Rheinisch Westfälische Technische Hochschule Aachen
Lehrstuhl für Software Engineering

Just-in-time (JIT) compilation for the NEST Simulator to extend the boundaries of neural network sizes on HPC clusters

Master Thesis

presented by

Benelhedi, Mohamed Ayssar

1st Examiner: Prof. Dr. A. Morrison

2nd Examiner: Prof. Dr. B. Rumpe

1st Advisor: Charl Linssen

2nd Advisor: Dr. Jochen Martin Eppler

The present work was submitted to the chair of software engineering

Aachen, den June 15, 2022

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Benelhedi, Mohamed Ayssar	365558
Name, Vorname/Last Name, First Name	Matrikelnummer (freiwillige Angabe) Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

Just-in-time (JIT) compilation for the NEST Simulator toextend
the boundaries of neural network sizes on HPCclusters

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen
*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Ort, Datum/City, Date

Unterschrift/Signature

Abstract

NEST (**??**) is a simulator for large-scale networks of spiking and non-spiking neural networks used in computational neuroscience to build models of brain-like or brain-inspired neuronal circuits. It profits from the efficiency of C++ to simulate networks with large numbers of neurons and synaptic connections and is parallelized using MPI and OpenMP. By providing a Python interface (*PyNEST*; **?**) that interacts with the *NestKernel* in C++ (**?**), users can easily define their simulations and add new models dynamically at the runtime by means of the function `nest.Install()`.

Originally, models had to be manually written in C++. Since some years, the model creation process is supported by *NESTML* (**?**), which generates the required code from a high-level specification and compiles them into dynamic libraries that are ready to be used from within *PyNEST*. NESTML is a user-friendly, flexible and Turing-complete domain-specific language. It simplifies the modeling process for neuroscientists by allowing them to express their models in domain terminology, both with and without prior training in computer science. By specifying the target type – either a different hardware or new supported simulator – the user can use the same model to generate code for the given target without any manual interventions.

Running a simulation script in NEST using an external custom model requires the user to provide certain configurations to the NESTML code generator. Such configurations cover the model source code location, model dependencies to other models (e.g., custom synapse models) and the target platform. Although the NESTML code generation is completely automated, the user still has to explicitly invoke the NESTML interface for the model to be processed and compiled into an extension module for NEST, which is especially cumbersome if many neuron/synapse combinations are used.

The first goal of this thesis is to eliminate the explicit calls to the NESTML interface in the simulation script by a *seamless* extension to PyNEST that intercepts calls to `nest.Create()` and `nest.Connect()` and controls the workflow logic of the NESTML functions behind the scenes. Depending on certain conditions this integration decides if the model should be instantiated right away, or if the code generation and compilation can be delayed to a later point in time. As the workflow is only triggered when the user invokes specific functions instead of invoking NESTML upfront, this extension is called just-in-time compilation (*JIT*).

One problem that arises from a delayed availability of the model instances when using JIT is that model parameters cannot be queried before the actual start of the simulation. This prohibits adaptive parameter choices for following calls to PyNEST and thus restricts the flexibility for the users. A possible solution would be to cache parameters on the Python level until the instances are available, albeit at the cost of doubling the memory requirements. A more efficient solution is to provide a model independent storage of parameters in the form of data vectors. This second solutions also vastly increases the cache-efficiency of models by applying a single-instruction-multiple-data (*SIMD*; **?**) paradigm that increases the performance of the simulation script by storing all objects as an array of structures (*AoS*).

The second part of this thesis thus focuses on extending the NestKernel to support *vectorization* by modifying the models to become the central unit for controlling the state of node instances: Instead of storing attributes in the instances direcly in the nodes, the vectorized model contains attributes in the form of *vectors*, and the size of these vectors corresponds to the number of instances created during the simulation of that particular model type.

Together, *JIT* and *vectorization* result in a much more modular NestKernel in which *JIT* is responsible for generating and assembling the components of the model into executable code, while the *vectorization* is responsible for linking the components together in a way that can be efficiently run on all machines from laptops to supercomputers.

Contents

Chapter 1

Introduction

1.1 Overview

The human body has always been an inspiration for many technologies. Scientists have successfully achieved to fully understand the mechanics behind different parts of the human body and have created models that imitate the functionalities of the studied parts. However, The human brain is one of the most complex organs in the human body (**?**) that we still don't fully understand its mechanics. Creating machines that can outstrip human capabilities such as high-level cognition associated with conscious perception is a bit of a controversial goal and before diving deep into that topic, we need to understand how specific information processing of the brain would be replicated by machines and how to create models that target specifics activities in the brain and replicate their behavior in connection with other parts.

The successful understanding and transfer of knowledge gained from neuroscience to the development of *neural simulator* is strongly dependent on the interaction between researches from both the fields of computer science and neuroscience. Such interaction may lead to accelerate the growth of the simulation frameworks and to a better understanding of the human mind. Distilling intelligence into a machine and comparing its behavior with the human mind might yield interesting insights into demystifying some mysteries of the brain, such as understanding the source of creativity, dreams and possibly even understanding consciousness.

Using the Nest Modeling Language *NESTML* together with the Neural Simulation Tool *NEST*, users can write custom neural and synapse models with different specifications and use them in their simulation scripts by simply installing the generated and complied library of the model at the runtime. In general, there are two use cases for generating such models. The first and simple case is generating the code for simple neurons without specifying the synaptic connection of the model. The second and more complex is co-generating a neuron model and synapse together. The latter case is important as it tackles some combinatorial and performance issues. Unfortunately, *NEST* can't know in advance which case must be handled, and it must be explicitly given to *NESTML* to process these configurations. Apart from these cases, the user must specify the location of the model, target platform and if needed a custom template defining how to generate the `C++` code for the given models.

The thesis focuses on *virtually* integrating *NESTML* in *PyNEST* (the Python interface for *NEST*). This virtual integration means that we won't modify the *PyNEST* module by introducing new logic or implementing new functions, but instead we intercept the interface function calls and then decide to do some processing before or after the function call or just totally ignore the function and execute it later in the script. Mathematically described, each function f in *PyNEST* is replaced by one of the following execution paths:

$$
\begin{aligned}
f &\mapsto \emptyset \\
f &\mapsto f \circ before \\
f &\mapsto f \\
f &\mapsto after \circ f \\
f &\mapsto after \circ f \circ before \\
f &\mapsto g: \ \text{\#replace the call of } f \text{ with } g \in \{before, after\}
\end{aligned}
$$

The empty set means that we intercept the function call but do nothing. The functions `before` and `after` refer to a pre-processing and post-processing steps. It is not always the case that we have to call both functions, as we may have to call either one of them or even, like depicted in the last case, we can make the call to the function f make something totally different and hide this new behavior inside one of the processing functions. Together, the interception logic with *NESTML*, the *JIT* mechanism should be able to control the execution of the simulation script and decide when and which models should be instantiated and make all this decisions transparent for both, the user and the simulation script.

Apart from virtually extending the *PyNEST* interface, we adjust the `NestKernel` infrastructure to support *Vectorization* (**?**). The approach is based on modifying the data structure representing the created neurons models to a new data structure in a *vector* form. This new form supports the parallel and independent processing of neurons and compared to the current representation of neurons in the *NestKernel*, it should reduce the number of cache misses (**?**), and thus we expect a speedup and performance gain during the simulation with a large network.

1.2 Task Description

My main task in the thesis is to implement a wrapper around *PyNEST* to support the *Just-in-Time (JIT)* mechanism. They are two main requirements that must be satisfied. The first is making the *JIT* implementation as an option. It means the user can explicitly disable or enable *JIT*. The second requirement is having fewer changes in the simulation script when enabling the *JIT* mechanism. Thus, comparing the simulation script with and without using *JIT*, we should observe minimum variations in the code.

The second challenge is extending the architecture of the *NestKernel* to support a data structure of neurons in a vector form. We refer to this new architecture as *Vectorization*, and it should make use of *single instruction, multiple data* execution and, depending on the used models, it may reduce the number of cache misses and thus gain more speedup.

1.3 Thesis Outline

In **??**, we introduce *PyNEST* and the *NestKernel* with the focus on the major components involved in supporting both the *JIT* and *Vectorization* implementations.

?? dives deep into understanding the requirements of the *JIT* mechanism, explaining the chosen solution and comparing it with different possible approaches. Having the suggested solution explained, we point out to the design and implementation decisions. Finally, we illustrate an example how to enable *JIT* in the simulation script and discuss the few changes that must be applied to the script.

?? introduces the data structure and the necessary changes in the *NestKernel* to support *Vectorization.* Furthermore, we discuss the wide range of optimization features that together *Vectorization* and *JIT* may provide.

In **??**, the benchmark results are presented together with a set of use cases that exploits the limitations of *Vectorization.*

Finally, **??** contains a discussion of the solutions that are developed in the thesis and proposes a set of additional implementation ideas that could make use of *Vectorization.*

Chapter 2

Fundamentals

To understand the logic behind the implementations made in the thesis, we have to understand the execution workflow between the different modules (*PyNESTML*, *PyNEST* and *NestKernel*). Those three modules may only be used together if we want to build our own custom neuron and synapse model, and therefore, in this chapter we will only focus on the execution of the simulation script using a user defined external model. *PyNEST* does not enforce any order when to generate the code of the external model and install it, as along as the instance of the model is not used yet in the script. Thus, to keep everything simple and consistent, the simulation script will always start by generating all models and then installing them.

2.1 PyNESTML The Code Generator

The first step in each simulation script using an external model starts with calling the `generate_target` function that is responsible mainly for everything in making the model available to the *NestKernel* (generating the code, building the library and adding it to the installation path). The function takes different necessary parameters to ensure the correctness of the code generation. Mainly, the first parameter is the `input_path` which holds the location of the model in the file system. This parameter can also be a list of paths, and in this thesis we only use the list when we want to co-generate the neuron and synapse models together and in all other cases we only generate one model at a time.

The second important parameter is the `codegen_opts` which is responsible for providing extra information about the neuron and if the code generation of the neuron should use a different logic in creating the library code. More importantly, this parameter is used for specifying the (*neuron*, *synapse*) pairs that are handled in `PyNESTML` to co-generate an efficient code and binding the `synapse` model only to the listed `neuron`.

PyNESTML can generate code for different target platforms, but for the scope of the thesis, we only focus on *NEST* as the `target_platform`. In the scope of the thesis, the remaining parameters are not of importance and do not affect the execution flow.

A *neuron* model in *NESTML* is the composition of different three blocks. The *State* block defines a set of state variables representing the initial values of the model upon instantiation, and they can be updated during the simulation. The *Parameter* block declares a set of variables depicting the model's constants that won't be modified during the simulation. Finally, we have the *Internals* block. The block behaves like the *Parameter* block, with the only difference that its variables can not be set outside the model (the user cannot explicitly modify the values before or after the simulation) and their initialization expression can only reference parameters or other internal variables.

The *Neuron* model also has other important blocks, like the *equation* block that handles *differential equations* describing the time evolution of variables in the *State* block. An *update* block for updating the state of the neuron instance during the simulation run, and finally the *input* and *output* blocks for defining the incoming and outcoming signals respectively.

Each *NESTML* neuron model is read by a parser and converted to an intermediate representation known as the abstract syntax tree (`AST`). After parsing the model and making sure that the described model is syntactically correct, the generated intermediate representation `AST` undergoes several transformations. Such transformations cover analyzing differential equations, processing the *update* block and creating *hooks* for the GNU Scientific Library. After applying these transformations, the `AST` goes through an optimization step, which includes *constant folding* by eliminating expressions that calculate a value that can already be determined before code execution.

Figure **??** below summarizes the complete workflow of the code generation in *PyNESTML*. The process starts by feeding the *.nestml* model to the code generator pipeline. A parsing and a validation steps take place, where the model is checked if it is syntactically correct and satisfies the code generator rules. Once the checks are complete, the pipeline moves to transforming the model into an intermediate representation in order to target different platforms. The third step is then the actual code generation, which is based on a set of templates written in *Jinja* providing the implementation of the model in `C++` for the desired target platform. Finally, the last step in the pipeline controls the compilation and building of the model into `C++` *shared library* that can be loaded at runtime in the *NestKernel*.

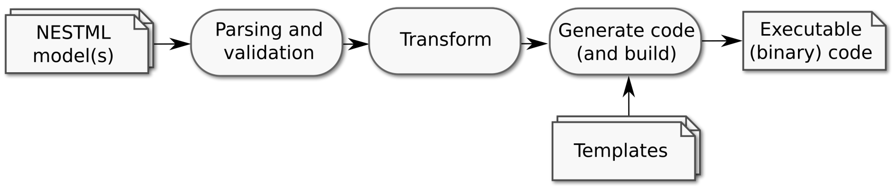

Figure 2.1: An overview of the processing workflow of *PyNESTML*: The process starts by feeding the *NESTML* models to the code generation pipeline. The first step in the pipeline starts by parsing the models and validating its syntax. Once the first step is completed, the models are transformed from a *NESTML* files to an *AST* objects. The pipeline moves to the next step by transforming the *AST* objects. The transformation may extend the structure of the *AST* by introducing new attributes or modify the name of the current model's attributes. Finally, we have the last step in the pipeline that takes the transformed *AST* object and generates the *C++* code for the model by using custom defined templates. Are all steps completed without any failure, the pipeline may compile and build the library of the generated code of the model on request and make it available for the simulation.

Is the model compiled, built and ready for use, the next step in the simulation is to make use of the model by introducing it to the *NestKernel* and this happens by the call to `nest.Install()`. Therefore, in the subsequent section, we examine the execution path taken by registering new models in *NEST*.

2.2 NestKernel The Simulation Backend

This section serves as a basic introduction to the most important functionalities that will be affected by the new implementation in **??**. Registering new custom models and creating new nodes are the first steps in each simulation script, and usually the user should not worry about how nodes are created. Once the model is registered, the *NestKernel* creates a *factory* object responsible for instantiating nodes from the model and managing any modification made to the model by the user during the simulation. To further understand the *factory* concept in registering and creating nodes, we have to take a deeper look at the source code and inspect the important steps required for making the correct execution of the simulation. In the subsequent section, firstly we review the logic behind registering nodes and secondly, how to create node instances from these models.

