CST IB Group Project Lima

Testing Report

Overview of Testing Performed

Testing for a probabilistic system, one that in its design will run for a long time before giving its results and then over a significant number of computers was always going to be a challenge. Various invariants and relations can be checked through the process which allows a certain amount of built-in testing during the proceedings so that faults can be detected earlier rather than later, but these do not provide full testing. Testing of individual modules was essential to ensure that components used to build the whole system functioned correctly according to their specifications so that when the system was integrated, failures should have been due to the integration failures and incompatibilities between modules. However, whilst this is helpful, testing of the whole system running through some factorisations was essential in order to ascertain whether the system as a whole was able to perform correctly. This is especially true in the cases where there is a lot of interoperability between classes so the module granularity is quite large.

There were two main things that needed to be ensured by testing. Firstly, the aim of the project is to produce a system which factorises large composite integers. Hence if the system is to be tested properly, tests must ensure that this is what it does. Correct factors must be drawn out of the composites in the running of the program and these should be prime. Fortunately, testing whether test resulting factors are valid factors is trivial. Checking whether they are prime or not, however, is not. Unfortunately, deterministic primality checking is very costly, and although it can be done in polynomial time[1], the constants are still large making it infeasible. Therefore probabilistic methods again had to be used for testing, in this case using Miller-Rabin. Such testing was already built in to the system however to check when to stop factoring and so the facility existed for final checking. Furthermore, by using in our tests composites generated from known primes, we know what the output should be with absolute certainty and hence have a good check which can be performed.

The other thing that was specified for the project was that it should include “mechanisms that allow the relevant computations to be spread over a number of machines”[2]. Therefore it is an essential part of the testing to verify that the computation is spread over the network, that it can cope with any network problems and that communication works properly. In our implementation, we have a clearly defined separate distribution module and hence this is testable quite independently from the factorisation using simple custom workunits, work unit generators and a very simple client. However, it must naturally also be tested in conjunction with the factorisation modules and to run ECM and Quadratic Sieve over the network to a large number of clients.

Review Of Tests

Module level

Testing at module level was easier in some cases than others. Generally, the aim was to take the module in isolation and run it with as little of our own code as possible (except for verbose debugging information) to perform an automatic verification of its correct functionality on some given test data.

[1] http://www.cse.iitk.ac.in/news/primality.html
[2] Project briefing booklet, Michaelmas 2002, p12

The distribution server (DServer) was tested early on with a test client which deliberately broke the rules (not submitting work units, repeat submissions etc) in order to simulate failures in the actual system. On the other side it was provided with TestWorkUnits by a special source and sink which set up some TestWorkUnits in a known state and then validated that the work unit had run correctly and only once upon return. As with other module testing, by outputting information about the tests to stdout, this could be parsed by scripts to produce a PASS/FAIL indicator for particular tests.

Each of the algorithms were tested as separate modules, where possible. Trial Division and Pollard Rho are able to stand alone. Sample inputs were generated randomly from some known primes these were then passed into the algorithms. The results could then be compared with the known source primes which were stored internally in the same way as the Factorisation used. Elliptic Curve Method (ECM) and Quadratic Sieve both are distributed systems and depend on the distribution system to run. Therefore the could only be tested in conjunction with this and hence after the distribution system has been itself tested.

Various utility classes were simply tested by feeding in some input values and ensuring that the data retrieved out was correct as well as attempting to pass in faulty data and ensuring that it coped with this situation. For example, with lima.mathematical.Factor, as well as creating valid Factor objects, invalid ones were created too in order to ensure that it would fail in such circumstances.

The GUI was a late addition and was tested by human interaction and ensuring that it performed as expected over a period of time, receiving and submitting work units, connecting and disconnecting at difference stages and managing to cope if it lost (hopefully temporarily) connection with the server.

System level

The same tests which were able to test the different algorithms could also call Lima, since Lima extends Factorisation in the same manner as the others. Since Lima performs the entire operation, this provided a testbed for testing the system. However, since it needed to be tested across a network of computers only a couple of computers could be started automatically as clients (via ssh), this generally had to be run with someone setting up manually lots of clients connecting to it. However, this did mean that at times up to 60 clients were talking with the server, pushing the distribution server very hard. Outputs during these tests from clients and servers were sent to log files so that they could be reviewed in the case of failure.

