Automation System for Batch Running Simulations in Parallel in OMNeT++

Anthony Medico
Department of Computer Science
Brock, University
St. Catharines, Ontario

Abstract

Intelligent Transportation Systems (ITS) is an increasingly important and practical area of research. Researchers who do work in this field often use simulation software to test solutions to the challenges that arise from mobile vehicles communicating. Running simulations can be a time consuming task, especially as the density of cars in the simulation increases. We introduce a batch running system to help automate much of the process, including three main benefits: streamline configuring batch runs, increase run time efficiency by processing the batch runs in parallel, and collecting and organizing the simulation results.

1 Introduction

Research in various fields of computer science sometimes requires the use of simulation software to perform the research task in a simulated environment. Depending on the complexity of the simulation system, these simulation runs can take a long time. To make things even more time intensive, simulations often need to be run multiple times to obtain an average, reducing outliers in the data affecting the results. The parameters of a simulation may also be changed and experimented with, creating the need for a very large number of runs. Batch running is the process of doing many runs at once and obtaining all the results at the end. Since these simulations are very resource intensive, the ability to perform batch runs in an efficient way will allow researchers to optimize their time, obtain results sooner, and therefore accomplish more work in less time. We set out to create a system that will automate much of the batch running process. The simulation software that we are working with is Veins, SUMO, and Simu5g, which are being run with OMNeT++. These simulations are for research in the field of Intelligent Transportation Systems (ITS). Various issues and areas of research in this field include Handover, Resource Management, and vehicle motion prediction. Simulations that are run with this software will test different solutions on efficiently handling these issues. Our software will automate batch runs as efficiently as possible, using various adjustable parameters, and collect and organize the results of each run.

2 Background

2.1 OMNeT++

OMNeT++ is a simulation framework and library used for building network simulators. It is based on the C++ programming language. It is highly flexible and modular, and is used primarily for academic research, or even in the industry, to create and model complex systems. OMNeT++ is also an IDE, based on Eclipse, and provides an interactive graphical runtime environment (Qtenv), or a command-line interface (Cmdenv). An important feature of OMNeT++ is extendibility using other frameworks and libraries, some of which are used in the simulations this program is automating, and are described below. [5]

2.2 INET

The INET Framework is an open-source model library for use in OMNeT++. It is intended for researchers and students who are working with communication networks by providing protocols, agents and other models. There is wide support for man different types of communication networks, such as wired, wireless, mobile, and ad hoc networks. It also contains Internet protocol models such as TCP, UDP, IPv4, and IPv6. There are other simulation frameworks that use INET as a base, and extend it to a more niche area, such as vehicular networks. [6]

2.3 Veins

Veins is another open-source framework, used for running vehicular network simulations. It extends OMNeT++ and SUMO and allows these two frameworks to connect and work together. OMNeT++ takes care of the network simulation, and SUMO does the road traffic simulation. With these two simulators tied together with Veins, research on the influence of intra-vehicular communication can modeled and then examined and analysed. [4]

2.4 SUMO

SUMO (Simulation of Urban Mobility) is an open-source traffic simulation suite. With SUMO you can control every aspect of the simulation, including the modeling of each individual vehicle, pedestrian, and public transportation. Traffic lights can also be modified, and vehicle communication is possible through connecting to network simulator such as OMNeT++. Simulated objects (vehicles, etc.) can be controlled during live simulations using the Traffic Control Interface (TraCI). [2]

2.5 Simu5G

Simu5G is a 5G New Radio simulator. It is built on the OMNeT++ and INET frameworks, and is used to simulate 5G networks, providing an environment for modelling and analyzing the performance of 5G networks. It implements the 5G NR protocol stack, supporting both the Standalone and Non-Standalone Architectures. Simu5G can also be used to develop your own new 5G protocols, or enhancements to existing ones. [3]

2.6 Python

Python is a general-purpose programming language, widely used in different domains, which include data analysis, artificial intelligence, web development, and automation scripting. The standard library is extensive and includes many modules for handling a wide variety of tasks. There is also a vast amount of third-party libraries and frameworks for Python, allowing it to be easily used in many different applications. [1]

2.7 Simulation Scenario

With advancing wireless technology and smart vehicles, the ability for vehicles to communicate with each other as well as cellular towers, allows for complex Vehicular Networks that present many research opportunities. The simulation software above is used by researches and industry professionals to test novel approaches to issues like Handover Management and Resource Allocation. Simulations include a Vehicular Network of a specified number of cars, using real-world datasets of traffic in real cities. In our testing, datasets are used representing a real-world map of the Cologne, Germany metropolitan area, which is a large area consisting of both highways and city roads. The following choices of number of cars are available in the simulations: (7, 100, 400, 700, 1000, 1300, 1600, 2200, 2500, and 2800). These numbers can be selected and used as the seed parameter in a simulation representing vehicle density. There are also a number cellular towers present - 10 in our testing. Throughout the simulation, data is collected containing various information including number of messages that have been sent between vehicles and towers. This information is logged during the simulation.

