15% loss). The xwtr trace exhibits more moderate losses, with overall 23.52% and 12.08% loss after 51 generations for each day, respectively.

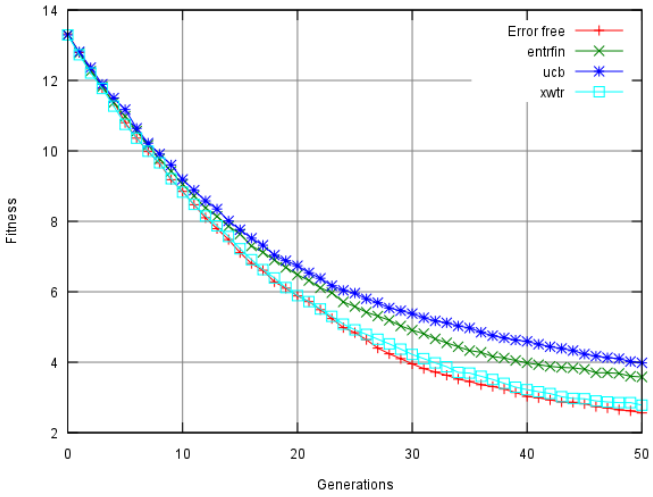

Figure 3: EP5 fitness vs. generation (Day1)

We performed 100 runs for the Day1 and 2 periods of each trace, accounting for the fact that different individuals can be lost depending on which individuals were arbitrarily assigned to which hosts. Fig. 3 shows the obtained fitness versus the number of generations, averaged over the 100 experiments, for each trace. The figure also shows the results assuming a failure-free environment.

The obtained fitness in the error-free case is 2.56, and it is equal to 3.58, 3.98, and 2.78 for entrfin, ucb, and xwtr, respectively (see Tabs. 4 and 5 for statistical significance of results). The worst fitness is obtained when the individual loss rate is the highest, i.e., the ucb trace with 68% of lost individuals overall (see Tab. 2). The next worst fitness is for the entrfin trace, with which almost 50% of the population is lost in the first generation, but with a stable population afterwards. As a result, the obtained fitness is comparable to that in the failure-free case. Finally, the trace with fewer losses leads to a fitness very similar to that in the failure-free case. Tab. 2 shows a summary of the different obtained fitness values for all the traces.

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 the 50%, then solution quality is significantly diminished. Since real-world DG platforms 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.

Table 2: Obtained fitness for EP5 and 11M

		EP5	11M
Trace	Loss(%)	Fitness	Fitness
Error free	0.00	2.56	14.4
entrfin (Day1)	48.02	3.58	30.36
entrfin (Day2)	13.04	2.44	18.84
ucb (Day1)	68.00	3.98	49.04
ucb (Day2)	96.15	5.13	67.07
xwtr (Day1)	23.52	2.78	20.92
xwtr (Day2)	12.08	2.61	16.16

Table 3: EP5 fitness with increased population

Error Free fitness = 2.56						
	Day1			Day2		
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

Tab. 3 shows results for the increased initial population size, based on simulations for the Day1 and Day2 periods of all three traces. Overall, we see that increasing the initial population size is effective to tolerate failures while preserving (and even improving) solution quality. For the Day1 period of the entrfin trace, which leads to a loss of individuals of 48.02%, we see that starting with 50% extra individuals ensures solution quality on par with that in the failure-free case. Indeed, the difference between the error-free fitness, 2.56, and the obtained fitness of the parallel algorithm, 2.85, is insignificant.

For the Day1 period of the ucb traces the loss of individuals is 68%. Therefore, increasing the initial population by 50% is not sufficient to compensate for lost individuals. Nevertheless, solution quality, 3.17, is not far from the 2.56 of the failure-free fitness. Results with the xwtr trace show that when the loss of individuals does not exceed 25%, 20% extra individuals provides is sufficient to obtain solutions of the same quality as in the failure-free case, see Tabs. 4 for Day1 and 5 for Day2. In fact, each increase in the initial population size provides good fitness with the same quality as in the error-free case, except for the ucb traces where the extra 50% individuals are not enough for mitigating the large ratio of losses: 96.15% and 68%.

