Analysis, Performance, and Optimization of the 5G Network Slicing service

Nilson Peres

Computer and Mathematical Sciences Institute, São Carlos, Brazil
T-Mobile, Bellevue, United States
nilson.peres@usp.br, nilsontinassi2006@hotmail.com

1 Contextualization

Self-Organizing Networks (SON) are a significant advancement in the telecommunications field, allowing mobile networks to configure, optimize, and heal themselves automatically.

This report aims to explain in an objective, clear, and cohesive manner the functioning of self-organizing networks (SON), using the MantaRay SON implementation tool and its modules. The MantaRay SON tool uses Python as a programming language for creating custom functions. Special emphasis will be given to the network slicing function or module, for which an efficiency analysis and performance optimization proposal will be conducted.

1.1 Self-Organizing Networks (SON) Categories

Self-organizing networks are categorized into three main functionalities:

- Self-configuration: Facilitates the construction of neighbor cell lists, cell code reuse (PCI), and cell parameter adjustment.
- Self-optimization: Includes network performance report generation, real-time alerts, and parameter auditing, allowing continuous adjustment to maintain optimal performance.
- Self-healing: Focuses on load balancing and sleep mode recovery to ensure service stability and continuity.

Through these functionalities, self-organizing networks have stood out in telecommunications network architecture, providing a secure and dynamic solution for network maintenance and expansion.

1.2 Implementation Tool: MantaRay SON

MantaRay SON is a tool used to implement self-organizing network functionalities, developed by Nokia. Its operation is based on modules or functions, which can be divided into:

- Vendor Standard Modules: Developed by the tool manufacturer (Nokia) and offered as part of the tool package to solve problems common to most telecommunications network operators.
- Custom Modules: Developed on demand by the tool manufacturer (Nokia), specialized companies, or internal developers of the contracting company, with the objective of solving specific problems not covered by the tool's standard modules.

These modules are designed to meet different network needs, allowing high flexibility and adaptability through custom modules.

1.3 Base Language: Python

Both standard and custom modules are developed using the Python programming language, endowed with simplicity, versatility, and a vast amount of available libraries. These characteristics facilitate the development and implementation of specific solutions for self-organizing networks.

1.4 Network Slicing

Network slicing is one of many solutions that can be implemented in self-organizing networks, especially using the MantaRay SON deployment tool.

The network slicing technique allows dividing a physical network into several independent virtual networks, optimizing resource usage and improving efficiency.

A simple example of network slicing would be, for instance, dividing the network into 3 slices:

- Slice 1 - Connection Speed: Focused on devices like cell phones, computers, TVs, etc., that require high connection speed.
- Slice 2 - Reliability and Low Latency: Essential for critical applications like autonomous cars, remote surgery robots, among others, where reliability and low latency are crucial.
- Slice 3 - Massive Connected Devices: Directed toward the Internet of Things (IoT), which requires support for a large number of simultaneously connected devices.

One way to implement network slicing is through a specific custom module called NetworkSlicing, which was developed for this purpose.

2 Problem Statement

The efficient implementation of network slicing is essential to meet the growing demands of mobile networks.

The NetworkSlicing module algorithm must comprise four main stages:

- Input File Reading (template): This initial stage involves reading a template file that contains the parameter changes necessary to implement network slicing.
- Processing of Suggested Changes: In this phase, the algorithm verifies if dependencies are satisfied, if there is a need to update or create new parameters, among other technical aspects.
- Validation of Proposed Changes: The algorithm compares suggested values with current network values to verify if they are different and adequate.
- Application of Changes to the Network: After validation, changes are applied to the network, finalizing the implementation process.

Each stage involves complex processes that can be performed at the input file (template) level or at each cell level.

In the initial version of the network slicing module (NetworkSlicing), the algorithm execution time to process 2,000 cells was approximately 30 minutes. Due to the high processing time, a demand arose to refactor this code, aiming to optimize its performance and execution time.

