Parallel Testing of Distributed Software

Alexey Lastovetsky

Department of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland

E-mail: Alexey.Lastovetsky@ucd.ie

Abstract

The paper presents the experience of the use of parallel computing technologies to accelerate the testing of a complex distributed programming system such as Orbix 3, which is IONA's implementation of the CORBA 2.1 standard. The design and implementation of the parallel testing system are described in detail. Experimental results proving the high efficiency of the system are given.

1. Introduction

Software testing is a very labour-intensive and hence very expensive process. It can account for 50% of the total cost of software development [1-3]. Software testing is also a very costly part of software maintenance in terms of contribution in the total time of the process of maintenance. If the process of testing could be accelerated, significant reductions in the cost of software development and software maintenance could be achieved.

In this paper, we present a case study demonstrating the use of parallel computing for acceleration of the testing of a complex distributed programming system such as Orbix 3 [4], which is IONA's implementation of the CORBA 2.1 standard [5].

Orbix 3 is a distributed programming system used by thousands corporate users around the world. It is an axiom that any software system has bugs. The wider and more intensive is the usage of the software system, the more bugs are exposed during its exploitation. Orbix 3 is not an exception to the rule. Every day its users report new bugs affecting the functionality or performance of the Orbix 3 software. A dedicated team of software engineers is constantly working on the bugs and making appropriate changes in the Orbix 3 code.

The maintenance process includes running an Orbix 3 test suite before and after any changes made in the Orbix 3 source code in order to:

- See if the bug has been fixed;
- Check that the changes themselves do not introduce new bugs into the software.

The test suite consists of many hundreds of test cases. Each fixed bug results in one more test case added to the test suite. This test case should test the problem associated with the bug and demonstrate that the problem has been solved. Thus the number of test cases in the test suite is constantly growing.

The serial execution of a test suite on a single machine might take from 9 to 21 hours depending on the particular machine and its workload. The test suite must be run against at least three major platforms. For each platform the test suite must be run at least

twice, namely before and after the corresponding changes are made in the Orbix 3 source code. Thus, on average, the best time for running a test suite is 90 hours per bug. Often, however, it takes longer. For example, if a bug is reported in some minor platform, the test suite should be run against both all major platforms and the minor platform. If the bug has enough complexity, the very first solution of the problem may introduce new bugs, and hence more than one solution will have to be tested during the work on the bug.

In terms of time, serial running of the test suite is the most expensive stage of the maintenance process. So its acceleration could significantly improve the overall performance of the maintenance team. Since the local network of computers available to the maintenance team includes more than one machine for each major platform and most of the machines are multiprocessor workstations, parallel execution of the test suite seems to be a natural way to speed up its running.

2. Parallel execution of the Orbix test suite on a cluster of multiprocessor workstations

As all major platforms, against which Orbix 3 should be tested, are Unix clones, an immediate idea is to use the GNU `make` utility for parallel execution of different test cases of the test suite. On Unix platforms, the `'-j'` option tells `make` to execute many jobs simultaneously. If the `'-j'` option is followed by an integer, this is the number of jobs to execute at once. If this number is equal to the number of available processors, there will be as many parallel streams of jobs as processors. As the utility assigns jobs to parallel streams dynamically, the load of the processors will be naturally balanced.

This simple approach has several restrictions. One is that it can only parallelize the execution of a set of jobs on a single multiprocessor machine. Another restriction is that if some jobs in the set are not fully independent, the straightforward parallelization may not guarantee their proper execution. For example, a number of jobs may share the same resources (processes, data bases, etc.), whose state they both change and depend on in their behaviour. Such jobs should not be executed simultaneously, but the GNU `make` utility provides no direct way to specify that constraint.

A typical test case from the Orbix 3 test suite builds and executes a distributed application. It normally includes the following steps:

- Building executables of the server(s) and clients of the distributed application.
- Running the application.
- Analysing the results and generating a report. The report says whether the test case passed or failed, and includes the start time and end time of its execution.