From the start, we have mentioned that *PyNEST* is the Python interface for executing the *NestKernel* functions. That is actually not totally true. Each function call in *PyNEST* is directly forwarded to the *SLI* engine (**?**). SLI is a stack-based language derived from PostScript (**?**), and it is outside the scope of the thesis.

Figure **??** below depicts the three layers that are responsible for running the simulation, from the Python interface to the `C++` layer. *PyNEST* takes the arguments from the simulation scripts and processes the arguments to make them ready for the *SLI* interface functions. `SLI` then starts objects conversion and calls the exposed functions in the *NestKernel*.

2.2.1 Registering New Models

For abstraction, we use `N` as the custom model class type that will be registered and show the workflow in the message sequence diagram depicted in the **??**. Every custom model generated by *PyNESTML* is shipped with two main components (see **??**). The first component is the implementation of the model in `C++` and the second component is the derived class from the `SLIModule`, and it is responsible for registering the model in the *Nestkernel*. By calling the `nest.Install` function with the model library name as parameter, *SLI* searches for the library in the provided paths and dynamically loads it. Upon loading, *SLI* calls the `init()` functions in the derived `SLIModule` class. The `SLIModule` instance calls the `register_node_model` in the `ModelManager` class. The function required mainly two items. The first is the Class type name of the model as in the *template* name and secondly, the name of the model that will be available to the user to use.

The `register_node_model` function checks if there exists another model under the same name and only proceed to registering the model if its name is unique. Afterwards, a new instance of the `GenericModel<T>` class is created with our model `N` as the substituted type (i.e., `T = N`). In fact, the `GenericModel` class works as a factory responsible for creating the instances of `N`, which means that the user can not explicitly call the constructor of `N` and each creation of `N` is done by the `GenericModel`. Finally, the `ModelManager`

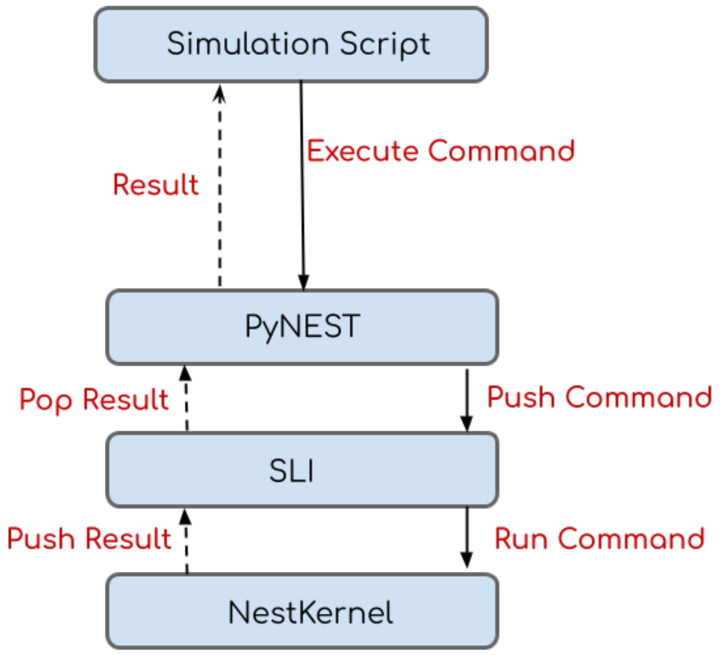

Figure 2.2: *NEST* layers: At the top level, we have the user simulation script that imports the *PyNEST* module. The *PyNEST* functions represents an intermediate layer between the simulation script and the *NestKernel*. They simply prepare the arguments of the command and push them onto the stack in *SLI*. *SLI* retrieves then the command and the arguments from the stack and call the corresponding function in the *NestKernel*. The result then takes the opposite path, from the *NestKernel* to the simulation script. After the completion of the command in the context of the *NestKernel*, the results are pushed onto the stuck, and then *SlI* comes again and removes them from the stack and make them available to the simulation script.

takes control again, sets the model parameters (i.e., `ID`) and lastly creates a `proxy` objects of the model in each *thread*. The `proxy` nodes are only required for nodes residing in remote processes, and they are meant for boosting the performance during connections setup and simulation. The `register_node_model` returns then the new `ID` of the model to `SLIModule` which itself returns the same `ID` to the initiator of the `install` command in *SLI*.

2.2.2 Creating New Nodes

In this subsection, we focus on three components in the *NestKernel*. The `NodeManager` which is responsible for creating nodes, handling the distribution of nodes over the available *threads*. It is important to keep in mind that each node is exactly assigned to one thread and each thread can only modify its nodes, non-local nodes are represented by *proxies*. Also, each thread knows the total number of the created nodes. The second component is the `GenericModel`, and as we have mentioned in the previous subsection that it plays the role of a *factory* responsible for creating the real instances of the registered model. Finally, the registered model `N` which provides an implementation for *cloning* that is required of

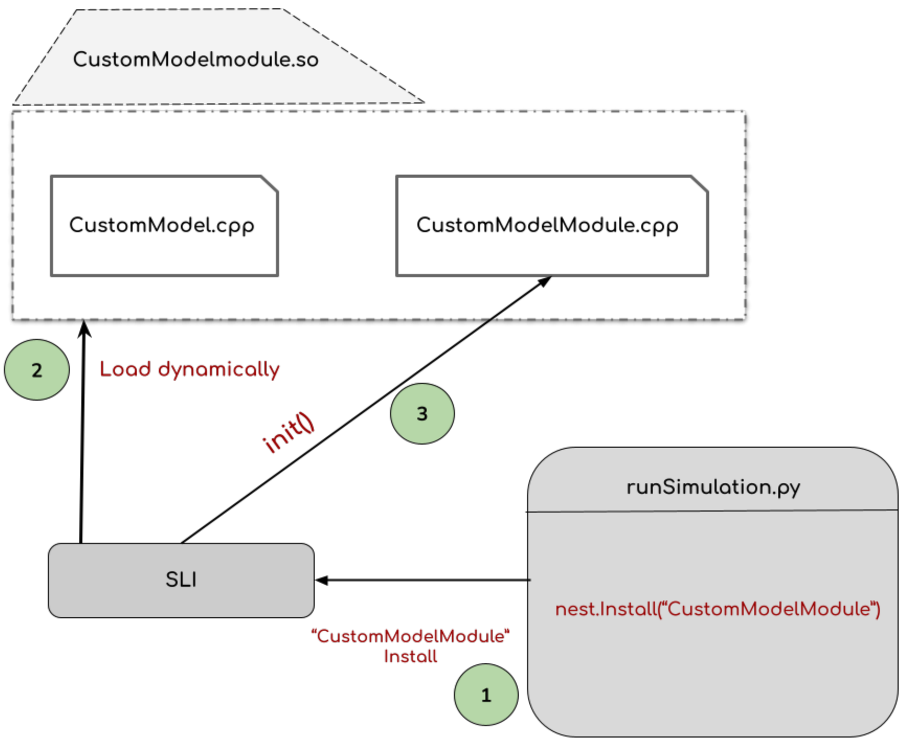

Figure 2.3: Installing an external module: Each generated model comes with two main *CPP* files. The first is implementing the model's logic and functionalities, and the second for registering the model in the *NestKernel*. The process as always starts at the simulation script level, the call to `nest.Install` with the library name as the argument to the function starts the process of registering the model. The function pushes the command name and the library onto the stack that will be processed by the *SLI* routine. *SLI* loaded the library and creates an instance of the `SLIModule` representing the loaded module. The instance initiates then the call to the `init` function that starts the registering workflow of the new generated model in the *NestKernel*.

creating new nodes. Upon requesting the creation of n nodes, the `NodeMananger` calls the `GenericModel` to create an instance of the registered model, which just creates a new empty copy of the model and returns it to the `NodeManager`. **??** depicts the steps involved in creating a new node.

2.3 PyNEST The Simulation Frontend

PyNEST and the *NestKernel* are completely separated and independent modules. The only way for both of them to communicate is through the `SLI` interface. As shown in **??** from (**?**), *PyNEST* is split into two layers. The *low-level API* is responsible for ensuring the smooth communication between *Python* and *SLI* by providing the necessary functionalities for converting data from *SLI* to *Python* and vice versa. The *low-level API* is based on the *Python C API*. It uses mainly three functions to communicate with *SLI*. The `sli_push()` function for pushing the function's arguments of the *SLI* command onto the stack, `sli_pop` for retrieving the returned value of the command from the stack and finally the `sli_run` for executing the command. the `sli_push` and `sli_pop` are also

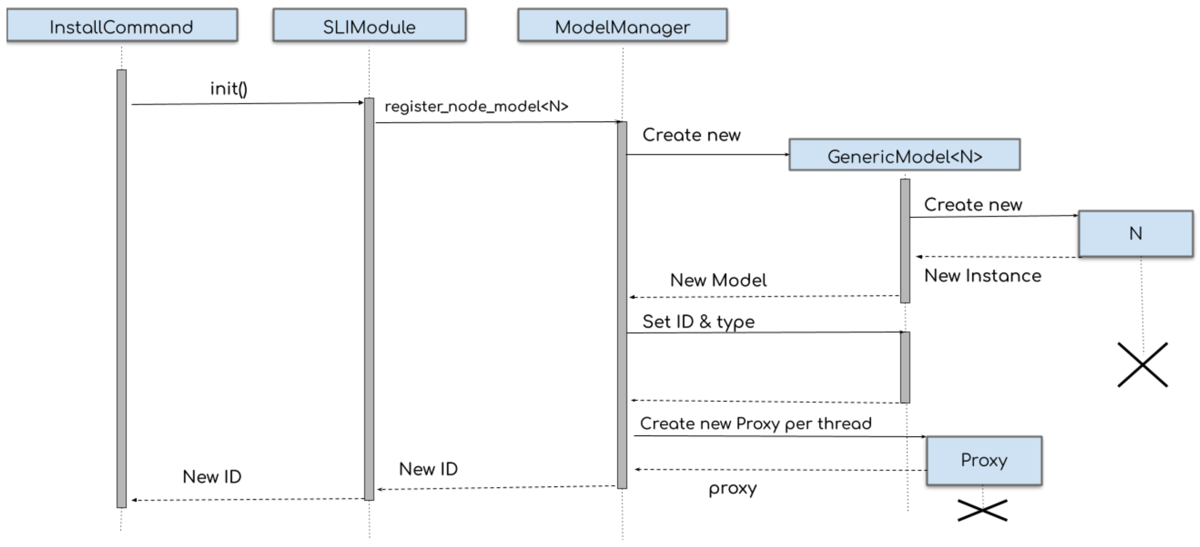

Figure 2.4: Registering new models in the *NestKernel*: The message sequence chart depicts the interaction between the components in the *NestKernel* responsible for registering new custom external models.

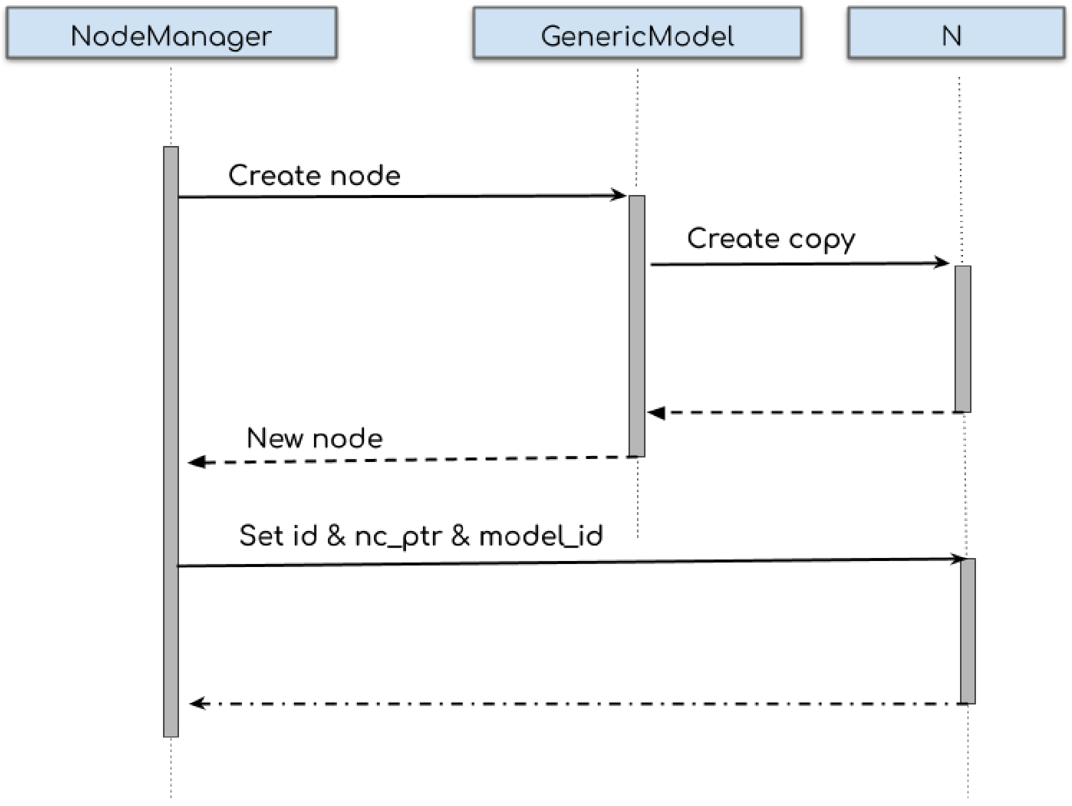

Figure 2.5: Creating new instances of the registered models in the *NestKernel*: The message sequence chart depicts the interaction between the components in the *NestKernel* responsible for creating new instances of any arbitrary registered model.

responsible for converting the data (arguments and results) between both interfaces.