Increasing the initial population size will require more compute time per generation since more individuals need

Table 4: EP5 fitness comparison between traces and error-free case using Wilcoxon test (Day1): Not significantly different means comparable fitness quality with error-free experiment.

Error Free fitness = 2.56			
Trace	Fitness	Wilcoxon test	Significantly different?
Entrfin	3.58	W = 6726, p-value = 1.843e-05	yes
Entrfin 10%	3.52	W = 6685, p-value = 2.707e-05	yes
Entrfin 20%	3.01	W = 5760, p-value = 0.05956	yes
Entrfin 30%	3.13	W = 5942.5, p-value = 0.01941	yes
Entrfin 40%	**2.80**	**W = 5355, p-value = 0.3773**	**no**
Entrfin 50%	**2.85**	**W = 5620, p-value = 0.1233**	**no**
Ucb	3.98	W = 7274, p-value = 1.789e-08	yes
Ucb 10%	3.75	W = 6927.5, p-value = 1.799e-06	yes
Ucb 20%	3.61	W = 6769, p-value = 1.123e-05	yes
Ucb 30%	3.33	W = 6390, p-value = 0.0005542	yes
Ucb 40%	3.35	W = 6408, p-value = 0.000464	yes
Ucb 50%	3.17	W = 6080, p-value = 0.007298	yes
Xwtr	**2.78**	**W = 5509, p-value = 0.2043**	**no**
Xwtr 10%	**2.40**	**W = 4762, p-value = 0.5532**	**no**
Xwtr 20%	**2.32**	**W = 4643.5, p-value = 0.3753**	**no**
Xwtr 30%	**2.46**	**W = 4802, p-value = 0.6221**	**no**
Xwtr 40%	**2.15**	**W = 4363, p-value = 0.1121**	**no**
Xwtr 50%	**2.13**	**W = 4296.5, p-value = 0.08027**	**no**

to be evaluated. However, it is important to note that this extra time is comparable with the extra time that would be required by standard fault tolerance techniques (e.g., failure detection and use of available workers for computing individuals previously lost). We therefore conclude that increasing the population size is sufficient for improving the GP quality of results when the failure rate is known.

4.2 11 Bits Multiplexer (11M)

For the 11M problem we use the same experimental procedure as in the previous section for the EP5 problem for the same host availability traces.

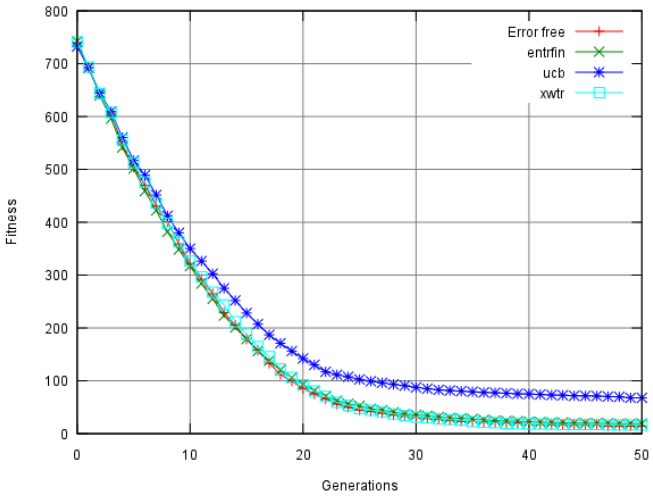

Figure 4: 11M fitness vs. generation (Day2)

In the failure-free case the fitness for 11M is 14.4, while the average values for 100 repetitions for the entrfin, ucb, and xwtr traces on Day2 are 18.84, 67.07, and 16.16, respectively, as seen in the last column of Tab. 2. Fig. 4 shows the fitness versus the generations for simulations using the Day2 period of our three host availability traces. The worst fitness is for the ucb trace due to the high loss rate. For the other two traces the obtained fitness values are closer to that in the failure-free case since fewer individuals are lost (under 20% see Tabs. 7).