Test Results

Individual Module Results

DServer			
	Reject duplicates from client		PASS
	Run just once		PASS
	Returning all units		PASS
	Work Unit fidelity		PASS
	Overall distribution		PASS
Mathematical.Factorisation			
	TrialDivision		PASS
	Pollard Rho		PASS
	ECM		PASS
		Curve Production	PASS
		Factor Searching	PASS
		Finds Factors	PASS
	Quadratic Sieve		
		Factor Base	PASS
		Initialisations	PASS
		Sieving	PASS
		Gaussian Ellimination	PASS
		Finds Factors	FAIL
	Factor		

		Constructor	PASS
		Validate exponent	PASS
		Validate base	PASS
		Default Exponent	PASS
		Default Bass	PASS
	FactorSet		PASS
	Utilities		
		BigIntegerFunctions	PASS
Validation			
	Validate objects		PASS
	Validate fail for nulls		PASS

System Results

Start Lima	PASS
Run TD	PASS
Run Pollard Rho	PASS
Start ECM	PASS
Detect DServer	PASS
Clients process ECM work	PASS
DServer handles ECM work in/out	PASS
ECM gets all data back	PASS
ECM finds factors	PASS
ECM terminates	PASS
Start QS	PASS
QS Initialisation	PASS
Detects DServer	PASS
Clients process QS work	PASS
DServer handles QS work in/out	PASS
QS gets all data back	PASS
QS trial division on returned WUs	PASS
QS removes singletons	PASS
QS matrix reduction	PASS
QS Gaussian Ellimination	PASS
QS Find Factors	FAIL
QS Terminates	FAIL

Problems Encountered and Lessons Learned

There were numerous difficulties encountered along the way with the testing. By far the thing which made testing hardest was exactly what anybody would expect - bugs. If it had all worked first time, then testing would have been much easier! However, given that a system never works first time, there were some other interesting problems encountered.

On a module level, the mathematical functions should not have been especially difficult to test, given that a number is put in and we just have to test the numbers which come out. However, we soon realised that we did not have a good structure for storing generated composites and their associated prime factors in order to compare them with the given answers, so we were forced to implement a much more similar (and complex) structure to make the testing easier. Previously we'd also had the issue of how we store around 100 different primes and retrieve them. In the end we formed a matrix of primes, which meant we had to construct a carefully formatted Java file containing the primes (now PrimesData.java) which was rather time consuming.

The distribution server was not too bad for testing, especially as Phil who wrote the distribution server outlined the means by which we could test it. There were some problems with work units and job matching in the hash table, but these related to a problem with the test work unit rather than the distribution server. It was however frustrating, as it took some time to resolve and was an effect of the distribution server depending upon something not included in the specifications for work units. The lesson clearly here is to ensure that the specifications are always carefully updated if changes are made or additional items are depended upon.

Mathematical utilities were generally fine, except for a small bug in isPrime. This called Java's BigInteger.isProbablePrime and whilst it worked fine on the developers' computers, it failed on the PWF. After some further experimentation, it materialised that there is a bug in Java 1.4.0 (although it does not appear to be documented anywhere on the web) which causes isProbablePrime to fail if you specify the maximum value that you possibly can for the level of certainty. If however you set it to that minus 1, it worked fine. The developers all have 1.4.1 in which it appears to be fixed.

The real fun came with system testing. It proved to be very problematic with the factorisations taking so long as it sometimes would fail after 5 minutes and sometimes after 2 or 3 hours. Therefore there had to be someone there checking for failures and watching the logs for problems. To a certain degree the time could also be used for other things such as supervision work and so it was not completely time lost, but it did prove frustrating and slow work. Many of the bugs were obscure, usually throwing fairly generic RMI exceptions on the client masking the real problem. We generally found that to get a proper picture of a fault, the logs of each client needed to be looked at, as well as the server and Lima logs. Then the point of failure could be identified. Furthermore there were occasions when new bugs were introduced by "fixes" to other ones which hindered progress even more. The moral there is clearly not only to carefully review bug-fix code, but also to look there immediately something goes wrong subsequently.

Finally, with the applet client, Jonathan as well as writing it did a lot of the testing and made it very robust. We did however encounter a very odd problem in that it would always throw a SecurityException unless it was run off localhost. After some considerable time spent web-searching and spec reading, we found that applets (unlike their application counterparts) can only call RMI functions via RMI on the machine from which the web page holding the applet was hosted. If we had read the applet security document from Sun, then we would have been aware of this. Hence the lesson to learn here should be to carefully read the relevant information document(s) from Sun when a problem occurs. This also posed the problem that we needed to run a webserver on the same computer as we wanted to run the distribution server on. This seemed to be problematic as we thought that we'd not be able to get one running on the PWF. However, we were given advice on a small web server which can run from a user home directory on a high port. This enabled us to continue using PWF computers as the servers which made things somewhat easier.

Conclusion

In conclusion testing managed to do a reasonable job of finding bugs, though far from perfect. It is noticeable that Quadratic Sieve above still fails at the end, and this was not picked up because it was previously failing at an earlier stage so it didn't make it this far to test before. There has generally been a very good relationship between testers and programmers and certainly the testers owe a large debt of gratitude to the programmers for the help they provided with the testing, in advice, writing some test code and assisting with the running of various tests. This I believe has been instrumental in making the project work.