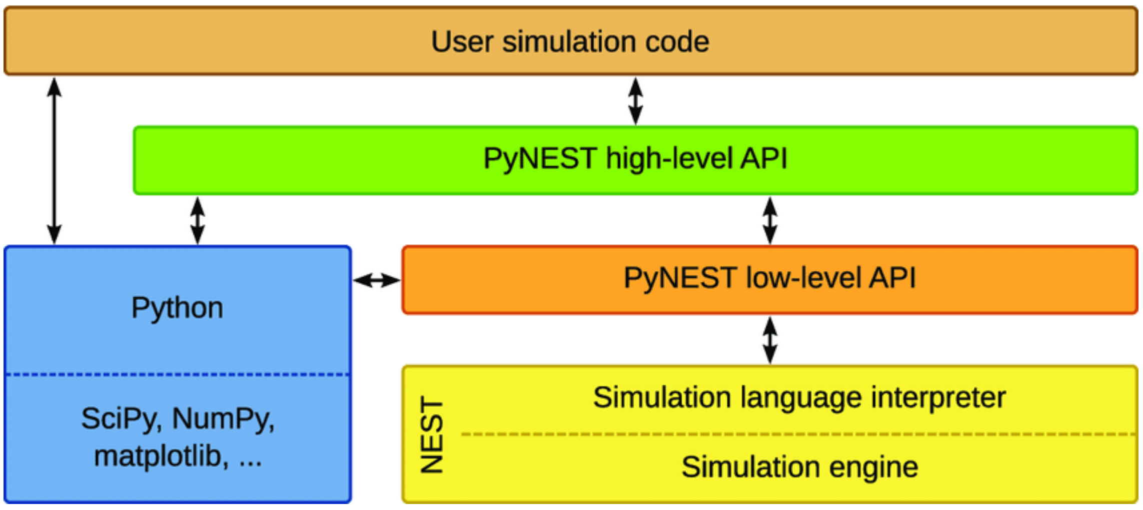

Figure 2.6: The architecture of *PyNEST*: The figure is taken from (**?**), and it shows the main communication levels between the user code and the *NestKernel*. The lowest level is the simulation engine. It is used by the simulation language interpreter and by the *PyNEST* low-level API. The*PyNEST* high-level API uses the low-level API to communicate with the simulation engine. The user's simulation code can use functions from *PyNEST*, from Python, and from its extension modules.

The *high-level API* provides the necessary functions for creating the network and running the simulation. Using the three provided functions in the *low-level API* is not convenient and might not be very intuitive. Therefore, the *high-level API* solves this problem by wrapping the `SLI` functions. It covers all the functionalities in *SLI* by using the functions from the *low-level API*, making all *SLI* commands available.

Each simulation script is based on the *high-level API* functions. The `Create` function is responsible for creating new instances of the registered models. It takes four parameters, the model name, the number of instances, the new model parameters values and lastly a spatial distribution of the nodes. Only the model name is a required parameter, and the others are optional. Omitting the number of instances, the `Create` function returns one single instance. Discarding the model parameters values, the created node will have the default model values and finally, the nodes will not have any spatial distribution if the `spatial` parameter is omitted. The returned value of the function is a `NodeCollection` object. The `NodeCollection` class is a compact representation of the nodes in the `high-level API` interface. Apart from the `Create` function, all functions in *PyNEST* expect a `NodeCollection` object in order to query or modify the created nodes. The `NodeCollection` stores only the `ids` of the nodes it contains, and thus less conversion overhead between *SLI* and *PyNEST*.

An important feature of the `NodeCollection` class, that it only stores a contiguous set of nodes. Thus, creating many nodes only requires storing the position of the first and the last position of the nodes in the list. Thus, the `NodeCollection` always holds a list of contiguous blocks. Since the class is also *iterable*, it supports indexing, splitting and concatenating.

The `NodeCollection` allows also retrieve different values from the created nodes by calling the `get` functions. Depending on the number of provided keys, the `get` function returns either a *list* or *dictionary* containing the values of the given keys. We can also change the values by calling the `set` function. Setting is a bit complicated, as the given value might have different interpretation in the *NestKernel*. Depending on the number of the nodes in the `NodeCollection` and the type of value (*single* or *list*), the setting might affect single nodes or all of them.

Another important feature is copying models. *PyNEST* and `Nestkernel` allow users to copy existing models and assign them a new name and different *default-initial* values. The function responsible for copying the models is called `CopyModel`.

Creating a network is made by calling the `Connect` function that takes two `NodeCollection` objects as the *pre-synaptic (source)* and *post-synaptic (target)* elements with extra parameters specifying the type of the connection and a set of rules defining the connection rules (i.e., controlling degree).

Finally, we have the `Simulate` function with the time t as parameter, which simulates the created network for t milliseconds.

Of course the *PyNEST* module has other important functions to provide, like inspecting the topology of the created network, changing the number of working threads and configuring the *kernel* attributes, but we are mainly interested in the above-mentioned functions that will be used in designing and implementing the architecture for the *JIT* module in the following chapter.

Chapter 3

JIT Module

The focus in this chapter is understanding how to make the concept of the *Just-in-time (JIT)* compilation work without directly affecting the workflow of *PyNEST*. As in any software development process, we start by listing the different requirements that the *JIT* must fulfil, discussing the different ideas about the design decision and finally showing what possible implications does the new software have on the simulation scripts.

3.1 Derived Requirements

In order to have a concise understanding of the *JIT* mechanism and to really evaluate the correctness of the chosen design, we need to have a list of the different requirements that should facilitate the communication between *JIT* and *PyNEST* and cover the different scenarios that may appear during the simulation.

3.1.1 Functional Requirement

Mainly, the *JIT* mechanism should interact with two elements, the *PyNEST* Python module and the simulation script. This interaction puts certain restrictions on how the *JIT* should work, affecting the logic behind writing the simulation scripts and how the *PyNEST* may treat certain models. Therefore, in the following, we list the most important requirements and explain them in details. We start with requirements coming from the simulation script and then those from the *PyNEST* module.

The Simulation Script:

/F1/ The user script should not explicitly import the *PyNESTML* module.

/F2/ Enabling and disabling *JIT* must be done in very straightforward way.

/F3/ Having *JIT* enabled, the simulation script should not vary a lot from the original script with *JIT* being disabled.

/F4/ The user should not explicitly specify the name of the neuron/synapse pairs when connecting the network elements.

/F5/ An instance of the desired model should be available after the call to `nest.Create`.

These five requirements are the base specifications for designing the *JIT*. The first requirement states that the user should not explicitly import the *PyNESTML* module in the script, and thus the *JIT* will decide if it is required to use the code generator or not. The second specification requires that enabling or disabling the *JIT* mechanism must not involve a complex a process and thus must be done in a single a call and the user should not worry about how it is done in the background. The third requirement ensures that there is no huge deviation in the script between having the *JIT* enabled or disabled. The fourth requirement tackles the problem that when co-generating a neuron with a synapse, both the name of the neuron and the synapse will be changed, and therefore the user has to manually use the new names when using the co-generated models in the `nest.Connect` function in *PyNEST*. Retrieving information about the model's instance or setting its attributes after creation is very common in most of the simulation scripts, and that must be granted by the sixth requirement that the model should be available after calling the `nest.Create` function.

PyNEST module:

/F7/ Running the simulation with or without *JIT* should not affect the network behavior.

/F8/ The *JIT* mechanism should not affect the workflow of *PyNEST*.

/F9/ Functions signature of *PyNEST* should not be changed when using *JIT*.

When creating instances of the model, the user can assign random values to some parameters of the model. Thus, when running the simulation script with *JIT*, the result must be correct and *reproducible*, and for that case, the seventh requirement ensures the correctness of the *JIT* module. The eight requirement emphasizes that *PyNEST* should be completely independent and unaware of the existence of the *JIT* module. The last requirement is a bit similar to the seventh requirement, and it states that *PyNEST* should not have its functions signature adjusted to make *JIT* work.

With all these requirements coming from the user level script and *PyNEST*, we came out with two approaches that satisfy all these requirements with each has its own drawbacks which will be discussed in the following section.

3.2 JIT Tool: The Two Approaches

With respect to the given requirements in the previous section, we have thought of two different approaches to fulfilling those specifications, with each having different drawbacks that make us accept only of these approaches to implement.

3.2.1 The Code Static Analysis

The first approach was to design a static analysis tool that examines the simulation script before running it. The tool analyzes the script against a set of defined rules and decides if the code requires some rearrangement of the function calls in *PyNEST*. More importantly, it predicts the expected amount of memory needed to allocate it beforehand.

This solution suffers from different problems. Usually, any simulation script can be split into different phases, as depicted in **??**. Each script starts with creating instances of any arbitrary model known as nodes, then the user may modify the parameters of these nodes. In the case when the user sets the parameters of the model to be generated from a distribution function, he may also want to inspect those chosen values before connecting the nodes creating the network. The Edges of the network may also be assigned a random value and thus, before simulating the whole network, the user can inspect the state of the network before and after stimulating the network by checking the initial set values and observing their changes during the simulation. The tool does not really bring a lot of optimization features, as in the most cases there is not a lot of happening between the creation of nodes and connecting them and the nodes must be directly available after calling the `Create` function. One other problem, allocating memory for future nodes is not really supported by the *NestKernel* which makes the tool useless in that case. Additionally, rearranging the script violates the requirement that the simulation script must not be changed. The rearranging must also satisfy the *reproducibility* requirement (F7), and thus it may not always be possible to control the order of the creation of some nodes, which might make the behavior of the produced script with the code analysis tool deviate from the original simulation script.

The tool also violates the fourth requirement (F4). As analyzing the code only, we can not tell if the *synapse* type involved in creating the connection is a *built-in* synapse or a user's custom defined synapse, and thus the user still needs to explicitly provide the name of the *synapse* coming from the code generator.

These violated requirements are very crucial for persevering the correctness of the *JIT* tool, and therefore we discuss another alternative that might solve these violated specifications.

The alternative approach requires understanding the logic behind the simulation script and splitting it into two parts. The first part for the functionalities that do not require *JIT* and the second part those with *JIT*. Even if we have a solution to solve this partition problem, it is obvious that we are again rearranging the code and ending up in violating the *F8* (reproducibility) and *F3* (deviation from original code) requirements.

3.2.2 The PyNest Wrapper

To again understand what is really happening in the simulation script, we have to revisit the **??**. Apart from simulating an empty network or a network with a single node, each

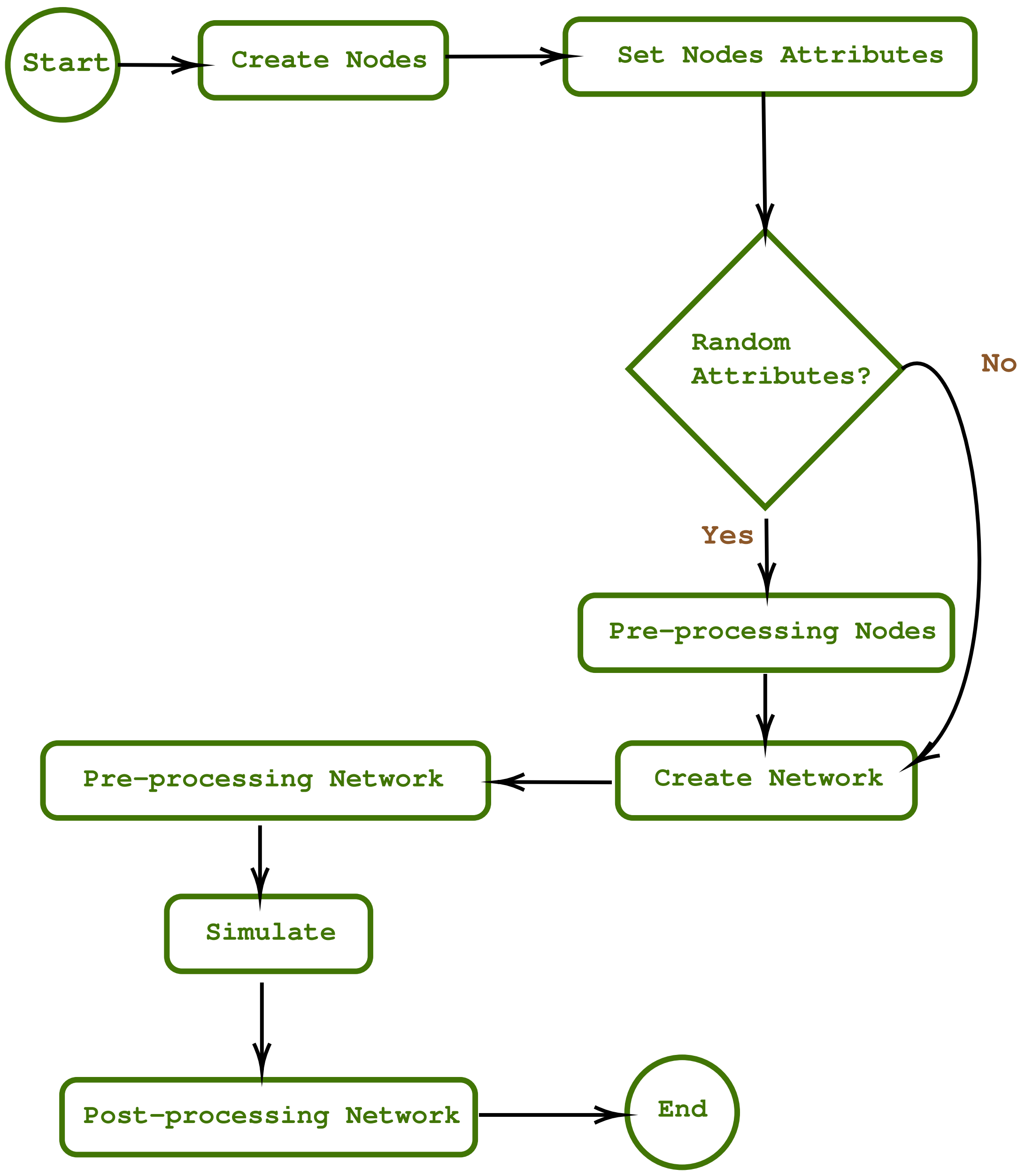

Figure 3.1: The execution workflow of any possible simulation: Any simulation starts by creating any arbitrary number of nodes as instances of the registered models. As models have different attributes, those attributes can be either assigned deterministic or random values. If random values are used, the user can inspect the drawn values and check the current state of the nodes before building the network. Same as the models, creating network can be either deterministic or random, and thus the user can still inspect the topology of the network before running the simulation. Both, the network and the nodes might have different states after running the simulation, and those states might be inspected after the simulation to make conclusion about the network and nodes behavior.

simulation script may contain at least two essential functions, ensuring the creation of the network and stimulating it. The first function is the `nest.Create` function. It takes four parameters, the model's name, the number of instances, the model's parameters and a spatial distribution of the instances. In this section, we are only interested in the first parameter, which is the model's name. After installing a new external model, the *NestKernel* stores these models in an array of models. When creating a new instance of the model, the *NestKernel* searches for the model by its name and once it is found, it creates the required number of instances. A precondition for the `nest.Create` function to work is to have the models installed and registered in the *NestKernel.*

The next function is `nest.Connect` and it takes five parameters. The *presynaptic* nodes as the *source* nodes, the *postsynaptic* nodes as the *target* nodes, a dictionary specifying the connection rules between source and target, another dictionary for specifying the synapse model and its parameters and finally a boolean indicating if the function should return a `SynapseCollection` object to inspect the created connections. For the connection part, when using both external neuron and synapse models, the user must explicitly co-generate the neuron and synapse models together, providing the ports mapping through the code generator before installing them.