This demand is understandable, considering that T-Mobile's network in the US has about 950,000 5G cells, resulting in an estimated time for complete implementation of 237 hours.

The objective is to significantly reduce execution time, aiming to process 30,000 cells in 2 hours, which represents a 74

To achieve this desired state, a detailed algorithm analysis and application of techniques to reduce system complexity are necessary.

3 Current State Analysis

Complexity analyses can be divided, most of the time, into two scenarios:

- Execution Time Complexity
- Memory Space Complexity

In the scope of this analysis and considering the demand, the chosen approach was Execution Time Complexity, since, although there are limiting factors regarding memory consumed by the module, these limits were never reached by the initial version of the module, so there is no demand for space optimization.

3.1 Simplified Counting

Performing simplified counting considering all operations would be unfeasible, since the SlicingNetwork module has more than 2600 lines of code.

Thus, in the simplified counting, mainly loops were considered

- s - number of spreadsheets in the input file;
- ri - number of rows in each spreadsheet of the input file;
- c - number of cells to be audited;
- n - number of DNs (distinguished names) to be updated;

Parameters **s** and **ri** are variables depending on the input file and may be updated over time, but for the analysis, the last template or network slicing configuration file provided by the network architecture sector was considered.

Parameter **c** is variable depending on the module execution instance, which may contain any number of selected cells, according to function usage needs.

T-Mobile has in its network more than 900,000 5G technology network cells for which network slicing should be applied, these cells are spread throughout the United States territory in different regions.

For the analysis, different values of **c** were considered.

The value of n (number of DNs) depends on the network configuration and each cell, according to satisfied or unsatisfied dependencies, current values, and suggested values for parameters.

A cell may have several DNs that need to be updated. It is difficult to estimate this number since in each module execution the number of DNs per cell may differ due to constant parameter updates in the network.

For each group of cells, a number of DNs that need to be updated was identified.

The approximate total number of operations is governed by the following approximation:

$$(c + 3 \cdot n) \cdot \sum_{i=1}^{s} r_i$$

with **n** equal to the number of DNs to be updated.

3.2 Worst and Best Case

As mentioned earlier, the worst and best cases depend heavily on network configuration and architecture.

Consider the case where network slicing has been completely implemented for a given set of cells **c**. In this scenario, no DN would need to be updated and, therefore, the value of **n** would be 0, leading to the best possible case:

$$\sum_{i=1}^{s} r_i \cdot (c)$$

On the other hand, assuming all cells need to update all DNs, the number of DNs **n** that will be updated will be the total number of DNs among all cells and parameters to be updated. This would be the worst-case complexity.

$$(c + 3 \cdot n) \cdot \sum_{i=1}^{s} r_i$$

with **n** equal to the total number of DNs.

4 Techniques Applied for Optimization

Different techniques were applied to achieve execution time optimization.

Among the applied techniques, the following stand out:

4.1 Updating the Execution Level of Operations

In the initial version of the SlicingNetwork module, many operations that could be executed at the configuration file (template) level were executed at the cell or DN level, meaning unnecessary repetition of operations that would always lead to the same result.

For example, each cell in the network architecture has a software version. Among the 1342 changes proposed by the configuration file (template), 380 have some dependency with the current software version of the cell.

This dependency is specified in the template in a specific column, with values such as:

- version_check¡=23R1;
- version_check==24R1;
- version_check¿23R4.

Considering an execution for 10,000 cells, split or text separation operations would be performed 10,000 times for each of the 380 proposed changes, totaling 380,000 operations.

By performing this operation directly on the configuration file, only 380 operations are needed, saving much time.

This same technique was applied not only to dependency processing from the input configuration file but also to the column of parameters to be updated and the column of values to be used in the update.

4.2 Operation Vectorization and DataFrame Usage

In addition to changing the level of operation execution, a change was also made in how operations were performed.

Initially, the configuration file (template) was read as a workbook object and each spreadsheet row was manipulated individually.

