Software testing is a fundamental part of system development. As software grows, its test suite becomes larger and its execution time may become a problem to software developers. This is especially the case for agile methodologies, which preach a short develop/test cycle. Moreover, due to the increasing complexity of systems, there is the need to test software in a variety of environments. In this paper, we introduce GridUnit, an extension of the widely adopted JUnit testing framework, able to automatically distribute the execution of software tests on a computational grid with minimum user intervention. Experiments conducted with this solution have showed a speed-up of almost 70x, reducing the duration of the test phase of a synthetic application from 24 hours to less than 30 minutes. The solution does not require any source-code modification, hides the grid complexity from the user and provides a cost-effectiveness improvement to the software testing experience.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging – Testing tools, Distributed debugging.

General Terms
Verification, Performance, Reliability

Keywords
Distributed Testing, Unit Testing, Computational Grid, JUnit

1. INTRODUCTION
Software testing is a fundamental part of system development. Examples of disasters caused by poorly tested software are widely available in the literature [1][2]. Therefore, there is an increasing demand for better support in the process of testing software.

Test cases are important artifacts since their execution usually provides useful information from a system under test. Automated testing adds power by helping gather and disseminate information quickly, to give programmers fast feedback.

Automated test suites are more beneficial in large software projects, which usually involve several development teams. In these situations, a test suite can be used by a development team to check if a software module provided by another team has the intended behavior, hiding the implementation details [4].

This way, large software projects usually have large test suites. There are industry reports showing that a complete regression test session of a 20,000 lines software could take as much as seven weeks of continuous execution [5].

Even software with fast running automated test suites may take a long time to be tested. That is the case of applications that need to be tested with varied environment configurations. Although software components may be extensively tested in the development environment before going into production, having the test run in a variety of environments can be very useful. This is because the production environment can be very different from the development environment, avoiding the detection of several failures. In these situations, the same test suite must be executed several times, multiplying the execution time by the number of target environments.

In the two situations above, the automated test suites may not accomplish the two main objectives of automated testing: quick detection of destabilizing changes in new builds and quick exposure of regression defects [3]. Therefore, we should deal with time consuming test suites.

Several different methods have been used to reduce the cost of time consuming test phases. Selection approaches are used to reduce the cost of testing by selecting a representative subset of the existing test suite [16][17]. Another approach is based in the prioritization of some tests among others, ordering the test suite and executing test cases with higher priority earlier than lower priority ones [17][18]. These approaches aim at revealing important defects in software earlier in the testing stage. Finally, there are distributed testing attempts. They hope to speed up the test stage by simultaneously executing a test suite over a set of machines [6][7][8]. However, until now, these solutions do not explore the parallelism and environment heterogeneity provided by computational grids [9][10], which, in our opinion, can be used to improve software testability.

In this paper, we discuss why grids are interesting testing execution platforms due to their heterogeneity and massive parallelism and describe the solution we have developed for testing software using the grid. The GridUnit tool aims at executing automated software tests on the Grid by distributing the execution of JUnit [11] test suites on a grid, without requiring modification in the application and hiding the grid complexity from the user.

2. TESTING ON THE GRID
The computational grid, or simply grid, is the computing and data management infrastructure that will provide the electronic underpinning for a global society in business, government, research, science and entertainment [9]. Grid computing offers a...
model for solving massive computational problems by making use of the unused resources (CPU cycles and/or disk storage) of large numbers of disparate, often desktop, computers treated as a virtual cluster embedded in a distributed telecommunications infrastructure. Grid computing's focus on the ability to support computation across administrative domains sets it apart from traditional computer clusters or traditional distributed computing.

The very high levels of parallelism provided by a grid can be used to speed up the test execution, increasing productivity and making the testing process of very time consuming test suites a less expensive task. Imagine as an example an open source project with a very time consuming test suite and several individual contributors distributed over the world. These contributors may decide to create a grid, combining the computational power of their own PCs, to speed up the execution of the project’s test suite.

The grid can also lower the costs of acquisition and maintenance of the test environment. For instance, a multi-national company, with several small software factories distributed over the world, working on different time zones, can create a grid to share its idle resources among its factories. By creating enterprise grids, companies do not need to buy additional machines to improve test speed.