The idea of this second approach is to create an interface that controls the *real* workflow of *PyNEST* in the background. It is clear that the model's name is sufficient to locate the model in either as `.nestml` file or in `.so` library and, depending on the situation, we can either install the model or first generate the model's code, build it and then install it. With this new setup, the user can then stop importing the *PyNESTML* module and explicitly calling the `nest.Install` function, as both will be done in the background upon calling the `nest.Create` function. Again, it is important to understand that we are not modifying the *PyNEST* module, we are simply implementing an intermediate interface between the simulation script and the real *PyNEST*, and this new interface should have the same functions available to the simulation script.

This approach works as a *wrapper* around *PyNEST*, as we do not really need to have a wrapper around each *PyNEST* function. It suffices to only choose those *high-level API* functions and intercept their calls and decide either to execute them directly or apply some logic before and after the real function calls or even disable them. Mathematically described, each function f in *PyNEST* is replaced by one of the following execution paths:

$$\begin{aligned}
f &\mapsto \emptyset \\
f &\mapsto f \circ before \\
f &\mapsto f \\
f &\mapsto after \circ f \\
f &\mapsto after \circ f \circ before \\
f &\mapsto g: \ \ \#\text{replace the call of } f \text{ with } g \in \{before, after\}
\end{aligned}$$

The empty set means that we intercept the function call, but we do nothing (the function will be disabled). The functions `before` and `after` refer to a pre-processing and post-processing steps. For example, when creating instances of the model, the `before` function searches for the model and ensures that the real `nest.Create` function will be called without errors. The `after` function returns the *NodeCollection* object containing the instances of the model. In this scenario, the *real* `nest.Create` is called inside `before` function, and it is not really being executed directly after the `before` call.

One issue left concerning the `nest.Connect` call and the co-generation of the neuron and synapse pairs. Recall that the user must explicitly specify the port connecting both the neuron and the synapse, and this information is handled by the code generator. To tackle this issue and remove completely the use of the code generator in the simulation script, we extend the synapse dictionary object in the `nest.Connect` function with a new key referring to the mapping between the neuron and synapse and then the `before` function processes the dictionary and removes the inserted key and forward it to the code generator and once the co-generation finished we call the *real* connect function in *PyNEST*.

The approach might be simple, but we have a lot of things happening behind the scene, as we are introducing new components that facilitate the logic behind intercepting the *PyNEST* functions calls. All those components should be discussed in the subsequent section by introducing their roles and how they can be used to simplify the *JIT* module and ensures the *reproducibility* of the results (*F7*).

More importantly, this solution satisfies all requirements mentioned in the *simulation script*. For the specification from the *PyNEST* module, only the *F9* is slightly violated by extending the `nest.Connection` function by the wrapper.

3.3 JIT Design: The Big Picture

In order for the *JIT* to work with respect to the given requirements, we have come with certain necessary components that facilitate the workflow. As depicted in **??** under the `models`, we have the essential components responsible for representing and managing the models as Python objects before making them available to the *NestKernel*. The `jit_model` stores information about the chosen *NESTML* model by storing its parameters and states and making it possible for the user to query on those attributes after creating the instances of the model. Secondly, we have the `model_query` which is important for searching the model either in the *nestml* files or in the created libraries. Depending on which format, the `model_query` returns an instance of the `model_handle`, which is responsible for making the model available to the *NestKernel*. The `model_handle` knows if the model is coming from a *nestml* file, and therefore can initiate the code generation process and install the model. On the other hand, if the model is coming from an already existing library, it will just install the model. The `model_manager` is responsible for storing all created objects in the *JIT* module, and it is the only way to communicate with the *PyNEST* module. Additionally, the `model_manager` manages the mapping of *IDs* between the created models on the Python side and the real models coming from the *NestKernel* with the help of the `model_indexer`. Finally, under the `models` directory, we have the `node_collection_proxy` that mimics the behavior of the `nest.NodeCollection` in *PyNEST* and it provides all functionalities like the `Setter/Getter` and *Indexing/Slicing*. The need for such a class will be explained later on in this chapter.

Under the `utils` directory, we have the needed components responsible for creating a *lightweight* version of the *nestml* models and controlling the code generation steps. The reason behind using a *lightweight* version should also be clear in the following sections.

Furthermore, we have the `wrapper` directory containing the necessary logic for wrapping the *PyNEST* targeted functions. In the `Wrapper.py` we have the `Wrapper` interface that contains the basic code needed for each wrapper to control the targeted functions and under `Wrappers.py` we have the derived concrete implementation of the `Wrapper`. Any further custom wrapping functionality should be added there, and it will be automatically registered. Details about how we can create a custom wrapper will be given later.

Finally, we have the `nest_manager` which initiates the process of wrapping the *PyNEST* module and calling the implemented wrappers for each target function. The `__init__` is obviously the starting point of the *JIT* that starts the `nest_manager` and overrides the `import` function in Python in order to delegate any further imports from *PyNEST* and add them to the *JIT*.

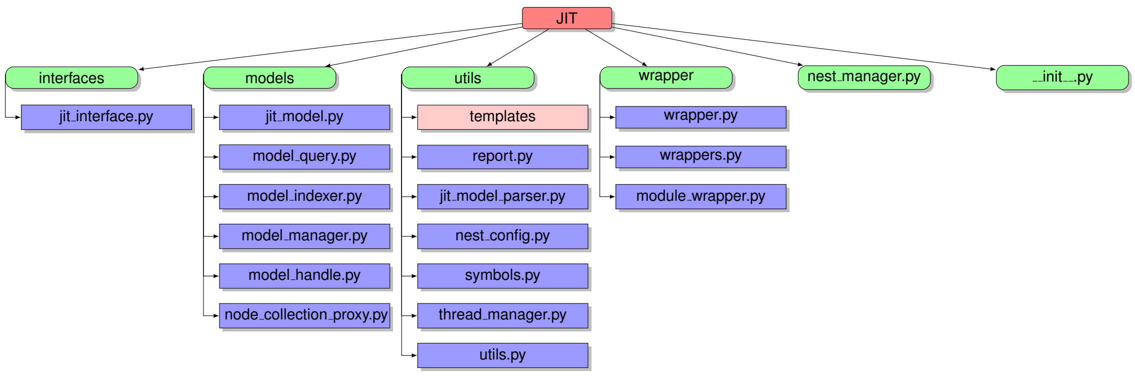

Figure 3.2: The *JIT* components and subdirectories: The project is split into different subdirectories. The *interfaces* folder contains the implemented folders. The *models* folder contains the components responsible for managing the *NESTML* models. The *utils* directory is for providing tools that help with creating the models. The *wraper* folder holds the implemented base `Wrapper` and its derived classes.

In order to have a complete overview of what is happening in the back scene of running *JIT*, we explain the important steps in their correct order of execution. We start explaining the wrapping mechanism and how to implement our custom wrapper. Secondly, we explain how the creation of instances look like with *JIT* enabled, and we discuss then how the `nest.Connect` function might be slightly modified and finally the role of the modified `nest.Simulate`. As the `NodeCollectionProxy` is part of the creation calls, we will introduce it in the same section of the `nest.Create` wrapper.

3.4 JIT Wrappers

In this section, we introduce the *Wrapper* interface responsible for intercepting the function calls in *PyNEST*, and we discuss the derived classes inheriting from this interface allowing to control the workflow of the targeted functions in *PyNEST*.

3.4.1 The Wrapper Base Class

The *Wrapper* interface as depicted in **??** shows the essential items that make *JIT* able to intercept and modify the calls to the *high level API* in *PyNESt*. Each *Wrapper* instance has two boolean attributes. The first attribute specifies if we are wrapping a function or a method. The second is to ignore the function call and skip its execution. By default, the second attribute is set to `false`. As we have mentioned in the previous section, each *Wrapper* object has three main functions to control the interception of the *PyNEST* calls.

the before function is mainly responsible for preprocessing the input values that are given to the *PyNEST* function. Secondly, we have the main, which takes the modified parameters from the before function and executes the real *PyNEST* wrapped function. Finally, we have the after function that takes the output of the main function and modifies it if necessary. In addition to those core functions, we have two important functions. The get_name function that returns the name of the function that we want to wrap, and at last we have the wraps_one that indicates if this Wrapper object is wrapping one or multiple functions. In the case if it is multiple functions, the get_name function must return a list containing the names of these functions. One function left which brings all pieces together, the __wrapper as it is responsible for calling the before, main and after functions.

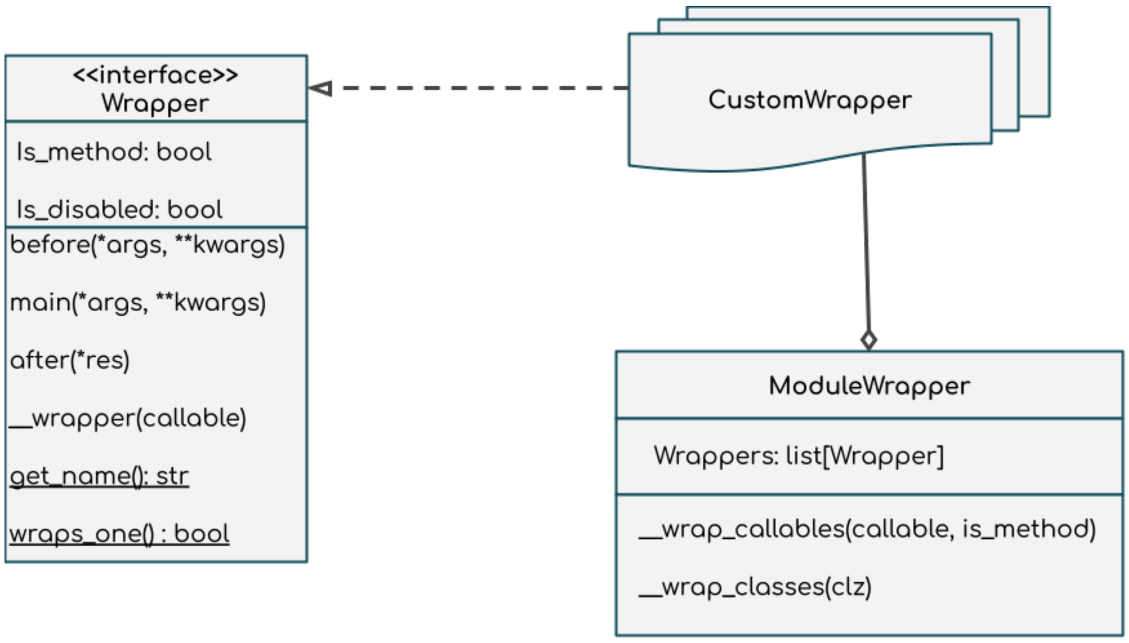

Figure 3.3: The Wrapper interface: The Wrapper is the interface that controls the workflow of its derived classes. All CustomWrapper classes derived from the Wrapper may implement the before, main and after functions. The __wrapper function is responsible for coordinating between the three main function and calling them in sequence. The static function get_name specifies the target function or class in *PyNEST*, and finally the other static function wraps_one indicates how many Wrapper instances should be created. Additionally, we have the ModuleWrapper that is responsible for creating the CustomWrapper instances and matching them with their target functions in *PyNEST*.

The ModuleWrapper is responsible for matching the custom implemented wrappers with the targeted *PyNEST* objects. It iterates overs all functions and classes and replaces them with their corresponding wrappers. finally, we have the CustomWrappers representing the derived classes from the Wrapper interface. Those derived classes can be seen in **??**.

3.4.2 The Wrapper Derived Classes

The *Wrapper* interface on its own does not do a lot for the *JIT* project, and it requires having a concrete implementation for each function we want to control its workflow. Each derived class from the *Wrapper* interface wraps either a *function* or a *class* in *PyNEST*. As shown in **??**, we depict the current implemented *wrappers*. Most of these wrappers use a `helper` class that does all the required logic to make *JIT* work. In the following, we list the implemented wrappers and discuss their usage.