Simulations are complex, and the number of messages can grow exponentially as the density increases. Testing at various densities is important, and when running simulations with high densities the runtime of each simulation can also grow quickly. This creates a sub problem and additional

overhead to one's research - how to run simulations in an efficient manor to maximize productive use of one's time. The use of a streamlined batch running system can greatly assist with this.

3 System Design

The batch running system will need to perform the following tasks: running batch simulations based on the set parameters, manage and organize the results, parallelize the simulations to maximize performance. The programming language to be used in automating these tasks is Python, using modules from the standard library to implement each task.

3.1 Running Batch Simulations

The method chosen to run multiple simulations is to start a new subprocess for each simulation, with the Python script handling the number of repeats, rather than OMNeT++. The parameters for each batch run are seeds and repeats. The seeds represent density, i.e. number of cars in the simulation. The seeds can only be specific numbers as identified in the XML files with the vehicle simulation data. Repeats must be a positive integer, and will be used to create a range of runs from 1 up the repeat amount. With these parameters set, a list of tuples can be formed for each seed-run pair, to be used as parameters when beginning a new simulation.

3.2 Parallelizing the Simulations

The multiprocessing library in Python can be used to parallelize the simulations. This will be done by creating a pool of tasks, with each task being a call to a start_simulation(seed, run) function for each seed/run pair. This library starts each task in the order you create them, which gives control over which parallelization policy you would like to use. This system will implement 3 policies: smallest densities (seed) first, largest densities first, and bag of tasks (random). Each policy can be implemented by creating tuples of seed/run pairs and ordering them appropriately.

3.3 Managing and Organizing Results

The results from each simulation are in the form of the returned stdout and stderr from each subprocess. The system will first create a main results folder to contain the results from each batch run. Within this folder it will create a folder for the batch run using the current date and time in the folder name. In this folder, more folders are created for each seed, and then each seed folder will contain folders for the stdout and stderr outputs, which will contain the stdout and stderr files respectively for all runs of that seed. An example of the file structure is shown in Figure 1.

3.4 Program Flow

The flow of the system will begin with creating the necessary folders, including a temporary folder for storing modified ini files. Then the original omnetpp.ini file is cleaned by removing all comments and white space. Since the simulations will be run in parallel, a separate ini file is created from the cleaned file for each run to avoid potential conflicts. The files are also named uniquely. The modified parameter in the ini file is the seed number which is used to choose the density of cars in the simulation. After all this setup is completed, a new subprocess is created to start SUMO via the veins_launchd.py script. This script will handle SUMO and its interactions with the simulation in the background. Next a list of tuples containing each seed/run pair is created to be used as parameters for each call to the start_simulation method. At this point, the list of tuples is then sorted according to the selected parallel computing policy, and a pool of tasks is created for parallelizing each simulation in the batch, utilizing the number of available cores in the user's CPU. The start_simulation function will be called for each seed/run pair as previous runs finish and cores become available. This function will remove its respective ini file from the temporary folder to the HOMng folder, create a custom opp_run command to launch the simulation and use it to create a new subprocess that begins the simulation. After the simulation is finished the ini file is moved back to the temporary folder, the output is retrieved from the subprocess and saved in the appropriate results folders. The complete workflow can be seen in Figure 2.