The 11M problem was also executed after increasing the initial population size by 10, 20, 30, 40 and 50%. Tab. 8 shows the obtained fitness for the entrfin, ucb and xwtr traces. We see that, like in the previous section, increasing the initial population size is effective. For the Day1 period of the entrfin trace, an increase of the 50% to the initial population size leads to an average fitness value of 10.08, which has the same quality as the fitness in the failure-free case (see Tab. 6). Therefore, we show here again that the increase in initial population size compensates for the loss of about half of the individuals due to failures. Experiments with the Day1 period of the ucb trace also show benefits of using a larger initial population (recall that 68% of the individuals are lost with this trace). The obtained average fitness improves from 47.44 with 10% extra individuals to 25.2 for 50% extra individuals.

Finally, experiments with the xwtr trace show that in all cases the achieved average fitness is close to or even slightly better than that in the failure-free case (see Tabs. 6 and 7). For example, for the Day2 period, for 40% extra individuals the average fitness is 1.20, while it is 14.4 in the failure-free case.

Table 5: EP5 fitness comparison between traces and error-free case using Wilcoxon test (Day2): Not significantly different means comparable fitness quality with error-free experiment.

Error Free fitness = 2.56			
Trace	Fitness	Wilcoxon test	Significantly different?
Entrfin	**2.44**	**W = 4778.5, p-value = 0.5815**	**no**
Entrfin 10%	**2.65**	**W = 5201.5, p-value = 0.6167**	**no**
Entrfin 20%	**2.29**	**W = 4571, p-value = 0.2863**	**no**
Entrfin 30%	**2.36**	**W = 4732.5, p-value = 0.505**	**no**
Entrfin 40%	2.01	W = 4098, p-value = 0.02458	yes
Entrfin 50%	1.92	W = 3994.5, p-value = 0.01213	yes
Ucb	5.13	W = 8735.5, p-value < 2.2e-16	yes
Ucb 10%	5.21	W = 8735.5, p-value < 2.2e-16	yes
Ucb 20%	4.68	W = 8266.5, p-value = 6.661e-16	yes
Ucb 30%	4.50	W = 8152, p-value = 6.439e-15	yes
Ucb 40%	4.71	W = 8325.5, p-value = 2.220e-16	yes
Ucb 50%	4.47	W = 8024.5, p-value = 6.95e-14	yes
Xwtr	**2.61**	**W = 5238.5, p-value = 0.5524**	**no**
Xwtr 10%	**2.66**	**W = 5215.5, p-value = 0.5927**	**no**
Xwtr 20%	**2.42**	**W = 4686.5, p-value = 0.4364**	**no**
Xwtr 30%	**2.33**	**W = 4611.5, p-value = 0.3336**	**no**
Xwtr 40%	1.96	W = 4033.5, p-value = 0.01574	yes
Xwtr 50%	2.24	**W = 4511, p-value = 0.2226**	**no**

Table 6: 11M fitness comparison between traces and error-free case using Wilcoxon test (Day1): Not significantly different means comparable fitness quality with error-free experiment.