On completion of the execution of the test suite, all individual reports produced by the test cases are summarised into a final report.

During the serial running of the test suite on a single computer, its test cases share the following resources:

- Basic system software such as compilers, interpreters, loaders, utilities, libraries, etc.

- An Orbix daemon, through which servers and clients of Orbix distributed applications interact with one other. The daemon is started up once, before the test suite starts running.

- An interface repository, which stores all necessary information about server interfaces. This information can be retrieved by clients to construct and issue requests for invoking operations on servers at run time.

What happens if multiple test cases are executed simultaneously on the same computer? Can the sharing of the above resources cause unwanted changes in their behaviour?

The basic system software should cause no problem. Each test case just uses its own copy of any compiler, interpreter, loader, utility, or static (archive) library. As for dynamic shared libraries, their simultaneous use by multiple test cases should also cause no problem. This is simply because a dynamic shared library is by definition a library whose code can be safely shared by multiple, concurrently running programs so that the programs share exactly one physical copy of the library code, and do not require their own copies of that code.

There further should be no problem with sharing one physical copy of the Orbix daemon by multiple concurrently running distributed applications. This is because that sharing is just one of the core intrinsic features of the Orbix daemon. Moreover, in terms of testing, simultaneous execution of multiple distributed applications is even more desirable than their serial execution as it provides more realistic environment for functioning Orbix software.

Problems may occur if multiple concurrently running test cases share the same interface repository. In order to specify the problems, let us briefly outline how interfaces and interface repositories may be used in the Orbix 3 test suite.

In order to stress object orientation of the CORBA distributed programming technology, server components of CORBA-based distributed applications are called server objects or simply *objects*. The CORBA Interface Definition Language (IDL) permits interfaces to objects to be defined independent of an objects implementation. After defining an interface in IDL, the interface definition is used as input to an IDL compiler, which produces output that can be compiled and linked with an object implementation and its clients.

CORBA supports clients making requests to objects. The requests consist of an operation, a target object, zero or more parameters and an optional request context. A request causes a service to be performed on behalf of a client, and results of executing the request returned to the client. If an abnormal condition occurs during execution of the request, the exception is returned.

Interfaces can be used either statically or dynamically. An interface is statically bound to an object when the name of the object it is accessing is known at compile time. In this case, the IDL compiler generates the necessary output to compile and link to the object at compile time. In addition, clients that need to discover an object at run time and construct a request dynamically, can use the Dynamic Invocation Interface (DII). The DII is supported by an interface repository, which is defined as part of CORBA. By accessing information in the interface repository, a client can retrieve all of the information necessary about an objects interface to construct and issue a request at run time.

In Orbix 3 an interface repository is implemented as a CORBA server object. As a

CORBA server, it is registered with an Orbix daemon under the name IFR. Usually it is registered as an automatically launched shared server. This means that the IFR will be launched once by the Orbix daemon on the very first request for its services, and will be shared by all clients associated with this Orbix daemon.

Orbix 3 provides 3 utilities to access the IFR. Command `putidl` allows the users to add a set of IDL definitions to the IFR. This command takes the name of an IDL file as an argument. All IDL definitions within that file are added to the repository. Command `rmidl` removes from the repository all IDL definitions given as its arguments. Command `readifr` allows the user to output IDL definitions currently stored in the repository. Note that the IFR is a persistent object maintaining its state, i.e., the contents of the interface repository, even though the server process, in which it resides, is terminated and relaunched.

The Orbix 3 test suite contains a number of test cases dealing with the IFR, including:

- Test cases checking functionality of the IFR utilities,
- Test cases checking functionality of the DII, and
- Test cases just using the IFR not written to test any IFR-related feature.

A test case, which checks functionality of one or other IFR utility, typically performs one or more IFR utilities, and compares the resulting contents of the interface repository with some expected contents. Obviously, it is not safe to simultaneously execute more than one such test case. IDL definitions, added to the IFR by one test case, may be deleted by the other and vice versa, resulting in the contents equally unexpected by all the concurrently running test cases.