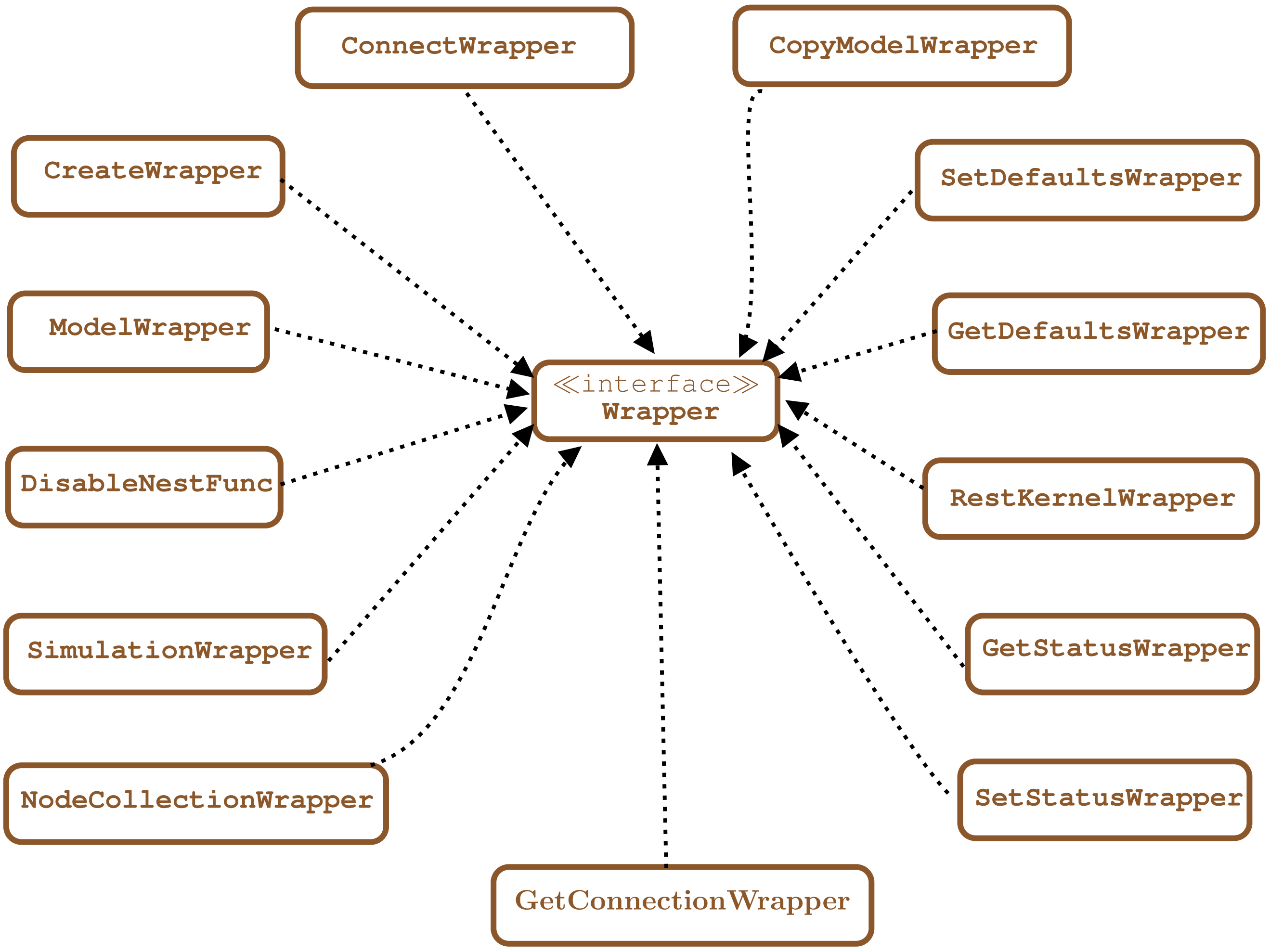

Figure 3.4: The implemented `CustomWrapper` classes derived from the `Wrapper` interface.

- CreateWrapper: responsible for searching for the model in the *file system* and in the *NestKernel* built-in models and creating instances from it.

- ConnectWrapper: responsible for making sure that we are always using the right model while calling the `nest.Connect` function. It may swap network elements depending on some conditions.

- CopyModelWrapper: shares the same functionality as the `CreateWrapper` for searching and installing the model in the *NestKernel*. In addition, it creates a copy of the found model.

- SetStatusWrapper/GetStatusWrapper: set/get the status of the model's instances.

- SetDefaultWrapper/GetDefaultWrapper: set/get the default values of the model. Even if the model is not yet registered in the *NestKernel*, the user can create instances of the copied model.

- GetConnectionWrapper: returns the existing connections between source and target.
- ResetKernelWrapper: resets both the *JIT* stored information about the models and calls the `nest.ResetKerenel` function.

- NodeCollectionWrapper: returns an object containing the real object from the *NestKernel* and the *JIT* instances.

- ModelWrapper: returns all registered models from both the *JIT* and *NestKernel.*

- SimulationWrapper: makes sure that all *JIT* instances are available in the *NestKernel* to start the simulation.

- DisableNestFunc: skips the call of the *PyNEST* function.

In the next step, we explain how to add a new wrapper in *JIT* and explain the workflow behind the most important implemented wrappers. Mainly, the `CreateWrapper`, `ConnectWrapper` and `SimulateWrapper` are of interest, the others should follow the same logic.

3.4.3 Adding a New Wrapper Class

As shown in **??**, wrapping a new function or class in *PyNEST* is very simple. We only need to add the new implementation to the `wrapper.py` Python file containing all implemented wrappers. At this point, there are no restrictions to how the `before-main-after` functions must be implemented. The implemented wrapper has the complete control on how the target *PyNEST* function should behave with respect to *JIT*.

It is also important to know that the developer is not really required to provide an implementation to the `before`, `main` and `after` functions. As for the default behavior of the `before` function will just return the input parameters unchanged, the `main` function calls the real *PyNEST* function and finally the `after` function checks if `main` should return and just returns the same object unchanged, otherwise it does nothing.

Registering a custom *Wrapper* is done automatically upon importing the *JIT* module in the Python script. In The `wrappers.py` file, we have the `install_wrappers` function shown in **??** that is responsible for iterating over all the subclasses implementing the `Wrapper` interface and inserting them in a dictionary, having as *key* the target function name and as *value* the derived wrapper class. The `ModuleWrapper` imports then the dictionary and iterates over its keys, comparing them with *PyNEST* functions and classes. Once a match is found, the `ModuleWrapper` creates an instance of the `CustomWrapper` and extends the *JIT* module with a new function or class having the same name as the one coming from the *PyNEST* module. The new inserted functions or classes in *JIT* are either the original ones or an instance of the `CustomWrapper`.

```python
from jit.wrapper.wrapper import Wrapper

class CustomWrapper(Wrapper):

    def __init__(self, nest_func):
        super.__init__(nest_func, is_method=False, is_disabled=False)

    def __process_input(self, *args, **kwargs):
        # do something with the input
        return args, kwargs

    def __process_output(self, *res):
        # do something with the output
        return res

    def before(self, *args, **kwargs):
        return self.__process_input(args, kwargs)

    def main(self, *args, **kwargs):
        return self.nest_func(args, kwargs)

    def after(self, *res):
        return self.__process_output(res)

    @staticmethod
    def get_name():
        return "nest.{func_name}"

    @staticmethod
    def wraps_one():
        return True
```

Listing 3.1: CustomWrapper example

```python
def install_wrappers():
    sub_classes = Wrapper.__subclasses__()
    to_wrap = {}
    for sub_clz in sub_classes:
        if sub_clz.wrapps_one():
            to_wrap[sub_clz.get_name()] = sub_clz
        else:
            for name in sub_clz.get_name():
                to_wrap[name] = sub_clz
    return to_wrap
```

Listing 3.2: Registering Wrappers

3.5 PyNEST: The Creation Function

Creating new models instances in *PyNEST* is done by calling the create function `nest.Connect(model, n=1, params=None, positions=None)`. The first parameter should indicate the name of the model, the second is for telling *PyNEST* how many instances we want to create, the third is for assigning new values for the attributes of those new instances and finally, the last parameter for indicating the spatial distribution of the created nodes. The return value of this function is a `NodeCollection` object, which is a compact and contiguous representation of the created nodes in the Python side. This `NodeCollection` objects stores only the *first* and *last* *ids* of a continuous space of nodes *ids*. Thus, if we create $n = 10$ instances of any model and only select those odd or even positions, the `NodeCollection` object will have five blocks pointing to the individual odd/even positions. **??** shows an example how the `NodeCollection` may look like. At the start, we have only a single block containing the `first` and `last` *ids* of the nodes. By splitting the created `NodeCollection` into even and odd *ids*, we get two `NodeCollection` objects, each with five blocks. Each of these blocks points only to a single *id*.

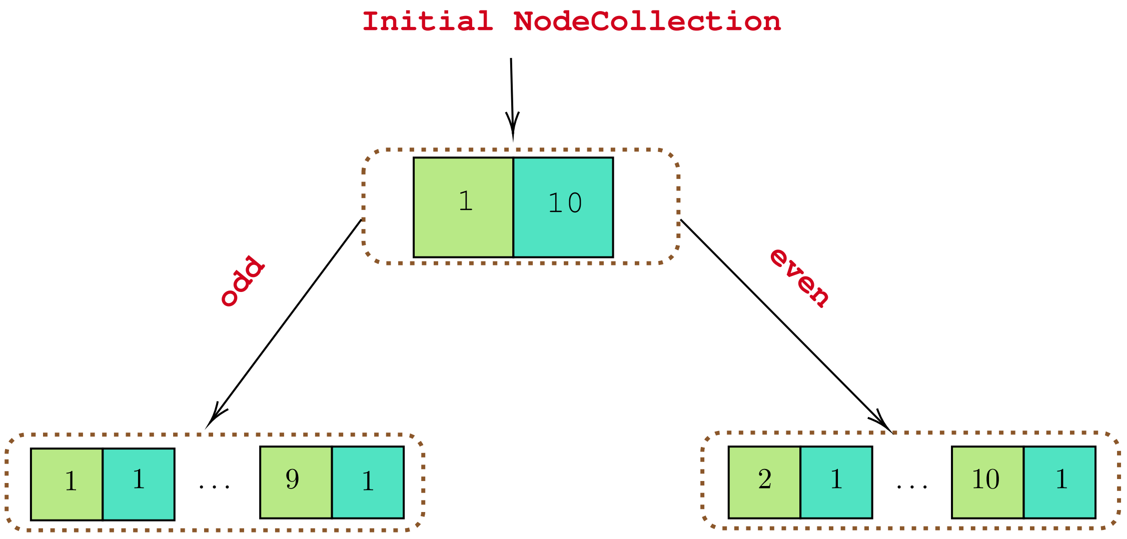

Figure 3.5: Splitting a `NodeCollection` object: The initial `NodeCollection` instance has ten items, and instead of storing all the ten items, the `NodeCollection` stores only the first and the last *ids* in the collection. Splitting the `NodeCollection` into even and odd *ids*, we get two `NodeCollection` object, with each has five items. The separation of *ids* in each split is due to non-contiguous space between them.

In comparison to the initial block, the new blocks store only the `first` and the `size`. This happens because the blocks contain only a single item, but in the case of having blocks larger than one, they will have the same structure as the original block. It is important to know that the size of the `NodeCollection` object is computed by summing the size of each block. Thus, the size of the original `NodeCollection` instance is not one, but instead is $last - first + 1 = 10 - 1 + 1 = 10$.

As depicted in **??**, aggregating `NodeCollection` objects results in having the original `NodeCollection` instance. Mainly, the logic behind the aggregation is to sort everything by *id* at first, then aggregate the *ids* by the model they belong to, merge them into blocks and finally split these blocks into contiguous spaces.

A pre-condition for the `nest.Create` function to work and to return an instance of the `NodeCollection` object is to have the *model* registered in the *NestKenel* (see **??**). The desired *model* is either already installed as a *built-in model* in the *NestKernel* or it must be manually installed by first running the code generation pipeline and calling

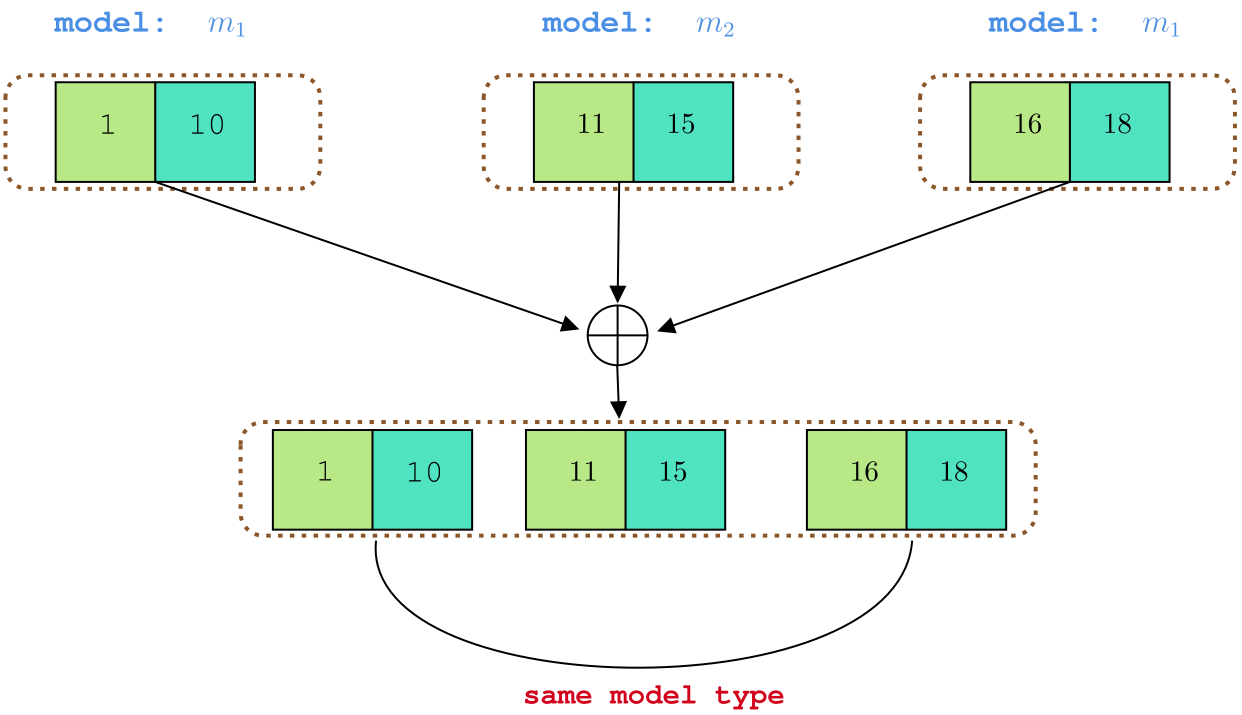

Figure 3.6: Aggregation of `NodeCollection` objects: We start with two models m_1 and m_2. The first model m_1 has two `NodeCollection` objects pointing to it. The first collection has ten items with *ids* between 1 and 10, and the second collection has only three items with *ids* between 16 and 18. The second model m_2 has only one collection with five elements with *ids* between 11 and 15. The aggregation of the three `NodeCollection` instances results in a once `NodeCollection` object that handles the items as a single list, but also it keeps the separation between the models and their *ids*.

the `nest.Install` method. These steps must be manually and repeatedly done in any simulation script trying to use an *external model* that is not already available in the *NestKernel*. To automate this process, the `nest.Create` function must know in advance where to find the given model, register it and make it available for use without having the user explicitly calling the code generator and the `nest.Install` function. For these purpose, we have the `CreateWrapper`, which exactly solves the problem by preparing everything automatically before calling the `nest.Create` function. In the following subsection, we will discuss the work flow of the `CreateWrapper` and discuss the important implemented logic to make it work correctly.