Error Free fitness = 14.40			
Trace	Fitness	Wilcoxon test	Significantly different?
Entrfin	30.36	W = 5858, p-value = 0.003781	yes
Entrfin 10%	28.23	W = 5558.5, p-value = 0.04374	yes
Entrfin 20%	26.20	W = 5592, p-value = 0.03438	yes
Entrfin 30%	**22.98**	**W = 5470, p-value = 0.08639**	**no**
Entrfin 40%	**8.88**	**W = 4866, p-value = 0.5631**	**no**
Entrfin 50%	**10.08**	**W = 4761.5, p-value = 0.2740**	**no**
Ucb	49.04	W = 6413.5, p-value = 9.58e-06	yes
Ucb 10%	47.44	W = 6288, p-value = 4.426e-05	yes
Ucb 20%	35.24	W = 5934, p-value = 0.001779	yes
Ucb 30%	39.20	W = 6306.5, p-value = 3.755e-05	yes
Ucb 40%	**23.09**	**W = 5483, p-value = 0.07804**	**no**
Ucb 50%	25.20	W = 5696.5, p-value = 0.01573	yes
Xwtr	**20.92**	**W = 5419, p-value = 0.1222**	**no**
Xwtr 10%	**15.36**	**W = 5120.5, p-value = 0.6318**	**no**
Xwtr 20%	**8.64**	**W = 4752.5, p-value = 0.2565**	**no**
Xwtr 30%	**6.8**	**W = 4730.5, p-value = 0.2165**	**no**
Xwtr 40%	**5.12**	**W = 4676, p-value = 0.1288**	**no**
Xwtr 50%	**10.32**	**W = 4844.5, p-value = 0.4939**	**no**

Table 7: 11M fitness comparison between traces and error-free case using Wilcoxon test (Day2): Not significantly different means comparable fitness quality with error-free experiment.

Error Free fitness = 14.40			
Trace	Fitness	Wilcoxon test	Significantly different?
Entrfin	**18.84**	**W = 5282.5, p-value = 0.2797**	**no**
Entrfin 10%	**14.36**	**W = 5077.5, p-value = 0.755**	**no**
Entrfin 20%	**10.40**	**W = 4881.5, p-value = 0.6093**	**no**
Entrfin 30%	**10.40**	**W = 4923.5, p-value = 0.7464**	**no**
Entrfin 40%	1.92	W = 4485, p-value = 0.00752	yes
Entrfin 50%	1.92	W = 4447, p-value = 0.003106	yes
Ucb	67.07	W = 7182, p-value = 2.870e-10	yes
Ucb 10%	73.76	W = 7516.5, p-value = 1.436e-12	yes
Ucb 20%	49.44	W = 6805.5, p-value = 7.762e-08	yes
Ucb 30%	54.98	W = 7009, p-value = 4.39e-09	yes
Ucb 40%	46.80	W = 6478.5, p-value = 4.842e-06	yes
Ucb 50%	50.78	W = 6758, p-value = 1.449e-07	yes
Xwtr	**16.16**	**W = 5098, p-value = 0.6926**	**no**
Xwtr 10%	**10.56**	**W = 4923.5, p-value = 0.7464**	**no**
Xwtr 20%	**8.62**	**W = 4788.5, p-value = 0.3423**	**no**
Xwtr 30%	**5.76**	**W = 4645, p-value = 0.08845**	**no**
Xwtr 40%	1.2	W = 4438.5, p-value = 0.002680	yes
Xwtr 50%	1.6	W = 4483.5, p-value = 0.007348	yes

Table 8: 11M Fitness with increased population

Error Free fitness = 14.40						
	Day1			Day2		
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

4.3 Summary of Results

Based on two standard GP well-known applications, we have shown that PGP applications based on the master-worker model and running on real-world DG platforms that exhibit large numbers of failures can achieve solution qualities close to those in a failure-free environment, without resorting to any fault tolerance technique. There is a degradation of the solution quality as the failure rate increases, and this degradation is approximately linear. This is a form of *graceful degradation*. We can now say that PGP inherently provides graceful degradation, without need for built-in fault tolerance. Additionally, we have shown that if one knows an estimate of the failure rate of the target platform for the application at hand, then it is possible to mitigate the effect of failures by incrementing the initial population size.

5. 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*. Our results suggest that for cases in which optimal initial population size is employed, and the fraction of lost individuals due to failures is lower than 30%, then it is possible to run the application without using any kind of fault tolerance and still obtain high-quality results. If an estimate of this loss rate is available, then it is possible to improve solution quality by increasing the initial population size accordingly. This makes it possible to achieve high-quality solutions even for loss rates higher than 30%.

