ECSE 420 - Fall 2020
December 6th, 2020

Project Group 21

Project Report

Alexa Normandin (260803665)

Alexis Franche (260791358)

Maxime Cardinal (260802076)

Oliver Murphy (260799897)

Link to github project: https://github.com/MaximeCardinal/Test-Coverage-Optimization

Table of Contents

Instructions

Libraries

To accomplish the parallelization of code coverage and furthermore help a developper optimize its percentage of code executed by algorithmically computing the best matching combinations that will achieve the highest coverage and return to the user those very inputs we have used a multitude of libraries. An important note is that this program was built using Python Version 3.8 (Also tested on Python 3.9). The core of our project relies on the Coverage library that can be installed through pip using the command “pip install coverage” and installing version 5.3. The coverage library is used to pass a python file with inputs that you want to be injected into and returns information on the lines executed, the missed lines and the code coverage score. Another important library is the built-in multiprocessing library that we used to instantiate processes and do operations in parallel. The rest of the libraries used are utility libraries readily available in python that we used to help us accomplish our goal such as “sys” for input manipulation, “os” for executing operating system commands, “json” to manipulate the generated json data, “itertools” to generate input combinations and “subprocess” to execute command line operations.

Compilation / How to run program

To successfully execute the file test_coverage_optimization.py the user must be in his terminal at the same location as the file or add the file to environment variables to be executable anywhere. The program can be executed with the following command:

python test_coverage_optimization.py *path_to_test_program arg1 arg2 arg3 …*

Where *path_to_test_program* is the path to the program on which the optimization will be performed, and *arg1, arg2, arg3, …* the list of the types of arguments of the program on which the optimization will be performed. Possible values are: string, integer, float, boolean. It is also possible to specify a specific list of inputs rather than its type for an argument. The format is a comma separated list of the possible inputs. Ex: "3,4,5".

The program can also be run with the default set of input that we provide (Make sure that the file “default_test_program.py” is located in the same directory as “test_coverage_optimization.py”). with the following command:

python test_coverage_optimization.py

Note that if the test file takes in two inputs it is important to list two series of inputs to pass and our program will generate combinations of two inputs and for three inputs it needs three series of comma separated values and so on. In the end a report.txt will be generated containing all pertinent information such as the code coverage achieved, which combinations of inputs generate this best coverage, the total number of lines covered, which lines were executed, the total number of lines missed and which lines were not executed.

Problem

Testing is an important part of software development and it can be time consuming. Our goal is to develop a program that would facilitate testing by computing the optimal input set a tester should use to achieve the highest path coverage of a given program. However, achieving great path coverage can demand extensive trial and error for inputs to be generated and injected in the given programs. Thus, our goal is to parallelize the process of data injection and program testing in order to find inputs that will generate the greatest path coverage in the shortest amount of time possible.

Code Structure

The first input to our program is the path to the python file on which to run the test coverage. Next, the arguments to the program to test are passed. They can be specified as the type of argument to be passed (string, integer, float, boolean) or specific values can be passed for each argument as a comma separated list (e.g. 3,4,5). When the type of argument is passed, the program assigns a comprehensive pre-defined list of values of the given type. Once all we have the values for each input of the program to test, we generate all the possible input combinations. These combinations are run in parallel using the "coverage" library to determine the code coverage. Now, we have the path coverage of all the combinations. From this, we find the combination of input which has the largest code coverage. If this input does not have 100% code coverage, we find the lines which it could not cover. We find another input combination which complements the largest input combination coverage. That is, it contains the missing lines of the largest input combination coverage. This runs until we attain the base threshold of code coverage or if the added input combinations are poorly contributing to the combined code coverage (it is not worth adding more inputs because they barely increase the code coverage). The output contains the best input combinations to run and the code coverage they produce.

Design Process

Our design process evolved quite a lot during the creation of the final product. Our initial thought was to develop our tool in Python using PyCUDA. This went well at first. Python simplified our lives for things such as string manipulations and memory management. We also made use of various Python libraries to avoid having to implement existing functionality. For example, Python has a library called "itertools". This allowed us to find all the possible combinations of inputs with one function call. However, we started having issues when it was time to make kernel calls. Our lack of experience with the technology proved troublesome. After much debugging we concluded it was in our best interest to use CUDA with C since we had more experience with it. We kept the Python code and saved all essential information to a file the C program could read. Our intention was to call the C program from Python. After implementing the C program, we found out that we do not have access to the “coverage” library from the GPU. To use it we would need to implement its functionality manually. The “coverage” library was a key component of our product, so we were forced to look for a new solution. We found a library in Python called “multiprocessing”. This library allows us to parallelize processes by using our machine’s processors.