Recall in **??**, precisely in **??**, we have split the code generation process into three steps. The Parsing and validation, the transformation and finally the code generation and building the extension module. The simulation script can be split into three phases. At first, we have to create the nodes and, depending on the simulation scenario, some nodes might need to have some random configurations and thus the user might want to inspect the drawn values and apply some pre-analysis for the current state of these nodes. Secondly, once the nodes are ready, we proceed to the next step and connect the nodes and create the network. Similar to the nodes, the network can be a *random network*, where nodes are connected with probability p. Once the network is set and the random parameters are drawn, the user can inspect the topology of the network. The last step in the simulation script is to simply run the `nest.Simulate(t)` function and wait for the simulation to finish, and afterward the user can then analyze the behavior of the network and check the final reached state of the nodes. It is important to know that the configuration of the nodes and the of the network are two separate things, and they do not influence each other. The `CreateWrapper` makes use of this separation by splitting the code generation pipeline into two steps. The first step uses only the parsing and the transformation steps, whereas the code generation and building the extension module is executed in the background and the `nest.Create` function will not wait for the complete code generation pipeline to finish.

The main tasks for the `CreateWrapper` are to firstly search for the specified model in the *built-in* models, in the existing folders of the built libraries and finally in the folders containing the `.nestml` files. For the first case that the model is a *built-in* model, the `CreateWrapper` simply calls `nest.Create` with the provided parameters. If the model already exists in one of the external libraries, then the wrapper calls the `nest.Install` function to register the model and then calls the `nest.Create`. In the worst scenario case, the model's library does not really exist, and therefore the code generation pipeline must be executed. Recall again that we first reach the transformation phase, then we generate a *lightweight* version of the model by extracting its parameters and states and making them directly available after the create call. The code generation and building of the library is done then in a background process. Only the `nest.Connect` and `nest.Simulate` functions can check the status of the running background process and on success the complete model will be registered, and it will be complete available.

In order for the `CreateWrapper` to hide all these steps for making the model available after the create call, we need to have a class similar to the `NodeCollection` that works on the *lightweight-partial* version of the model. The name of this class is called the `NodeCollectionProxy`, and it will be explained later in this section. Along with this new class, the `CreateWrapper` uses a help class called the `CreateHelper` that takes responsibility for searching for the model and making it either completely or partially ready to query and to modify. All these components will be discussed in the following subsections.

3.5.1 The CreateWrapper

Due to the simple `Wrapper` interface, the `CreateWrapper` as depicted in **??** uses only the `CreateHelper` that encapsulates the logic behind the `nest.Create` function. The `CreateHelper` has a `create` function that has the same signature as the `nest.Create`, but instead of returning a `NodeCollection` object, it returns a `NodeCollectionProxy` object. All of those steps are done in the `before` function, and the `NodeCollectionProxy` object is stored as a member object of the `CreateWrapper` class. As the `main` function is completely ignored in this context, the `before` function returns just an empty tuple and an empty dictionary. Finally, the `after` function returns the stored `NodeCollectionProxy` object, and it does nothing else.

3.5.2 The CreateHelper

The `CreateHelper` is the essential component for controlling the workflow of the `nest.Create` function. In order to understand what is really happening in each of the functions in the **??**, we need to understand the role of each of the import lines. The first

```python
from jit.helpers.create_helper import CreateHelper
from jit.wrapper.wrapper import Wrapper

class CreateWrapper(Wrapper):

    def __init__(self, func):
        super().__init__(func, is_method=False, is_disabled=False)
        self.nodeCollectionProxy = None

    def before(self, model_name, n=1, params=None, positions=None):
        createHelper = CreateHelper()
        self.nodeCollectionProxy = createHelper.Create(
                                    model_name, n, params, positions)
        return (), {}

    def main(self, *args, **kwargs):
        pass

    def after(self, *res):
        return self.nodeCollectionProxy

    @staticmethod
    def get_name():
        return "nest.Create"
```

Listing 3.3: The `CreateWrapper`

line *imports* the `ModelQuery` which responsible for finding the model and returning its *handle* that knows if the model is coming from a *nestml* file or from a library. The second line is for importing the `NodeCollectionProxy` that mimics the functionality of the `NodeCollection`. The `ModelManager` in the third line holds the `nest` module, with which the `CreateHelper` can call the `nest.Create` function. The fourth line contains three important elements for the *JIT* to work. The `JitModel` stores information about the model, like the states and parameters and their default values. The `JitNode` is a compact contiguous representation for the created instances. It stores the name of the model, the *first* and the *last ids* of the instances. Finally, we have the `JitNodeCollection` which is a list of `JitNodes`. In Short, if we have a tree like, the `JitNodeCollection` will be the root, the `JitNodes` are the direct children of the root and the `JitModels` are the leaves. The last import line is the `JitThread` which responsible for running the code generation and the building of the library in the background.

The explanation of the `Create` function will be left as the last function to discuss. We start with the `handle_built_in` method. It is the simplest case, where we just call the real create function from the *PyNEST* module, and then we store the `NodeCollection` returned object in the `NodeCollectionProxy`. An important part of the `CreateHelper` is to keep the order of *ids* consistent. With the help of the `ModelManager`, we can query the last assigned *id* and update it with the size of the `NodeCollection`. Finaly, we store the `NodeCollectionProxy` in an array of nodeCollectionProxies in the `ModelManager`.

The task of the `handle_external_lib` can be split into two parts. The second part is exactly like the `handle_built_in` function. The first part takes care of installing the library.

Now to the most important part, the `handle_nestml` function that is responsible for making the *nestml* model available in the simulation. The process starts by asking the

```python
from jit.models.model_query import ModelQuery
from jit.models.node_collection_proxy import NodeCollectionProxy
from jit.models.model_manager import ModelManager
from jit.models.jit_model import JitModel, JitNode, JitNodeCollection
from jit.utils.thread_manager import JitThread


class CreateHelper:
    def __init__(self):
        # create an empty NodeCollectionProxy instance
        self.nodeCollectionProxy = NodeCollectionProxy()
    def Create(self, model_name, n=1, params=None, positions=None):
        # handle different cases

    def handle_built_in(self, model_name, n=1, params=None, positions=None):
        pass

    def handle_external_lib(self, model_name, n=1, params=None, positions=
None):
        pass

    def handle_nestml(self, model_name, n=1, params=None, positions=None):
        pass

    def handle_jit_model(self, model_name, n=1, params=None, positions=None)
:
        pass
```

Listing 3.4: The CreateHelper

handle to start processing the model by doing the first two steps in the code generation (i.e., Parsing and Transformation). The `ModelHanlde.get_models` function is then called to iterate over the `ASTNeuron` and `ASTSynapse` objects and create the *lightweight* version of the model that has only the *state* and *parameter* blocks of the model. Once the *lightweight* version is returned from the `ModelHanlde.get_models` function, the `handle_nestml` checks then if the provided attributes names and types are correct or not and then gives the control to the `handle_jit_model`, where the `JitNodeCollection` object is created and stored in the `NodeCollectionProxy`. After that, the `handle_nestml` resumes and initiates the thread to work in the background to generate the library code and build it. The `handle_jit_model` can also be executed separately from the `handle_nestml`, as it may happen that the simulation script may request the same model twice before it becomes available, and therefore we do not really have to execute the search for the model and initiate the code generation pipeline, but instead we update the `JitModel` instance by extending its internal structure to store information about the new desired instances.

Finally, we have the `Create` function that calls all the above described methods. It first checks if the `ModelManager` has already seen the `JitModel` with the given name. If it is the case, then the `handle_jit_model` method will be called, otherwise we check if the model is in the *built_in* models and then call the `handle_built_in`. If the model name is neither registered in the `JitModels` nor in the *built_in* models, we check the `ModelHandle` instance if the model is coming from a library or a *nestml* file. For the first case we call the `handle_external_lib` and for the second one we call the `handle_nestml` function. Independent of what was executed, the `Create` function returns a `NodeCollectionProxy` object that can hold both a `NodeCollection` and a

JitNodeCollection.

3.5.3 The NodeCollectionProxy

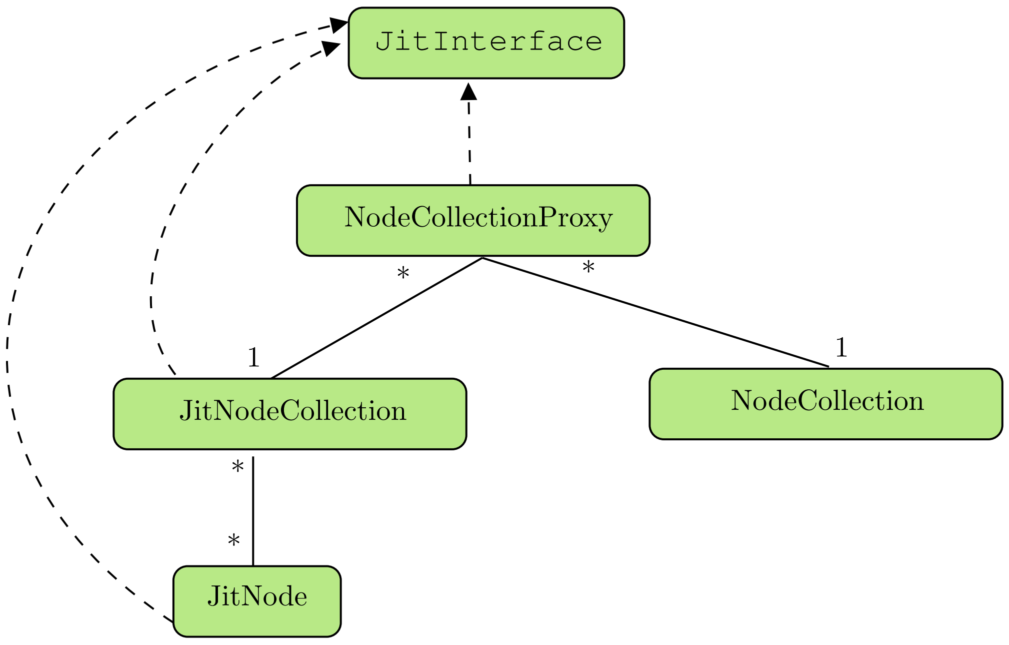

Figure 3.7: The NodeCollectionProxy design: The NodeCollectionProxy can be considered as a *tree-like* structure, and it may have at most two children. The first left child is the JitNodeCollection and the second right child is the nest.NodeCollection class. The JitNodeCollection has also a *sub-tree* attached to it, may have any arbitrary number of children. The children of the JitNodeCollection are instances of the JitNode. The JitInterface provides the common functionalities for retrieving the value of an item, updating it or indexing and slicing the collection. The interface is implemented by the NodeCollection, JitNodeCollection and the JitNode.

The **??** shows the essential elements for making *JIT* work in the background without enforcing any new changes in the simulation script or the way how the user uses the *PyNEST* module. We will explain the items in the figure from the bottom to the top. The JitNode always represents a contiguous space of the model's instances. It stores only the *first* and *last ids* of that contiguous space, and it is the only class that can directly modify the created instances, such that any query or modification to the single instances are processed at the JitNode level. Next we have that JitNodeCollection which a collection of JitNodes. The JitNodeCollection is the equivalent to the NodeCollection as it provides the same functionalities. As the JitNodeCollection holds a list of JitNodes which themselves point to a contiguous space, *slicing* and *indexing* of these classes is a bit complicated and slightly different from the NodeCollection. Finlay, we have the NodeCollectionProxy that has at most two *children*, the NodeCollection and the JitNodeCollection. The class provides the same logic as the NodeCollection, but at the same time allows using the *lightweight* instances after call the nest.Create function.

Since the relation between the classes is a *tree-like* relation, they all share the same logic for *indexing*, *slicing* and retrieving or modifying certain elements. Therefore, those classes implement the same *interface*, preventing us from duplicating the implementation of the mentioned functionalities.

```python
class JitInterface():

    def get_children(self):
        pass

    def get_number_of_children(self):
        pass

    def get_keys(self):
        pass

    def get_relative_pos(self, global_pos):
        pass

    def __iter__(self):
        pass

    def project_dict(self, dict):
        pass

    def get_tuples(self, items):
        pass

    def get(self, *args. **kwargs):
        pass

    def set(self, params=None, **kwargs):
        pass

    def get_node_at(self, global_pos):
        pass

    def __getitem__(self, key):
        pass
```

Listing 3.5: The JitInterface

The `NodeCollection` is simply a list of *ids* pointing to the real instances of the model in the `NestKernel`. Mapping the list structure to a *tree-like* structure requires certain functionalities in the `JitInterface` that allows a smooth conversion.