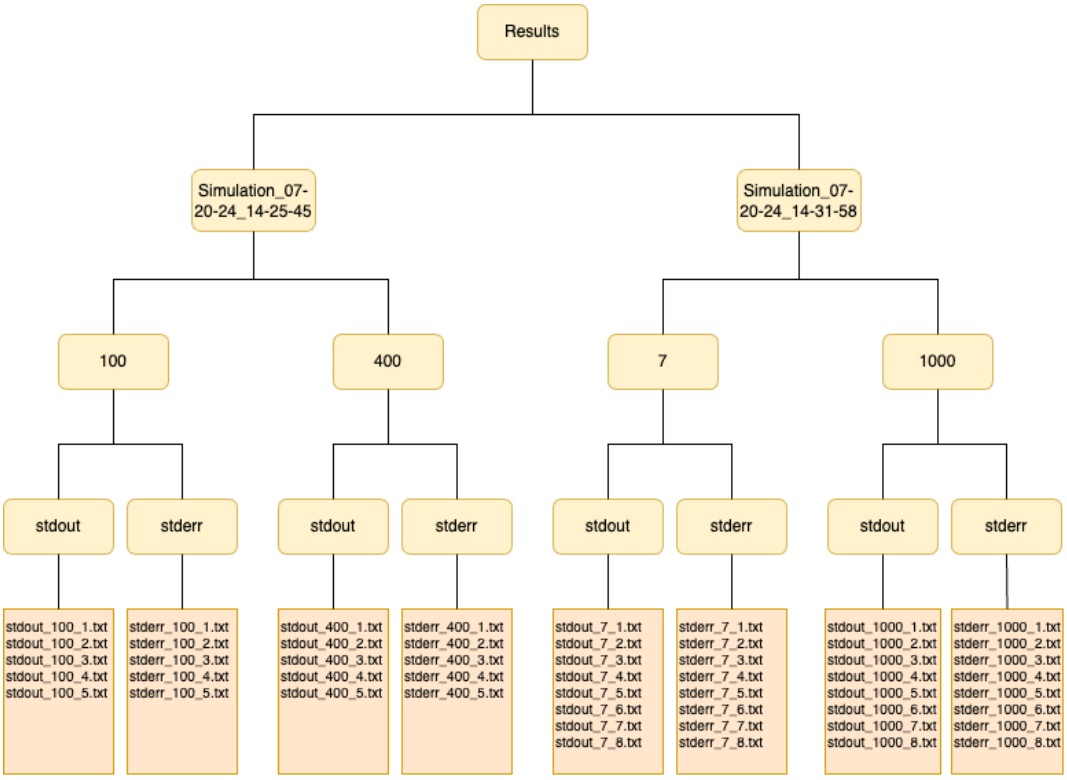

Figure 1: Sample file structure of simulation results.

4 Proof of Concept

This section first describes the setup used with the batch running system. The setup focuses on the software used in the tests and chooses to ignore hardware (aside from a few noteworthy resources allocated to the virtual machine). This is because the specific hardware used is irrelevant and will likely be different in other users' machines. It is assumed that users running these simulations have a machine that is capable of running them in a realistic time frame.

What *is* important is comparing the relative speed of running a batch of simulations with, and without our batch running system, which will provide some insight into the increase in efficiency from using our system. After describing the setup we describe a testing scenario for these comparisons, followed by the results.

4.1 Setup

4.1.1 Virtual Environment

- **Virtual Machine:** Oracle VirtualBox 7.0
- **Operating System:** Ubuntu 20.04
- **Allocated Resources:** 8192 MB base memory, 4 processors