In the new approach, data reading was performed using dataframes.

Having data in dataframe format allows exploring advanced data manipulation techniques through vectorization, instead of operating on each data individually.

Vectorization allows batch operation execution instead of explicitly iterating over individual elements. This means operations are applied to entire datasets at once, instead of one cell or row at a time. For example, adding 1 to each element of a pandas dataframe column is much more efficient than iterating over each element and adding 1 individually.

Additionally, vectorized operations in pandas are implemented in C or Cython, which are much faster than pure Python code.

Another important factor is that many vectorized operations are internally parallelized, meaning they can take advantage of multiple CPU cores without the need for additional code for thread or process management.

Still using the cell software version verification example, through vectorization it is possible to perform the split for all spreadsheet rows in a single batch, instead of reading each row and applying the operation individually.

4.3 Cache Usage

Each change proposed in the configuration file is composed of the following specification set:

- Parent MO: Parent object identification;
- MO: Object to be updated identification;
- Parameter: Parameter to be updated identification for the object;
- Value: Value that the parameter to be updated will assume;
- Dependency: Dependencies that must be satisfied to update the parameter.

In the initial version of the NetworkSlicing module, for each row containing these specifications, the list of DNs to be updated was determined using Parent MO and MO.

A DN is a path that identifies the object that will have the parameter updated.

For example: PLMN-T3OSS/MRBTS-105300/NRBTS-105300/NRCELL-301/

In the DN above, it is possible to identify several MO levels:

- PLMN-T3OSS is the highest MO level;
- MRBTS-105300 is a MO level with lower granularity, in this case referring to a node or antenna in the network;
- NRBTS-105300 is a MO level with even lower granularity, which identifies the cell technology type, in this case NR or 5G;

– NRCELL-301 is a MO level with lower granularity that identifies a cell itself.

With the example DN, it is possible to access all parameters of cell NRCELL-301.

There are even lower levels than NRCELL, such as: NRDU, which would result in the following DN: PLMN-T3OSS/MRBTS-105300/NRBTS-105300/NRCELL-301/**NRDU-1**.

In the case of SlicingNetwork, for each cell and Parent MO and MO specification given in the configuration file, the DN list was determined, but often this resulted in unnecessary operations.

In some cases, it is possible that dozens of cells are under the same antenna (NRBTS), so the DN list would be the same for all these cells. Still, the module recalculated this DN list every time.

An update was made to save DN lists in a cache, so that for all other cells at the same granularity level, it is not necessary to redo all operations.

In practice, consider updating a parameter at the MRBTS level, which on average contains about 100 cells.

In the previous version, all operations were performed 100 times. After the update, operations are performed only once, saving much execution time.

The same cache technique is used regarding parameters and dependencies.

For example, of the 1342 changes proposed in the configuration file, there are 66 version checks ¿= 23R4. In the previous version of the module, this would mean checking the version for each cell 66 times. Considering 10,000 cells, it would mean at least 660,000 operations.

Using cache, this process is performed only once for each cell, reducing from 660,000 to only 10,000 operations.

5 Results Analysis and Conclusion

The application of the above techniques, together with other simplifications, allowed reducing the module code size from 2600 lines to approximately 900 lines.

It is clear that the number of code lines does not say much about its efficiency, but it helps to realize that refactoring allowed eliminating much code that could be simplified, also helps to realize that there was plenty of room to make the code more functional (use of functions instead of copy/paste).

But, regarding execution time, the following test parameters are followed:

– Instances executed at the same time;
– Instances executed with the same cell scope;
– Instances executed with the same input file;

The analysis performed considered, therefore, the following parameters:

– s = 2;
– r1 = 627;
– r2 = 715;
– c1 = 12;
– c2 = 250;
– c3 = 500;
– c4 = 1,000;
– c5 = 2,000;
– c6 = 8,000;

And the results are arranged in the following table:

The results show that for executions with few cells (c1 = 12), the new version of the module is less efficient than the initial version.