Using Python’s multiprocessing library was a design decision that we concluded would offer what we were looking for as opposed to the Python “threading” library. A major aspect to take into consideration while introducing parallelization in Python specifically is that we need to take into account the Global Interpreter Lock. Due to the Global Interpreter Lock, Python programs do not leverage a system’s specifications because they are bound to a single CPU core. The reason for the Global Interpreter Lock to be necessary is that Python is not thread safe and ensures data atomicity by preventing race conditions. Therefore, Python was designed to operate on a single core. Thus we have chosen to work with the multiprocessing library in order to bypass the Global Interpreter Lock and be able to use the entirety of the CPU cores. The way the multiprocessing works is that it gives each process its own interpreter as well as their own Global Interpreter lock. This information means that in Python threading in parallel is fundamentally impossible; it uses concurrent execution and context switching, hence the multiprocessing library was a better design choice to leverage true parallel execution on all the accessible CPU cores of the system.

Analysis

Decomposition Technique

The main workload of our program is to run the coverage program with all the generated combinations of inputs which grows a lot depending on the number of arguments of a program. As such, we distributed this work over the different processors of a machine. For instance, if a machine has 4 processors and the program generates a list of 100 combinations, each processor would work on computing the coverage of 25 combinations. We are therefore able to divide significantly the amount of work to be completed in parallel (bounded by the number of cores).

Maximum Possible Parallelism

The maximum possible parallelism that can be achieved in our program using a library such as multiprocessing is bounded by the number of cores on the system which the program is being executed on. Say a system has four CPU cores then true parallelization is achieved on these four cores where processes with their own address space are being executed at the same time. However, it is possible to spawn more processes than the number of cores available on the system. What will happen in this situation is that for one core there will be concurrent execution which although is not true parallelization will allow for a context switch between processes and enable better performance. Nonetheless, creating a lot of processes exceeding core count is a slippery slope because at a certain point having too many will become a source of bottleneck. In our case, the source of the bottleneck arises when doing file I/O because of the additional overhead needed for process switching when creating and deleting the JSON files containing the code coverage statistics.

Critical Path

The critical path of our program can be described as follows, with the branches representing the section that can be performed in parallel:

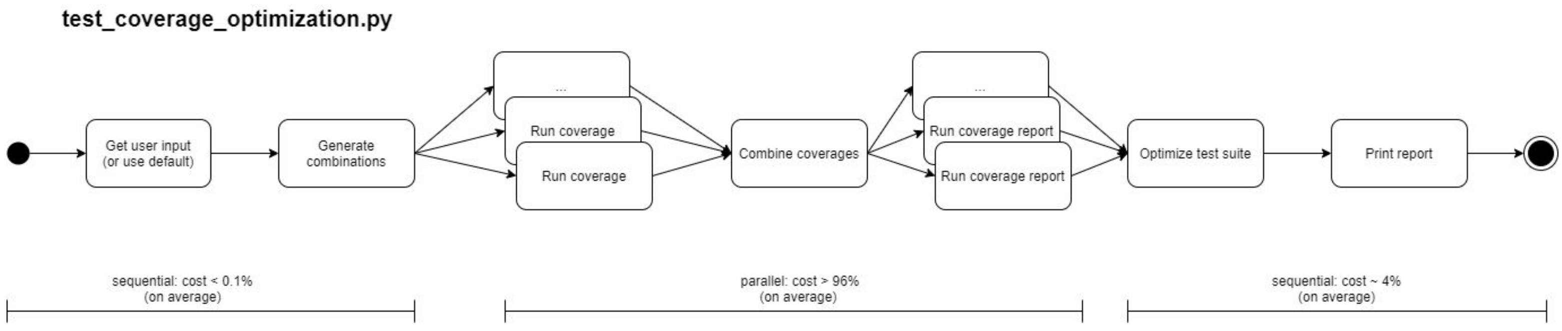
test_coverage_optimization.py
Get user input (or use default)
Generate combinations
...
Run coverage
Run coverage
Combine coverages
...
Run coverage report
Run coverage report
Optimize test suite
Print report
sequential: cost < 0.1% (on average)
parallel: cost > 96% (on average)
sequential: cost ~ 4% (on average)

Figure 1. Critical path and the cost of the 3 main sections of “test_coverage_optimization.py”

We calculated the runtime of the sequential parts of the code versus the parallel parts of the code. As can be seen in Figure 1. The parallel part of the code consumes, on average, 96% of the execution whereas, the sequential part of the code is responsible for 4% of the execution time.

Speedup

All the following tests were performed with default program inputs, which resulted in a set of 507 combinations. The first set of tests was performed on a machine with the following specifications:

Specification	Description
Processor	Intel(R) Core(™) i5-7300HQ CPU @ 2.50GHz 2.50 GHz
RAM	16.0 GB
Cores	Quad-core

Table 1. Machine 1 specifications.

Results:

Number of Processes	Execution Time (sec)
1	287.08
2	148.59
3	109.87
4	91.44
5	92.46
6	90.86
7	90.97
8	90.03

Table 2. Execution time of “test_coverage_optimization.py” on Machine 1 with default inputs.