4.1.2 Simulation Software

- OMNeT++ 6.0
- INET 4.3.2
- SUMO 1.6.0
- Veins 5.2
- Simu5G 1.1.0

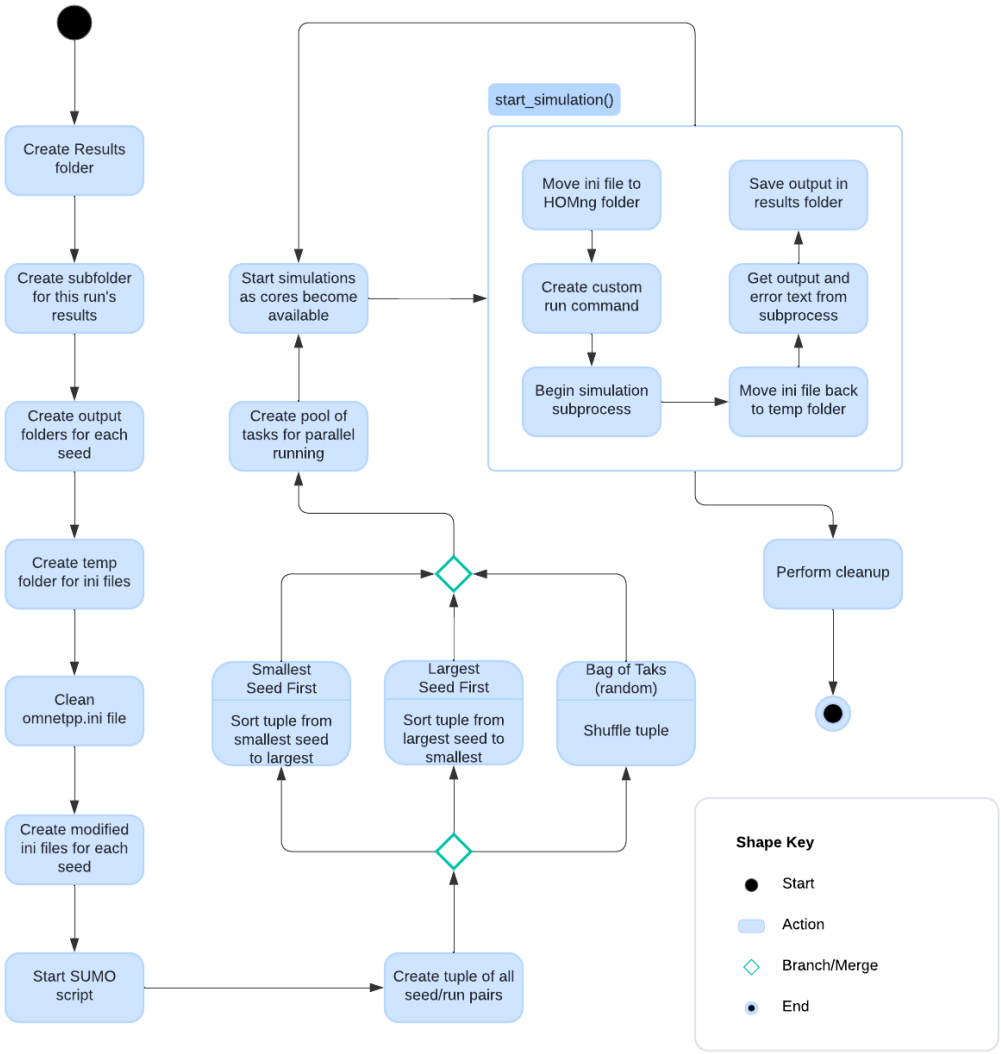

Figure 2: System workflow

4.2 Programming Language

- Python 3.8.10
- **Libraries:**
  - **subprocess:** for running terminal commands
  - **multiprocessing:** for running simulations in parallel
  - **pathlib:** for handling file paths
  - **shutil:** for deleting directories (cleanup)
  - **psutil:** to kill a running process (cleanup)
  - **uuid:** to create unique IDs used in temp folder names
  - **Tkinter:** for GUI creation

4.3 Testing

4.3.1 Scenario 1

This test performs two identical batch runs, one using our batch running system, and one using a simple command-line call. The parameters of the test are as follows, where seed represents the number of cars in the simulation, simulation time limit is the amount of simulated time the cars are moving in the simulation, and repeats is the number of repeat simulations run for a specific seed.

- **Seed:** 7
- **Simulation Time Limit:** 100 seconds
- **Repeats:** 8
- **Notes:** Doing a batch run from a command-line call with these settings in the ini file will run the simulations sequentially, giving us a baseline of how long it takes to do a normal batch run. Using the batch running system with four cores should give a best-case scenario of how long it could take when the runs are parallelized. Theoretically, the first four runs should start and end at the same time, then the last four should start just after that and end at the same time. A 9th run would dramatically alter the time taken since it isn't a multiple of the core count, leaving three cores unused for $\frac{1}{3}$ of the total batch run. For this simple test, we will use a multiple of the core count to see what the best-case yields. Since there is only one seed for this test, the parallel processing policy is irrelevant here.

4.3.2 Scenario 2

This test performs a more realistic series of runs including multiple seeds and repeats. The chosen parameters are tested without using our system via command-line calls, and then with our system, using each parallelization approach (smallest seed first, largest seed first, and random used twice). The parameters for this test are shown below, with each seed:repeats pair shown together. The simulation time limit has been reduced to shorten the experiment time since larger seeds are being used, however the relative simulation times will give insight into the increase in efficiency.

- **(Seed:Repeats):** (7:6, 100:5, 400:4, 700:3)
- **Simulation Time Limit:** 10 seconds
- **Notes:**

4.4 Results

The results of the above tests are summarized in Table 1 and Table 2 for scenarios 1 and 2 respectively.