Finally, by being a highly heterogeneous environment, the grid can be used to improve the reliability and coverage of the test suite. By using a grid as test execution platform, we can test software in several different environments, composed by varied software and hardware configurations and personalized to specific user profiles. This can practically eliminate test result contamination by specific development configurations and increase the test suite coverage since the application can be tested on a set of different target environments. Additionally, the grid provides resource virtualizations for its nodes, creating isolated playpens on which the tests can be executed. This bounded environment impedes that the execution of test alters the normal outcome of other tests, avoiding test case contamination.

3. THE GRIDUNIT DISTRIBUTED TESTING SOLUTION

Based on the previous observations, we decided to develop a tool to explore the intrinsic characteristics of grids for software testing. We called this tool GridUnit. It is a grid-based testing execution solution able to distribute the execution of JUnit [11] test suites in a grid with minimum user intervention. GridUnit is an open-source project, licensed under the GNU LGPL license terms, and can be freely downloaded from http://gridunit.sourceforge.net. GridUnit was developed on top of the OurGrid solution [12], although it can be easily adapted to use other grid flavors, such as Globus [13], for example.

OurGrid is an open, free-to-join, cooperative grid in which users donate their idle computational resources in exchange for accessing other users’ idle resources when needed. It uses the Network of Favors [14], a peer-to-peer technology that makes it in each user’s best interest to collaborate with the system by donating its idle resources. OurGrid leverages from the fact that people do not use their computers all the time. OurGrid is composed by workstations and some dedicated clusters.

OurGrid is in production since December 2004 and now encompasses around 500 machines in 30 sites distributed over Brazil and United States. A fresh snapshot of the running system can be seen at http://status.ourgrid.org.

3.1 GridUnit Architecture

GridUnit can be seen as an intermediary agent between the user/developer, who wants to run a JUnit test suite, and the computational resources needed to run the tests, in this case, the OurGrid. This brokering process involves four main tasks: i) creation of a job description from the JUnit TestSuite; ii) scheduling of the job’s tasks for execution on the grid; iii) monitoring of the execution; and iv) presentation of the execution results to the user.

Figure 1 summarizes the GridUnit high level architecture showing how it is constructed on top of OurGrid and how it uses JUnit components.

Figure 1: GridUnit High Level Architecture

The GridTestRunner is responsible for the creation of the job description from the TestSuite and for the scheduling of the job for execution on OurGrid.

A JUnit TestSuite specifies a set of independent TestCases that should be executed with no pre-determined order. An OurGrid job description specifies a parallel application composed by one or more independent tasks in which each task may be executed in any order and at any time. We call this kind of parallel application a Bag-of-Task. Due to this similarity of concerns, in fact a JUnit TestSuite is already a Bag-of-Tests (or a Bag-of-Tests), the conversion from a JUnit TestSuite to an OurGrid job description is straightforward. We simply create a job describing the TestSuite and a task for each TestCase in the TestSuite.

After the conversion, the GridTestRunner schedules the execution of the job on the grid using the WQR [15] scheduling heuristic provided by OurGrid.

Another component of GridUnit, the GridTestMonitor, is responsible for monitoring the execution of the tests on OurGrid and for converting the data resulted from the execution into a JUnit TestResult. This is not a trivial task due to the nature of the grid. There is no predefined order in the execution of the tests, and therefore, any test can end at anytime. Moreover, the new set
of failures that can occur due to grid faults further complicates this task. So, the GridTestMonitor must be able to distinguish between test failures and grid failures in order to provide reliable results to the user.

The last component of GridUnit is its graphical user interface, showed in Figure 2. This interface presents to the user all information about the execution of tests as if they were being executed in the local machine. In fact, to the GUI there is no difference in remote or local execution at all.

Figure 2 shows an example of the execution of a test suite on OurGrid. In this example, the test suite is composed by 288 test cases. The status bar, in the lower corner of the window, shows that at that moment 75% (or 217) of the 288 test cases were executed and seven of them have failed due to unsatisfied assertions. The progress bar is red because there are test cases that failed due to these unsatisfied assertions. It would be gray if some test case failed due to unanticipated errors (e. g. due to the grid platform) and green if no test case failed at all. The tree at the left corner represents the test case hierarchy and shows the status of the execution of each test case using tiny colored icons. It provides an overview of the execution process. The GridUnit GUI provides the same amount of detail about the execution of a test case as the traditional JUnit test runners along with some additional information about the environment where the tests were executed.