In general, it is unsafe to concurrently run multiple test cases dealing with the IFR if at least one of them changes the contents of the interface repository. In this case, other test cases will encounter unspecified and occasional change of the state of the IFR, which typically results in their failure.

As all test cases dealing with the IFR do change its contents, we can conclude that in order to guarantee their expected behaviour, execution of all those test cases should be serialised.

One simple way to serialise access to the interface repository by concurrently running test cases is via a mechanism for mutual exclusion. Such a mechanism acts as a lock protecting access to the IFR. Only one test case can lock the mechanism at any given time getting an exclusive right to access the IFR. Other competing test cases will wait until this test case unlocks the mechanism.

This approach automatically serialises test cases that should not be executed concurrently. Unfortunately, it cannot guarantee the best execution time if the test suite is running in parallel on a multiprocessor computer. Let us run the test suite on a p-processor Unix workstation ($p > 1$) so that we have p parallel streams of running test cases. Let n and m be the total number of test cases and the number of IFR-related test cases respectively. Assume that all IFR-related test cases are distributed evenly across the streams. Assume also that in each stream IFR-related test cases go first. Then, until all IFR-related test cases are completed, there will be only one running test case at any given time. Correspondingly, the total execution time will be

$$t \approx \sum_{i=1}^{m} t_i + \frac{\sum_{i=m+1}^{n} t_i}{p},$$

where t_i is the execution time of the i-th test case (we assume that first m test cases are IFR-related). At the same time, if

$$\sum_{i=1}^{m} t_i \leq \frac{\sum_{i=m+1}^{n} t_i}{p-1},$$

then exactly p test cases can be executed in parallel at any given time, and the optimal total execution time will be

$$t_{opt} \approx \frac{\sum_{i=1}^{n} t_i}{p}.$$

Another disadvantage of this approach is that it is hard to extend it from a single multiprocessor workstation to a cluster of workstations.

We shall present another approach to parallel execution of the Orbix 3 test suite, which is oriented on a cluster of Unix workstations and yet suitable for a single multiprocessor Unix workstation as a particular case of such a cluster.

The idea is to use a parallel program running on the instrumental cluster of workstations and working as a test manager. This program will:

- Accept a test suite as an input parameter. Individual test cases in the test suite are considered as atomic jobs to be scheduled for execution among available physical processors.
- Automatically partition the test suite at run time into as many parallel streams of test cases as physical processors available. This partitioning is performed so that:
    - Balance the load of physical processors based on the actual relative speed of workstations and the average execution time of each test case.
    - Exclude parallel execution of test cases using the same interface repository. It is assumed that each workstation of the instrumental cluster will run its own Orbix daemon, and its own interface repository registered with the Orbix daemon.
- Launch all the streams in parallel so that each workstation runs as many streams as it has processors.
- Wait until all streams of test cases complete.
- Collect local reports from the streams and produce a final report on the parallel execution of the test suite.

The test manager is implemented in the mpC language [6-8]. Therefore, it uses the

relevant mpC means to detect the number of physical processors and their actual relative speed at run time.

The average execution time of each test case is calculated based on information about the test suite, which is stored in a file accepted by the test manager as an input parameter. This file has one record for each test case. The record includes the name of the test case, the total number of its runs, and the total execution time of the runs. A simple `Perl` script is used to generate and update this file by processing a final report on the execution of the test suite. Recall that such a report includes the start and end time for each test case.

Another input parameter of the test manager is a file storing information about restrictions on parallel execution of test cases. In general, the file includes a number of separated lists of test cases. Each list represents a group of test cases that should not be concurrently running on the same workstation. In our case, the file includes only one list of all test cases using the interface repository. As each workstation of the instrumental cluster runs its own Orbix daemon and its own interface repository, the restriction is equivalent to the restriction that prohibits concurrently running multiple test cases using the same interface repository.