The `get_children` function returns the direct successors. In the case of the `NodeCollectionProxy`, the direct successors are the `JitNodeCollection` and the `NodeCollection`. For the `JitNodeCollection`, the successors are instances of the `JitNodes`.

The `get_number_of_children` returns the number of successors, so for the `NodeCollectionProxy` that would be at most two, and it is important to note that is different from summing the size of the `JitNodeCollection` with the size of the `NodeCollection`.

The `get_keys` function returns a list of strings containing the model attributes names. The `get_relative_pos` is for converting a global position to a local position in the node. The range of the *global ids* is in the `NodeCollectionProxy` is in $[0,$`len(NodeCollection) + len(JitNodeCollection)`$)$, and in the

`JitNodeCollection` is $[0, \sum_{n \in \texttt{JitNodes}} \texttt{len(n)}]$. The `__iter__` function is for making each of the classes *iterable*.

The `project_dict` is for splitting the dictionary object into sub-dictionary containing the new values to be set for each instance of the model. The other functions are *helper* functions that facilitate the work of the above-mentioned functions.

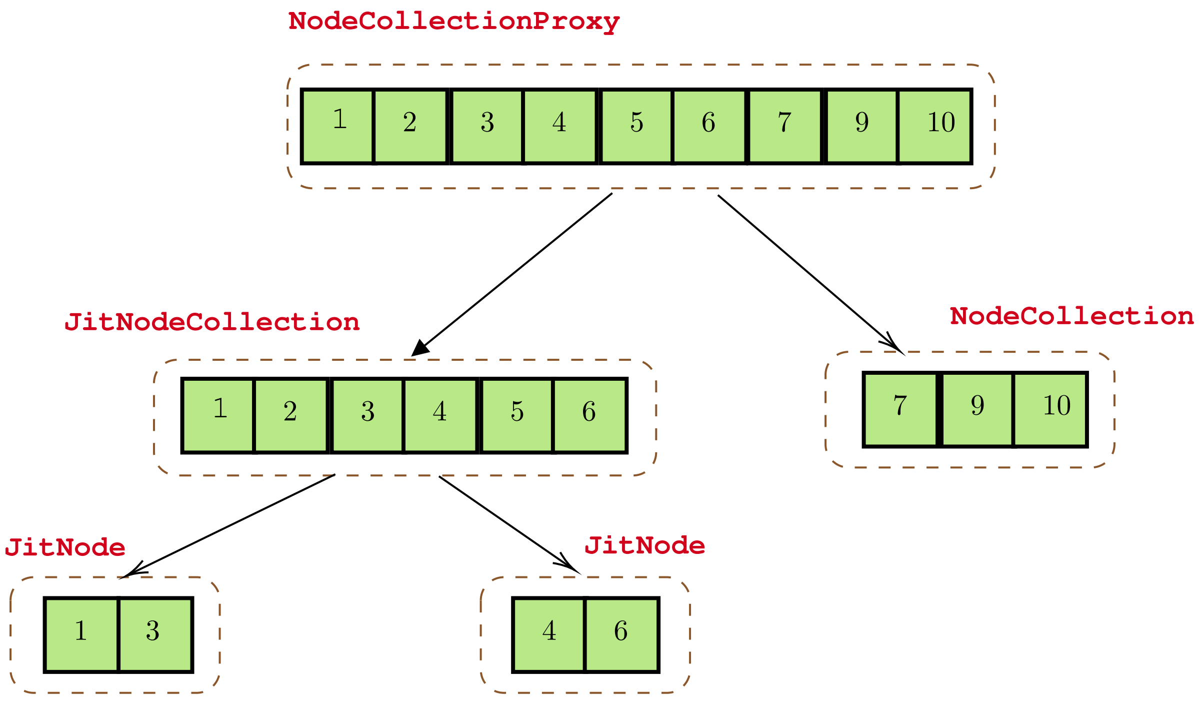

Figure 3.8: From list to *tree-like* structure: The initial `NodeCollectionProxy` has 9 items. The first 6 items are from the `JitNodeCollection` and the last 3 are from the `nest.NodeCollection`. The six items in `JitNodeCollection` are from two different models, and these instances of the models are represented by the `JitNode`. The first `JitNode` spans a contiguous space of *ids* between 1 and 3. The second `JitNode` spans another contiguous space between 4 and 6.

Due to the structure that the `NodeCollectionProxy` is not really a list of element, but instead a *tree-like* structure as depicted in **??**. Retrieving an element from the `NodeCollection` at position i is not very a straightforward task. Since the root has only two children nodes, we check if i is less than the size of the `JitNodeCollection`, otherwise we choose the `NodeCollection`. If i ends up in the range of the `JitNodeCollection`, then we check if i belongs to one of the `JitNode` blocks. Assuming we have n `JitNode` blocks and each block points to k_j instances of any arbitrary model m for $j \in [1, n]$. A `JitNode` block b_l is selected with $l \in [1, n)$ if $\sum_{j=1}^{l-1} len(b_j) \leq i < \sum_{j=1}^{l-1} len(b_j) + len(b_l)$. Once the block b is found, we can convert the `global id` i to the *local id* with respect to the found block b_l. Thus, the *local id* is computed as $local_{id} = i - \sum_{j=1}^{l-1} len(b_j)$. Since the `JitNode` is *indexable*, we can simply get a new `JitNode` having in the `first` attribute the $local_{id}$ as value and for the `last` attribute the value $local_{id} + 1$ and finally the `model_name` will be set to m. The result then for querying the `NodeCollectionProxy` to return the element at the position i will be a new `NodeCollectionProxy` with the size of one, and it contains only a `JitNodeCollection`. The `JitNodeCollection` instance will have a single `JitNode` object pointing to one element. To illustrate an example, let's take the `NodeCollectionProxy` in **??** with ten elements and let $i = 5$. The `JitNodeCollection` has six elements and the `NodeCollection` has only four, and therefore $i = 5$ ends up in the `JitNodeCollection` child. For the first `JitNode` in the `JitNodeCollection` we have three items, in the second one we have two elements, and therefore we have $l \in \{1, 2\}$. For $l = 1$, we have $0 \leq i = 5$ and $5 \not< 0 + len(b_l) = 0 + len(b_1) = 0 + 3 = 3$. For $l = 2$, we $0 + len(b_1) = 0 + 3 \leq 5$ and $5 < len(b_1) + len(b_2) = 3 + 3 = 6$. In that case, it seems that $l = 2$ satisfies the required condition, and therefore $i = 5$ belongs to the second `JitNode` block. The result then will be a `NodeCollectionProxy` with only a `JitNodeCollection` child, which has a single `JitNode = <first= 5 - 4, last=6, name=`m`>`, with four is the index where the second `JitNode` starts and m is the name of the model.

Slicing a `NodeCollectionProxy` is basically similar to indexing, with the only difference that we have a sequence of items $(a_1, \ldots, a_k)$, with k the size of the sequence. The problem with *slicing* is that the items are not necessary the direct successors of each other. In other words, it may be possible that $a_{i+1} = a_i + l$, for $l > 1$, and a_i and a_{i+1} might point to different `JitNodes`. In order to solve that problem, we start by mapping each a_i to the corresponding `JitNode` block, then we group all items together that share the same block. For each group, we convert the *global ids* a_i to the relative *local ids* in each `JitNode` block. By doing so, we should have the following mapping: $(a_1, a_2, \ldots, a_k) \mapsto (P_1, \ldots P_l)$, with $|\bigcup_{i=1}^{l} P_i| = n$. P_i is the partition of the items containing the *local ids* of the original sequence. For each partition P_i, we aggregate the elements inside them in a way that each group is a contiguous group of *ids*. Each of these groups then constructs a new `JitNode` and all the new `JitNodes` will be packed in a new `JitNodeCollection`.

Basically, for *indexing* and *slicing*, we iterate over the *tree-like* structure twice. In the first iteration, we examine the tree from the top to the bottom by mapping the *global ids* to *local ids* and retrieve the correct `JitNode` instances. The second iteration, we traverse the tree from the bottom to top by merging and aggregating the results together until constructing the final `NodeCollectionProxy`. Getting or setting the value of certain *keys* in the `NodeCollectionProxy` undergoes the same logic. By requesting the value of some *keys*, the `NodeCollectionProxy` asks both the `NodeCollection` and the `JitNodeCollection` to retrieve the value of given keys. In return, the `JitNodeCollection` does also the same by asking each of the `JitNode` blocks. The returned value will be either the correct value that is stored in instances of the model or a `NoneType` object, indicating that the model does not have the requested *key* as an attribute. Again, the result from each level will be aggregated and delivered to a higher level in the tree until we reach the root (i.e., the `NodeCollectionProxy`). The *set* function in the `NodeCollectionProxy` has the exact behavior as the *get* function, with the single different that the new values will be split among the children in a way that each child gets a sub-dictionary containing only its proper *keys*.

3.5.4 The JitModelParser

Generating the code for the *lightweight* version of the model undergoes practically the same logic as generating the code for the whole library. The `JitModelParse` is the component responsible for creating the *lightweight* version. The component takes as an input the `ASTNeuron` and returns as an output an executable `C++` code. Mainly, the `C++` code contains the declared *State* block and the *Parameter* block, including the *constructor* that is responsible for initializing the attributes in those blocks. Additionally, some of these attributes might be assigned a random value drawn from a well-supported distribution in the *NestKernel.* The problem here is that the random number generator is purely coming from the `NestKernel` library, and we do not really want to include anything related to the *NestKernel* at this point. Furthermore, it is very important to preserve the order in which the random numbers are drawn to ensure the *reproducibility* of the simulation results.

Luckily, to tackle this problem, the `JitModelParser` extracts the *symbols* referring to the random number generator in the `C++` generated code for the *lightweight* version from the *constructor* and replaces them with a variable. The declaration of this variable will be inserted in the *constructor.* Thus, if we have n statements in the *constructor* that use a random number generator in their expression, then we replace the occurrences of each of these generators with a new variable r_i, for $i \in [1, n]$ and extend the *constructor* with n new parameters in the form of `double` r_i. The `JitModelParse` compiles the *lightweight* version and makes the new class available in the Python interface with the help of the *CPPYY* module (**?**). The `JitModelParse` is also responsible for calling the extended constructor, and provides the value of the new added parameters. It simply calls the *PyNEST* functions responsible for generating the random numbers and passing them to the *lightweight* version class constructor to initialize its attributes in the *State* and *Parameters* blocks.

Unfortunately, the `lightweight` version requires having a duplicate space for the model's attributes. Those coming from the `lightweight` model and the others after completing building the library and creating the real instance of the model. Also, drawing the random numbers is done on a single computing node, and it might influence the results and performance if more computing nodes are used. To tackle this problem and partially solve it, we decided to make the *model* more modular that can be decomposed into independent interchangeable subcomponents. More to the *modularity* will be discussed in the next chapter.

3.6 PyNEST: The Connect Function

Previously, all used neuron-synapse combinations involving synapse models with a dependency on post-synaptic variables, such as spike-timing dependent plasticity (STDP), had to be provided manually to the code generator before running the simulation. This dependency is now hidden in the `ConnectWrapper` that extends the functionality of the `nest.Connect` function and depending on the synaptic connection, the code generation process might be triggered for the second time after being called in the `nest.Create`, but this time with different configurations that allow the code co-generation of the neuron model and its synaptic element.

3.6.1 The ConnectWrapper

The `ConnectWrapper` with its three core functions (i.e., `before-main-after`) intercepts the calls to the `nest.Connect` function, ensuring the correct use of models during building of the network. The `before` functions have mainly the same signature as the `nest.Connect`. It takes a source as the presynaptic part of the connection, a target as the postsynaptic element. Additionally, the function takes two dictionaries, one for specifying the connection specification such as the connectivity rule between the source and the target. The second dictionary is for specifying the synapse model and its properties. The task of the `before` function in the context of building the connections in the network is to make sure that both source and target are the real instances of the models and the `NodeCollectionProxy` contains only a `NodeCollection` whereas the `JitNodeCollection` is empty.

The `before` function starts by checking if either the target (postsynaptic element) is a `NodeCollectionProxy` containing a `JitNodeCollection` child and the synapse model type is an external model. If both requirements are satisfied and, depending on the situation, if both the postsynaptic neuron and the synapse are already existing in a library or coming from a *nestml* files. The `before` function delegate the work to the `ConnectHelper` that takes care of converting the postsynaptic nodes residing in the `JitNodeCollection` to the real instances. Next, we check if the source and target are contained inside a `JitNodeCollection`, which means they are at least two *threads* running and building the code for the source and target neurons models. We then explicitly wait for these two threads, and when they finish without any failure, we install the two libraries built by the threads and convert the `JitNodeCollection` objects from the source and target to the `NodeCollection` objects. Depending on the order of calling the `nest.Connect`, we might need to delete certain connections and replace them with others.

From the *JIT* perspective upon calling the `nest.Create` function, we do not really know what will happen to the model and which connections it will have, and it is only really known when calling the `nest.Connect` function. Thus having a `nest.Connect` function that connects the source and the target with an external synapse model will require deleting connections and replacing them with others. The reason behind the deletion and replacing of the edges is because the target neuron model will be assigned a new name from the code generation pipeline and the actual target model will not be valid anymore. Of course, if we call this particular `nest.Connect` function with the external synapse type as the first function involving that postsynaptic neuron, then no deletion or replacing will be required, and the new neuron model will be directly available to all subsequent calls to the `nest.Connect` function. As the final step, the `before` function returns the converted `JitNodeCollection` to `NodeCollection` together with the other parameters as input parameters to the `main` function in the `ConnectWrapper`. Both the `main` and `after` functions are the default implementation from the `Wrapper`, and therefore the `main` function just calls the real `nest.Connect` function and the `after` does nothing.