3.2 Main Features

The main features of GridUnit are those pointed by Kapfhammer [6] as important aspects that a test distribution tool must consider in order to improve the cost effectiveness of the testing process:

- **Transparent and Automatic Distribution**: The GridUnit considers each JUnit test as an independent task, scheduling its execution on the grid without any user intervention. Moreover, GridUnit does not require any modification in the application source code.

- **Test Case Contamination Avoidance**: Each test is executed using the resource virtualization provided by OurGrid preventing that the execution of a test alters the normal outcome of other tests.

- **Test Load Distribution**: The WQR scheduler provided by OurGrid achieves load distribution by allocating each test for execution in the first available grid machine. So, each grid machine receives a slice of the work proportional to its computational power.

- **Test Suite Integrity**: The default JUnit test runner runs each unit test as an independent task. For each test, it creates an instance of the TestCase class, calls the setUp() method, calls the testMethod(), calls the tearDown() method and then destroys the instance. The GridUnit reproduces the same behavior with the difference that each test is potentially executed in a different location on the grid.

- **Test Execution Control**: The GridTestRunner and GridTestMonitor modules provide a mechanism to build a centralized test execution and monitoring point. The GridUnit graphical user interface provides controls to start and stop the execution of the tests of a given test suite. It also monitors the execution of the tests and presents the result of the execution of each test as soon as it is available.

4. PRELIMINARY RESULTS

We consider that there are two classes of applications that would benefit from being tested with GridUnit: an application that needs to be tested faster and an application that needs to be tested in a variety of environments. Note that these classes overlap since testing software in a variety of environments may be also a time consuming task and, therefore, the user may desire to execute it faster.

Since faster execution is useful to both classes of applications we decided to start the development and validation of GridUnit focusing on speeding up test execution.

To have clear evidences of the benefits of GridUnit we conducted several experiments using it to distribute the execution of a very time consuming test suite. We create a synthetic application that has a test suite composed of 288 test cases, each one taking exactly 5 minutes to be executed. The overall run time is 24 hours when executed using the normal JUnit test runners.

![Figure 2: GridUnit Graphical User Interface](Image 274x301 to 558x522)

It is important to note that typical JUnit test cases can take much less than 5 minutes. In such situations, the network overhead caused by file transferences can be considerable. GridUnit solves this problem by allowing the distribution of clusters of test cases. The user can specify, for example, that her test suite should be divided into clusters of 10 test cases each. Since the system under test files need to be transferred only a single time per cluster this dramatically reduces the network overhead.

We conducted 162 experiments in a period of 15 days. This was necessary to alleviate the effects of network fluctuations during the testing process since OurGrid is composed of nodes geographically dispersed over Brazil and United States, connected through the Internet.

The application under test has a file size of approximately 5 MB. This was the medium of the sizes of the five most popular Sourceforge projects in 01/09/2005. File size is worth mentioning since the tests are executed in remote grid machines, GridUnit...
must transfer all the application files prior to the execution, and therefore, there is an overhead involved in this file transfer phase.

Figure 3 presents the evaluation results. As can be seen, approximately 25% of the experiments took less than 30 minutes to be concluded, 85% of the experiments took less than one hour and less than 8% of the experiments took more than 2 hour to be concluded. The combined time of all experiments was 152 hours. This is only 3.9% of the 3888 hours that a normal JUnit test runner would take to run 162 times the same test suite.

The fastest execution observed was carried in approximately 20 minutes using 111 grid machines (speed-up of 71.11x) while the worst execution took about 9 hours using 29 grid machines (speed-up of 2.68x).

![Figure 3: GridUnit Performance Evaluation](image)

5. CONCLUSIONS
We have showed that the computational power provided by Computational Grids can be used to parallelize the execution of tests, speeding up the process of testing software. Additionally, we discussed how grids can bring the software testing process a step closer to the software production environment.

We created an open source Java-based tool, called GridUnit, able to distribute the execution of a JUnit test suite over a grid without requiring any source-code modification, providing a cost-effective improvement to the testing processes.

Our current work is focused on improve GridUnit to add support for specifying and deploying test scenarios in order to better explore the environment heterogeneity of the grids. We are currently studying techniques to specify test scenarios and metrics to quantify how the test coverage is influenced by running the tests on the grid.

ACKNOWLEDGMENTS
This work has been partially developed in collaboration with HP Brazil R&D. Authors would also like to thank the financial support from CNPq/Brazil and the helpful comments and suggestions from the anonymous reviewers.

REFERENCES