So, the algorithm of partitioning of the test suite for parallel execution on *p* physical processors can be summarised as follows:

- It is assumed that the test manager consists of *p* parallel processes, and each workstation runs as many processes as it has physical processors. Each process manages a stream of serially executed test cases. It launches the test cases and produces a local report on their completion.

- First, the processes detect the total number of workstations, *nc*, and divide themselves into *nc* non-intersecting groups, so that each group consists of processes running on the same workstation. In other words, the processes recognise the structure of the instrumental cluster of workstations and their mapping to the cluster.

- Then, each group of processes selects a lead. Each group of test cases, which should not be simultaneously executed on the same workstation, is distributed among the lead processes as follows. At each step, next test case is assigned to a lead process, which minimises the ratio

$$\frac{\sum_i t_i + t}{s},$$

  where t_i are the average execution times of the test cases, which have been assigned to the lead process, t is the average execution time of the scheduled test case, and s is the relative speed of this lead process. The motivation behind this heuristic algorithm is the following.

  - Intuitively, the average execution time of a test case characterizes the amount of work to be performed during the execution of the test case. This is particularly true if the average execution time is calculated based on the statistics collected from different serial runs of the test suite.

  - At each step of the algorithm, by minimizing the above ratio we try

and assign next test case to such a lead process that will execute all test cases, which have been already assigned to the process, plus the newly assigned test case faster than any other lead process. This way we try and keep the workload of the corresponding processors balanced. In the end, the amount of work to be performed by each lead process (expressed by the total execution time of all test cases assigned to the process) will be approximately proportional to the relative speed of the process.

- Then, the remaining test cases are distributed among all p processes as follows. At each step, next test case is assigned to a process, which minimises the ratio

$$\frac{\sum_i t_i + t}{s},$$

where t_i are the average execution time of test cases, which have been assigned to the abstract processor, t is the average execution time of the scheduled test case, and s is the relative speed of this process. The motivation behind that heuristic algorithm is similar to that of the algorithm of distribution of test cases over the lead processes.

As shown in [8], the above algorithm may produce a more accurately balanced distribution if the test cases are preliminary re-ordered in the decreasing order of their average execution times. This modification exploits the obvious observation that the smaller are things, the easier they can be evenly distributed. Hence, bigger things should be distributed under weaker constraints than smaller ones. For example, if you want to distribute a number of balls of different size over a number of baskets of different size, you better start from the biggest ball and put it into the biggest basket; then put the second biggest ball into the basket having the biggest free space and so on. This algorithm keeps balance between ball sizes and free basket space and guarantees that if at some step you don't have enough space for the next ball, it simply means that there is no way to put all the balls in the baskets. In our case, we decided not to use this modification because the test suite consisted of a very large number of relatively small (in terms of the execution time) test cases.

After the test suite is partitioned, each process generates a number of shell scripts and executes them. The scripts manage serial execution of test cases assigned to the process.

Note that the test manager is launched from a wrapper script, which:

- Initialises the mpC programming system on the instrumental cluster of workstations;
- Launches the parallel test manager on the cluster;
- Waits until the test manager completes its work;
- Merges reports produced by parallel streams into a single final report.

3. Experimental results

A representative series of experiments were conducted with the system for parallel

testing of the Orbix 3 software in the end of 2001. At that time, the total number of test cases in the Orbix 3 test suite was 588. About 15% of the test cases were dealing with IFR. We experimented with Solaris, HP/UX and IRIX shared memory mutliprocessor workstations. Specifications of the Solaris workstations used in the experiments are given in Table 1.

The experiments showed that the system significantly accelerates the execution of the Orbix 3 test suite. The acceleration shown on single multiprocessor computers was close to the ideal one ranging from 3.8 to 3.9 for Solaris 4-processor workstations from Table 1.