Test Scenario 1	Total Time	Average Per-Run Time
Without our system	42 minutes 25 seconds	5 minutes 16 seconds
Using batch runner	12 minutes 50 seconds	1 minute 36 seconds

Table 1: Comparison of simulation run times with and without our batch running system (Scenario 1)

4.4.1 Scenario 1 Results

The results in Table 1 show that the batch running system runs the simulations in this scenario roughly 3.29 times faster than a command-line call. This implies that the time increase doesn't correlate exactly with the number of cores, which in this case would be exactly four times faster, however this test still proves a significant increase from parallelizing the runs.

4.4.2 Scenario 2 Results

Test Scenario 2	Total Time
Shortest Seed First	14 minutes 10 seconds
Largest Seed First	10 minutes 50 seconds
Random 1	10 minutes 11 seconds
Random 2	13 minutes 54 seconds
Command-line (without our system)	36 minutes 53 seconds

Table 2: Comparison of simulation run times using different parallelization policies, and without using our batch running system (Scenario 2)

The results in Table 2 describe the total batch run time for the given set of parameters using each implemented parallel processing policy, which includes two runs using the Random policy, and finally initiating the same amount of runs using the command-line (without our system). Again, our system shows similar results to the increase in efficiency as testing scenario 1. In this particular test, choosing the Largest Seed First policy was significantly better than Shortest Seed First, and both Random runs demonstrated a significant difference from each other, however Random run 1 performed best overall, suggesting there may be a policy that isn't random that can perform better than the others.

5 Future Work and Improvements

Currently our system allows for a detailed amount of control to set parameters for batch runs, which includes selecting any number of seeds, each with their own number of repeats, as well as selecting a preferred parallelization policy. Users are also able to select where they want the results of the runs to be stored on their system. The GUI makes selecting these parameters easy, reducing possible errors by error-checking input before allowing simulations to run, and using GUI components to select necessary directories and files instead of the user manually typing in file paths. Once a simulation has started, a progress indicator shows the total progress based on how many individual simulations have been completed. Lastly, our system collects and organizes the output of each run by retrieving the standard output and error output from the process that initiates each run.

An important and high priority future improvement to the system would be introducing better performing parallel processing policies. An algorithm that selects the given seeds and repeats and maximizes the active time of all cores could achieve this.

Other potential future improvements to the system may include more advanced progress tracking, since runs with larger seeds will take noticeably longer than runs with smaller seeds. A potential solution may include collecting data on the relative run times between all possible seeds, and assigning weights to each run based on the seed. Another possible improvement to showing progress is to implement a GUI component that shows the runs, possible each as a red circle, which then change colours in real-time as they are executing (blue), and again once the run has completed (green). This detailed view will allow users to see exactly which runs have been started and completed.

Finally, another improvement could potentially be made to how simulation results are handled. Currently, the given output from each run is just copied as-is (with minimal formatting to remove blank lines). The system could parse these files in more depth to provide useful summaries to the user.

6 Conclusion

With the complexity and significant time requirement of running Vehicular Network simulations, a solution to increase efficiency is certainly worth exploring. Having a software system to automate and streamline the process of configuring and running simulations not only simplifies the process for the user, but our system also implements parallel running of batch simulations to significantly speed up total batch run times. In our test setup, the system had four processing cores, which resulted in batch run speeds that were 3.29 times faster. Systems with even more cores will enjoy an even more significant boost in time efficiency. In addition to the streamlined experience and increase in run time efficiency, our system automates collecting and organizing all simulation results for the user.

References

[1] Foundation, P. S. Python language reference, version 3.8.10. https://www.python.org/.

[2] Lopez, P. A., Behrisch, M., Bieker-Walz, L., Erdmann, J., Flötteröd, Y.-P., Hilbrich, R., Lücken, L., Rummel, J., Wagner, P., and Wiessner, E. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems* (2018), IEEE.

[3] Nardini, G., Sabella, D., Stea, G., Thakkar, P., and Virdis, A. Simu5g–an omnet++ library for end-to-end performance evaluation of 5g networks. *IEEE Access 8* (2020), 181176–181191.

[4] Sommer, C., German, R., and Dressler, F. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE Transactions on Mobile Computing 10*, 1 (January 2011), 3–15.

[5] Varga, A. *OMNeT++*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 35–59.

[6] Österlind, F., Varga, A., Körner, C., Requena-Esteso, M., and Elsts, A. Inet framework for omnet++ 4.0. https://inet.omnetpp.org, 2022.