This longer execution time can be justified by data processing, library loading, and low cache utilization (since in few repetitions, the benefit of using cache ends up being underutilized).

However, with the increase in the number of cells for which the module is executed, the optimization benefit becomes clear, since already for c2 = 250, a 285% faster execution time was observed in the new module.

Furthermore, while in the old version the execution time per cell becomes higher with the increase in the number of cells, in the new version the opposite happens, with execution time becoming increasingly smaller with the increase in the number of cells in scope.

While in the old version, with c1=12 cells the execution time per cell is 0.66 seconds/cell, for c6=8,000 cells the execution time reaches 2.73 seconds/cell.

Number of Cells	Old Version (s)	New Version (s)	Ratio New vs Old
12	8	32	400% (slower)
250	154	54	285% (faster)
500	448	72	622% (faster)
1000	437	102	428% (faster)
2000	2366	154	1536% (faster)
8000	21840	300	7280% (faster)

Table 1: Comparison of SlicingNetwork Module Execution Time Results

In the new version, the module initially takes about 2.66 seconds/cell (for c1=12 cells), but reaches extraordinary 0.03 seconds/cell (for c6=8,000 cells).

These results lead to the conclusion that the updates made to the module not only made its execution more efficient but completely changed the module's complexity behavior.

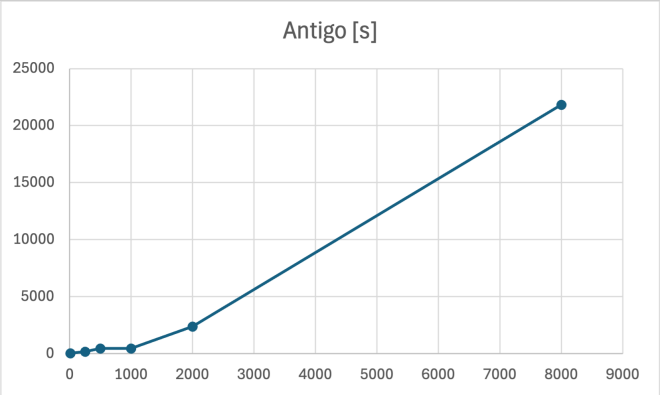

Fig. 1: Slicing Network Module Execution Time Curve - Old Version

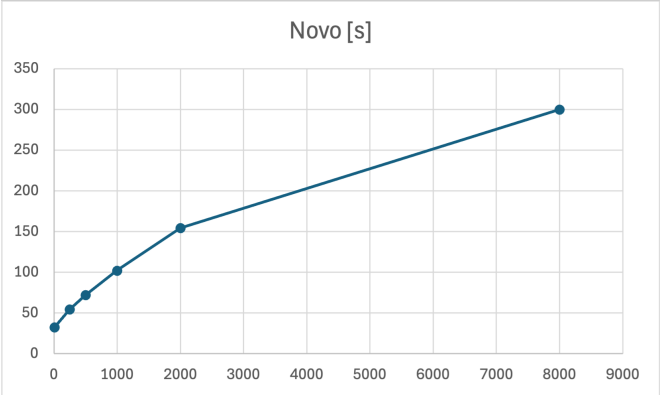

Fig. 2: Slicing Network Module Execution Time Curve - New Version

In the initial version, the module had increasing behavior (longer execution time for larger number of cells): a 600-fold increase in the number of cells resulted in a 2730-fold increase in execution time. If it were proportional behavior, increasing the number of cells by a factor k should also increase execution time by k, but the analysis revealed that an increase of k = 600 resulted in an increase of 4.55*k. This result would only tend to worsen considering the execution time curve 1.

The new version of the module demonstrates decreasing behavior (shorter execution time for larger number of cells), as also observed by the execution time curve 2.

These results allow us to affirm that the initial objective was achieved and that the new version of the module is capable of meeting T-Mobile's needs in the North American market, allowing parameter changes for network slicing throughout the company's entire network in a matter of hours.