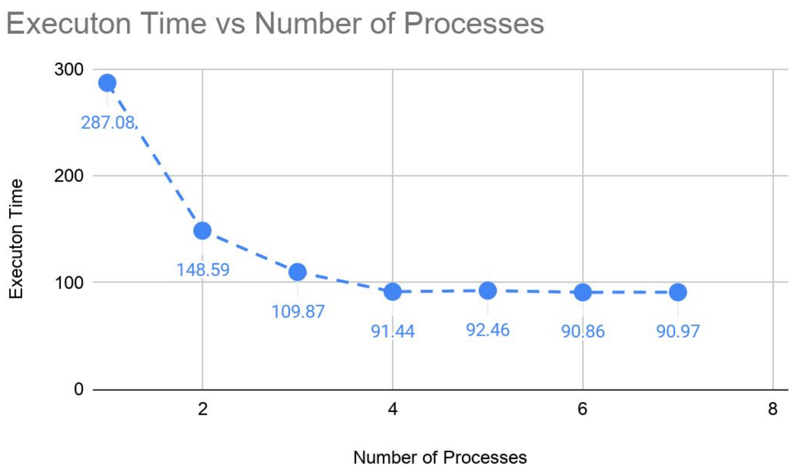

Figure 2. Speedup plot of the runtime of tests performed on Machine 1 with default inputs.

The second set of tests were run on the following machine, on the same file and with the same set of inputs which resulted in 507 combinations.

Specification	Description
Processor	Intel(R) Core(TM) I5-9600k @ 3.7 GHz
RAM	16.0 GB
Cores	6 Cores

Table 3. Machine 2 specifications

Results:

Number of Processes	Execution Time (sec)
1	180.29
2	94.67
3	70.04
4	54.79
5	50.21
6	47.98
7	49.34
8	48.72

Table 4. Execution time of "test_coverage_optimization.py" on Machine 2 with default inputs.

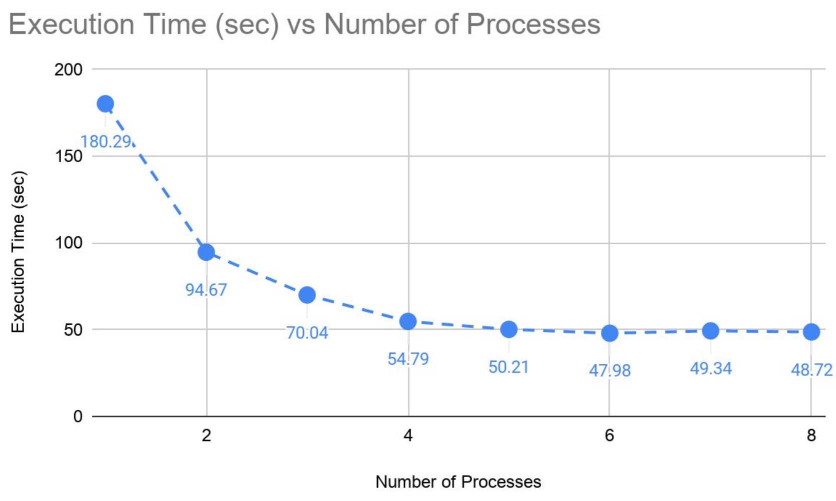

Figure 3. Speedup plot of the runtime of tests performed on Machine 2 with default inputs.

Explanation of results

As expected, when we run the programs with one core (sequentially), they produce the slowest time. When we increase the number of cores the execution times decrease until the program reaches the machine's number of cores (which is 4 for machine 1 and 6 for machine 2). At this point, cores will run in parallel, but processes on a given core run concurrently. This can increase or decrease performance. If a process must wait for something to happen it can be beneficial to run the process concurrently. However, concurrency adds additional overhead which is needed for process switching. Thus, it can slow down the execution time if the overhead becomes too large. We can see this phenomenon occur in the results above for 5 or more processes in machine 1 and 7 or more processes in machine 2. The execution time varies due to the overhead produced from running processes concurrently on each core.

Testing

To test our program we ran the same file with the same inputs on two different machines with two different numbers of cores, that is, 4 and 6. As we can see, our parallelization does work. Increasing the cores does decrease the execution time until we reach the machine's number of cores. Beyond this number, on each core running in parallel, there are concurrent processes running. This causes the runtime to vary. It can be slightly more or less efficient due to the overhead which occurs from the concurrency. Determining the optimal number of cores requires fine tuning,

To test for correctness, we started with small files with a small number of input combinations possible such that we were able to verify the programs output by manually looking at the code and figuring out which inputs should be output. We incrementally increased the size of the files and the number of input combinations until it was no longer viable to manually compute the output. Also, the library we use to determine the code coverage is a legitimate and popular tool, so its results can be trusted.

References

https://docs.python.org/3/library/multiprocessing.html

https://timber.io/blog/multiprocessing-vs-multithreading-in-python-what-you-need-to-know/

https://stackoverflow.com/questions/29089282/multiprocessing-more-processes-than-cpu-count

https://www.machinelearningplus.com/python/parallel-processing-python/#:~:text=In%20python%2C%20the%20multiprocessing%20module,in%20completely%20separate%20memory%20locations.

https://developer.nvidia.com/pycuda