We contend that our conclusions can be generalized and extended to Parallel Evolutionary Algorithms (PEAs) via similar experimental validation. We plan to conduct such validation in the future. We will also conduct simulations in which hosts that have become unavailable can later become available again and used by the application, and we will quantify the improvement over the worst-case scenario results presented in this paper.

6. 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.

7. REFERENCES

[1] D. 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.

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

[3] S. B. and G. G. A. A Large-Scale Study of Failures in High-Performance Computing Systems. In *Proceedings of the International Conference on Dependable Systems*, pages 249–258, 2006.

[4] W. Banzhaf and W. B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, Mar. 2002.

[5] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: a novel software system for fault-tolerant parallel processing on distributed platforms. *hpdc*, 00:122, 1995.

[6] C. C., H. T., L. P., P. L., R. A., R. E., and C. F. Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI. In *Proceedings of the ACM/IEEE SC Conference*, Nov. 2006.

[7] S. Cahon, N. Melab, and E. Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[8] K. D., F. G., C. F., C. A. A., and C. H. Resource Availability in Enterprise Desktop Grids. *Journal of Future Generation Computer Systems*, 23(7):888–903, 2007.

[9] 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.

[10] M. L. Douglas Thain. *The Grid 2*, chapter 19, pages 285–318. Morgan Kaufmann, 2004.

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

[12] 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*, 2003.

[13] F. Fernandez, G. Spezzano, M. Tomassini, and L. Vanneschi. Parallel genetic programming. In E. Alba, editor, *Parallel Metaheuristics*, Parallel and Distributed Computing, chapter 6, pages 127–153. Wiley-Interscience, Hoboken, New Jersey, USA, 2005.

[14] F. Fernández and D. Lombraña. Algoritmos evolutivos tolerantes a fallos en entornos de computación distribuida. In *XVII Jornadas de Paralelismo*, volume 1, pages 401–406, Albacete, Spain, September 2006.

[15] G. Folino, C. Pizzuti, and G. Spezzano. CAGE: A tool for parallel genetic programming applications. In J. F. M. et. al., editor, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 64–73, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[16] F. G., G. E., B. G., A. T., C. Z., P.-G. J., L. K., and D. J. Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems. In *Proceedings of International Supercomputer Conference*, June 2004.

[17] C. Gagné, M. Parizeau, and 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*, pages 201–208, May 11-14 2003.

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

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

[20] I. Hidalgo, F. Fernández, J. Lanchares, and D. Lombraña. Is the island model fault tolerant? In *Genetic and Evolutionary Computation Conference*, volume 2, page 1519, London, England, July 2007.

[21] D. Kondo, G. Fedak, F. Cappello, A. Chien, and H. Casanova. Characterizing resource availability in enterprise desktop grids. volume 23, pages 888–903. Elsevier, 2007.

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

[23] D. Lombraña and F. 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.

[24] D. Lombraña, F. Fernández, L. Trujillo, G. Olague, and B. Segal. Customizable execution environments with virtual desktop grid computing. *Parallel and Distributed Computing and Systems, PDCS*, 2007.

[25] S. Luke and L. Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, Fall 2006.

[26] J. Pruyne and M. Livny. Managing checkpoints for parallel programs. In *Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '96)*, Honolulu, HI, April 1996.

[27] G. R. and S. A. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[28] Sullivan, Werthimer, Bowyer, Cobb, Gedye, and Anderson. A New Major SETI Project based on project SERENDIP data and 100,000 Personal Computers. In *Astronomical and Biochemical Origins and the Search for Life in the Universe*, 1997.

[29] A. T. Tai and 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)*, pages 570–579, Washington, DC, USA, 2005. IEEE Computer Society.

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

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

[32] L. Trujillo and G. Olague. Automated Design of Image Operators that Detect Interest Points. volume 16, pages 483–507. MIT Press, 2008.