SE 390 - Test Plan
Brain Simulator Parallelization

Prepared by
Team Spike
Greta Cutulenco - gcutulen
Robert Elder - relder
James Hudon - jbhudon
Artem Pasyechnyk - apasyech

Presented to
Dr. Paul Ward

University of Waterloo
Software Engineering
December 14, 2012

Introduction

This document outlines the test plan for our project, “Brain Simulation Parallelization”. We will detail our systematic approach to testing our eventual software system. Please note that the actual testing workflow is subject to change as we go along.

Initial Test Plan

Our testing workflow has a few key components: git, GitHub, TravisCI (https://travis-ci.org/) and accompanying build and test scripts. All members are collaborators on a repository hosted on GitHub. All committed code gets pushed to this repository to be shared with the other team members.

From the perspective of other project members, development will be test-driven. That is, all code committed to our shared remote branch is to be accompanied with relevant tests. The developer is responsible for writing tests for his code. Furthermore, these tests will primarily test for the acceptance criteria of the given feature or bug-fix but will also include integration tests. Lastly, all code pushed to our shared remote repository should pass all regression tests and the new acceptance tests.

To ensure an efficient yet safe workflow, two test suites will be maintained. One test suite will be run at every push to our GitHub repository. This suite will run on an instance of our code built from the latest push using TravisCI. This will be an automated process to ensure that the repository is safe to pull from. The second test suite is a subset of the first that will run every time a developer compiles the project. This will be a set of core tests that run fast and test essential functionality. These will ensure that developers do not share code that is totally broken. We will make these tests run automatically by including them in our build process.

Furthermore, our git workflow will potentially use topic and bug branches. Building and testing on topic branches is supported by TravisCI and will allow developers to test experimental features or fixes without having to merge their experiments into the main development branch.

One last measure to ensure that pushed features and fixes are correct will be improvised code reviews. Any developer who did not contribute to a commit is encouraged to review the patch and leave comments on GitHub. It is then the author’s responsibility to respond in

a reasonable amount of time and either fix the issue or explain why it is not an issue. Any disagreement between reviewee and reviewer is to be resolved by a second reviewer.

Fortunately, the nature of this project allows for rather simple unit tests. The output of all the processing that occurs will be a large matrix of numbers. In practice, some of the intermediary weights will be initialized with random numbers, and this can change the output slightly for each iteration. For the purposes of unit testing, we can simply seed the random number generator with a specific value and this should allow the results to become deterministic. To process a unit test, we will first compute the results of a test in a controlled environment that has the most trivial implementation possible. This result will be saved and used to perform a 'diff' against any subsequent trial run of the program in the future, when that seed value is used. This will allow us to catch the most critical type of errors which could affect the integrity of the results generated by the model.

Metrics

Our primary measure of success for testing is code coverage. We do not yet know the platform and language we will be using to build our solution (choosing the right language is an important part of this project), but we will use the appropriate tools to report code coverage by our tests.

A second measure of success is the speed of our tests. The faster our tests run, the easier it will be for us to accumulate hundreds of them and have them all run within a short amount of time. This is essential to minimizing friction in our workflow.

Another measure of success is qualitative. Do our tests cover all acceptance criteria? Are they deterministic? Are they easily maintainable?

Finally, do we pass the "joel test"? We will need to periodically go over certain items in the test to ensure that we are maintaining good habits.

The Joel test:

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?

10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?