Parallel execution of the test suite on a cluster of two 4-processor workstations **Comp2** and **Comp3** provided speedup that varied from 6.8 to 7.7 depending on the additional external workload of the computers (the computers are used as servers by the Orbix 3 maintenance team). In terms of wall time, this means 1 hour 41 minutes instead of 12 hours 52 minutes (in the best case of the 7.7 fold speedup). Thus, the programmer can run the test suite up to 5 times a day and fully control four of the runs instead of running it only once a day and being not able to control in average more than half test cases.

Table 1. Specifications of computers used in experiments

Machine	Architecture	Processors	System clock frequency (MHz)	Main Memory (MB)	Cache (MB)
Comp1	SunOS 5.8 sun4u, Sun Enterprise 450	4 X UltraSPARC-II 480MHz	96	4096	8
Comp2	SunOS 5.6 sun4u sparc SUNW, Ultra-4, Sun Enterprise 450	4 X UltraSPARC-II 400MHz	100	2560	4
Comp3	SunOS 5.7 sun4u, Sun Enterprise 450	4 X UltraSPARC-II 400MHz	100	2048	4

4. Related work

Parallel technologies are used to accelerate the testing of computer hardware [9], [10]. At the same time, to the best of the author's knowledge, there is a single paper by Starkloff [11] which advocates the application of parallel and distributed technologies to testing of computer software systems as well. The paper summarises some general advantages of this approach and briefly reports on a multithreaded tester that can be used to run several independent test sequences in parallel.

A test framework for CORBA component model-based software systems addressing the problem of dependencies between separately developed components is presented in [12].

5. Conclusions

In this paper, we have demonstrated that parallel computing technologies can be

applied not only to solution of scientific problems. They can be efficiently used in software engineering practice to optimise the maintenance process.

The experience of integration of the mpC-based technology of heterogeneous parallel computing and the CORBA-based technology of distributed computing has proved that the integration can be easily made and does not require changes in the combined technologies. It appears that parallel computing technologies and distributed computing technologies are quite ready to work together, but more practical experience of their integration for solution of real-life problems is needed.

6. References

[1] D.S.Alberts, The economics of software quality assurance, *AFIPS Conf. Proc: !976 National Computer Conference*, vol. 45, AFIPS Press, Montvale, pp.433-442, 1976.

[2] R.DeMillo, W.M.McCracken, R.J.Martin, and J.F.Passafiume, *Software Testing and Evaluation*, The Benjamin/Cummings Publishing Company, 1987.

[3] G.J.Myers, *The Art of Software Testing*, John Wiley, New York, 1979.

[4] Sean Baker, *CORBA Distributed Objects – Using Orbix*, ACM Press, Adison Wesley, 1997.

[5] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.1, 1997.

[6] D.Arapov, A.Kalinov, A.Lastovetsky, and I.Ledovskih, A Language Approach to High Performance Computing on Heterogeneous Networks, *Parallel and Distributed Computing Practices*, 2(3), pp.87-96, 1999.

[7] A.Lastovetsky, D.Arapov, A.Kalinov, and I.Ledovskih, A Parallel Language and Its Programming System for Heterogeneous Networks, *Concurrency: Practice and Experience*, 12(13), pp.1317-1343, 2000.

[8] A.Lastovetsky, Adaptive Parallel Computing on Heterogeneous Networks with mpC, *Parallel Computing*, 28(10), pp.1369-1407, 2002.

[9] F.Karimi, S.Irrinki, T.Crosby, N.Park, and F.Lombardi, Parallel Testing of Multi-Port Static Random Access Memories, *Microelectronics Journal*, 34(1), pp.3-21, 2003.

[10] W.W.Kim, Parallel Testing Method by Partitioning Circuit Based on the Exhaustive Test, *Lecture Notes in Computer Science*, 3044, pp.262-271, 2004.

[11] E.Starkloff, Designing a Parallel, Distributed Test System, *IEEE Aerospace and Electronic Systems Magazine*, 16(6), pp.3-6, 2001.

[12] H.J.Batteram, and W.A.Romjin, A Test Framework for CORBA Component Model-Based Software Systems, *Bell Labs Technical Journal*, 8(3) , pp.15-29, 2003.