3.6.2 The ConnectHelper

The `ConnectHelper` is mainly responsible for checking the state of the threads that are involved in building the library of the models specified in the connect call. It installs the libraries and converts the `JitNodeCollection` to the `NodeCollection`. It handles the use case when the co-code generation pipeline must be triggered again and takes care of deleting and replacing the connections of the involved models.

The `wait_for_threads` method takes as parameters a list of *strings* containing the name of threads that we must wait for. The threads names are basically the names of the model they are building the library for. The function waits for the specified threads and inspects their states once they are done building the library and, depending on the state, we either continue the execution of the ConnectWrapper or throw a failure message and stop the execution of the simulation script. Finally, the function removes the terminated threads from the list of running threads. The list will be either further processed by another connect call or the nest.Simulate function.

Next, we have the install_modules method, and it takes the same parameters as the wait_for_threads function. The function is mainly responsible for installing the libraries, calling the nest.SetDefaults for models that have their default values changed under the provided list of models, coping models by calling nest.CopyModels in case if one of the given models must be copied with different initial values.

One of the important functions in the ConnecetHelper is handling the conversion of the postsynapse neuron when an external synapse model is involved. The convert_post_neuron takes as parameters a NodeCollectionProxy and the name of the synapse model. Depending on the involved elements, we distinguish nine distinct use cases when applying the conversion. Each of the neurons and synapse can be either from a *nestml* file, an *external library* or already are *built_in* models. Therefore, we can build a pair where the first position indicates the source of the neuron and the second is for the source of the synapse model. The first pair is *(nestml, nestml)*, which means that both models are only found in the *nestml* format. Therefore, we have the *handle_nestml_nestml* function that is responsible for starting the code generation phase to generate the code for the new model that exclusively supports the synapse model. The function does not parse again both models, but it retrieves them from the JitModels as both should have been already parsed and validated. The function only triggers the transformation phase, as some attributes from the synapse will be transferred to the neuron model. Once the code generation is completed and the library is installed, the function creates the real instance of the model and returns the new NodeCollection objects to be used further in the main function in the ConnectWrapper. The second pair is then the *(external, nestml)*, which when the neuron model is already exists in some external library and the synapse model is only in a *nestml* format. The function in that case tries to find the *nestml* version of the model, and if it is found, it will then execute the same logic as in the previous pair case. If the *nestml* of the model is not found, then we just throw an error and stop the execution, as it is not possible to bind the synapse with the model without having extra information how to generate the code of the *neuron-synapse* model.

The next pair is the *(built_in, nestml)* case, where the neuron model is a *built_in* model and the synapse from a *nestml* file. This case requires a prior conditions to be satisfied in order to continue the execution of the nest.Connect function. In general, if the target model name is *x* and the synapse name is *y*, and we want to connect with the source and target with the *y*, then the name type of the target model must be an instance of the *x__with_y* and the synapse also will be assigned a new name as *y__with_x*. In general, the synapse and the neuron code are complied and built in the same library, so that if *y__with_x* is installed then also is the *x__with_y* and vice-versa. In this particular case, we search if the synapse *y__with_x* can be found in any of the existing libraries and install it, which should then also register the *x__with_y* neuron model, and then we can continue creating the NodeCollection object.

Finally, we have the *(built_in, built_n)* that both neuron and synapse are already available in the *NestKernel*. As in the previous case, it is required that both the *x__with_y* and the *y__with_x* are registered, or we will just then stop the execution of the simulation script and throw the error that we can not find the models. In all the other remaining cases, we just do nothing and let the `nest.Connect` handle them.

At the last function in the `ConnectWrapper`, we have the `convet_to_node_collection` that converts an instance of the `JitNodeCollection` to a `NodeCollection` and updates the new created instance by the assigned values to the model. It is important to know the difference between the `convet_to_node_collection` and the `convet_post_neuron`, as for the second function is only called to convert the target `NodeCollectionProxy` if there is an external synapse model involved and the first function just converts any `JitNodeCollection` without any prior requirements apart from the model's library being already installed in the *NestKernel*.

3.7 PyNEST: The Simulation Function

The `nest.Simulate` method is intercepted by the `SimulateWrapper` that exactly does the same as the `ConnectWrapper`, but instead of having only models extracted from the source and target in the connect function, we apply the functions for all models. In other words, all the threads until running the *simulate* function must be awaited for, and their states will be examined for any failure and subsequently, all libraries built by the threads will be installed and then all `JitNodeCollection` instances existing in the simulation script will be converted to a `NodeCollection` objects, even those coming from *slicing* and *indexing*. Like any custom implemented `Wrapper`, the `SimulateWrapper` uses the `SimulateHelper` that implements the above-mentioned functionalities.

3.8 Usage

Here we reach in the end of this chapter by showing the new changes that must be applied at the simulation level to adjusted when using the *JIT* module. For the purpose of the following demonstrations, we will use the *STDP windows tutorial* from the official *NESTML* website in order to show the new adjustments made by *JIT* to facilitate the work for the simulation script.

The first changes that are affected by using *JIT* happens at the *import* level in the simulation script. As depicted in the **??** and **??**, we see that only the first three lines involving both the *PyNEST* and the *PyNESTML* modules are completely deleted and replaced with a single import call to the `nest` module from the *JIT* the module. Apart from these changes, the rest of the import lines stay unchanged.

Secondly, in the *NESTML* tutorial, we have the `generate_code_for` in **??** that takes as parameter the synapse model name. The function retrieves the *nestml* file for the given synapse and also the *nestml* file for the `iaf_psc_delta` neuron model and then triggers the co-code generation for both models together providing the `post_ports` that required for connecting the postsynaptic neuron together with the synapse. Finally, once the code generation pipeline finishes, it installs the new library, which makes the specified model available for use in the simulation script and returns the new expected name of the neuron and synapse that both have the *__with_* infix. Luckily, using *JIT* we can simply get rid

```
import nest
from pynestml.frontend.pynestml_frontend import generate_nest_target
NEST_SIMULATOR_INSTALL_LOCATION = nest.ll_api.sli_func("statusdict/prefix ::
    ")

%matplotlib inline
import matplotlib as mpl
mpl.rcParams['axes.formatter.useoffset'] = False
import matplotlib.pyplot as plt
import numpy as np
import os
import re
```

Listing 3.6: The Simulation script imports without JIT

```
from jit import nest

%matplotlib inline
import matplotlib as mpl
mpl.rcParams['axes.formatter.useoffset'] = False
import matplotlib.pyplot as plt
import numpy as np
import os
import re
```

Listing 3.7: The Simulation script imports with JIT

Figure 3.9: The Simulation script imports *JIT*

of this huge function and let it completely take care of the models, and their new names. Even the `nest.Install` function on the line 41 will also be deleted, and only the *JIT* is responsible for calling it.

Finlay, we reach the part where we have to create the nodes and connect them to build the network. Both the figures **??** and **??** share the exact the code, apart from a slight difference in the line 44 when calling the `nest.Connect` function. In the figure **??** we add an extra key in the `syn_spec` indicating the *post connection ports* between the synapse and the post-neuron. The new key then will be removed from the `syn_spec` dictionary and only be used in the code generation pipeline as one of the custom configurations in the *codegen_opt* in *PyNESTML* module. The rest of the dictionary is then forwarded to `ConnectWrapper` to initiate the correct call to the `nest.Connect` function.

In short, the *JIT* design and implementation does not affect the workflow of the simulation script and only slight changes are required to be adjusted without a huge intervention from the user.

3.9 Summary

We started by understanding the working environment between the three main components in this project. The *PyNESTML* module for generating the code of the external modules, *PyNEST* the high level *API* for creating the nodes of the network and connecting them, and finally, we have *NestKernel* as the core of the simulation that manages its execution and delivers results back to the *PyNEST* module. After that, we understood the basic workflow between them, we derived the necessary requirements that should be satisfied by the *JIT* design. Finally, we explained the most important components involved in the *JIT* and how each piece is necessary for intercepting the calls from the *PyNEST* and seamlessly modifying their execution paths, either by adding new functionalities or totally skipping the function and executing something different.

Two major drawbacks in the implemented solution, that the assigned values to the attributes of the model will be stored twice in the memory. The first values come from storing the *lightweight* version of the model, and the second values come after creating the real instances of the model in the `nest.Connect` and `nest.Simulate` calls. The second drawback is triggering the code generation pipeline twice when trying to connect a post-neuron with an external synapse type. To tackle these two problems, we decided to make the neuron models more flexible and modular that can be assembled in the runtime, instead of writing the whole code and compiling it into one single library. But in order to make the `NestKernel` support modularity and make it utilize its potential, we need to implement a new data structure for the models that should make the transition between the normal models and the modular ones more flexible. In the next chapter, we should introduce this new data structure and discuss its strong connection with the modularity.

```python
n_modules_generated = 0
def generate_code_for(nestml_synapse_model: str):
    """Generate code for a given synapse model, passed as a string, in combination with
    the iaf_psc_delta model.

    NEST cannot yet reload modules. Workaround using counter to generate unique names."""
    global n_modules_generated

    # append digit to the neuron model name and neuron model filename
    with open("models/neurons/iaf_psc_delta.nestml", "r") as nestml_model_file_orig:
        nestml_neuron_model = nestml_model_file_orig.read()
        nestml_neuron_model = re.sub("neuron\ [^:\s]*:",
                                     "neuron iaf_psc_delta" + str(n_modules_generated) + ":", nestml_neuron_model)
        with open("models/neurons/iaf_psc_delta" + str(n_modules_generated) + ".nestml", "w") as nestml_model_file_mod:
            print(nestml_neuron_model, file=nestml_model_file_mod)

    # append digit to the synapse model name and synapse model filename
    nestml_synapse_model_name = re.findall("synapse\ [^:\s]*:", nestml_synapse_model)[0][8:-1]
    nestml_synapse_model = re.sub("synapse\ [^:\s]*:",
                                  "synapse " + nestml_synapse_model_name + str(n_modules_generated) + ":", nestml_synapse_model)
    with open("models/synapses/" + nestml_synapse_model_name + str(n_modules_generated) + ".nestml", "w") as nestml_model_file:
        print(nestml_synapse_model, file=nestml_model_file)

    # generate the code for neuron and synapse (co-generated)
    module_name = "nestml_" + str(n_modules_generated) + "_module"
    generate_nest_target(input_path=["models/neurons/iaf_psc_delta" + str(n_modules_generated) + ".nestml",
                                     "models/synapses/" + nestml_synapse_model_name + str(n_modules_generated) + ".nestml"],
                         target_path="/tmp/nestml_module",
                         logging_level="ERROR",
                         module_name=module_name,
                         suffix="_nestml",
                         codegen_opts={"nest_path": NEST_SIMULATOR_INSTALL_LOCATION,
                                       "neuron_parent_class": "StructuralPlasticityNode",
                                       "neuron_parent_class_include": "structural_plasticity_node.h",
                                       "neuron_synapse_pairs": [{"neuron": "iaf_psc_delta" + str(n_modules_generated),
                                                                   "synapse": nestml_synapse_model_name + str(n_modules_generated),
                                                                   "post_ports": ["post_spikes"]}]})

    # load module into NEST
    nest.ResetKernel()
    nest.Install(module_name)

    mangled_neuron_name = "iaf_psc_delta" + str(n_modules_generated) + "_nestml__with_" + nestml_synapse_model_name + str(n_modules_generated) + "_nestml"
    mangled_synapse_name = nestml_synapse_model_name + str(n_modules_generated) + "_nestml__with_iaf_psc_delta" + str(n_modules_generated) + "_nestml"

    n_modules_generated += 1

    return mangled_neuron_name, mangled_synapse_name
```

Listing 3.8: The code-generation phase

```python
def run_network(pre_spike_time, post_spike_time,
                          neuron_model_name,
                          synapse_model_name,
                          resolution=1., # [ms]
                          delay=1., # [ms]
                          lmbda=1E-6,
                          sim_time=None,  # if None, computed from pre and post spike times
                          synapse_parameters=None,  # optional dictionary passed to the synapse
                          fname_snip=""):

    nest.set_verbosity("M_WARNING")
    #nest.set_verbosity("M_ALL")

    nest.ResetKernel()
    nest.SetKernelStatus({'resolution': resolution})

    wr = nest.Create('weight_recorder')
    nest.CopyModel(synapse_model_name, "stdp_nestml_rec",
                {"weight_recorder": wr[0],
                 "w": 1.,
                 "delay": delay,
                 "d": delay,
                 "receptor_type": 0,
                 "mu_minus": 0.,
                 "mu_plus": 0.})

    # create spike_generators with these times
    pre_sg = nest.Create("spike_generator",
                         params={"spike_times": [pre_spike_time, sim_time - 10.]})
    post_sg = nest.Create("spike_generator",
                          params={"spike_times": [post_spike_time],
                                  'allow_offgrid_times': True})

    # create parrot neurons and connect spike_generators
    pre_neuron = nest.Create("parrot_neuron")
    post_neuron = nest.Create(neuron_model_name)

    spikedet_pre = nest.Create("spike_recorder")
    spikedet_post = nest.Create("spike_recorder")
    #mm = nest.Create("multimeter", params={"record_from" : ["V_m"]})

    nest.Connect(pre_sg, pre_neuron, "one_to_one", syn_spec={"delay": 1.})
    nest.Connect(post_sg, post_neuron, "one_to_one", syn_spec={"delay": 1., "weight": 9999.})
    nest.Connect(pre_neuron, post_neuron, "all_to_all", syn_spec={'synapse_model': 'stdp_nestml_rec'})
    #nest.Connect(mm, post_neuron)

    nest.Connect(pre_neuron, spikedet_pre)
    nest.Connect(post_neuron, spikedet_post)

    nest.Simulate(sim_time)

    # rest is retrieving results from spikes
    ...
```

Listing 3.9: Creating the network without *JIT*

```python
def run_network(pre_spike_time, post_spike_time,
                          neuron_model_name,
                          synapse_model_name,
                          resolution=1., # [ms]
                          delay=1., # [ms]
                          lmbda=1E-6,
                          sim_time=None,  # if None, computed from pre and post spike times
                          synapse_parameters=None,  # optional dictionary passed to the synapse
                          fname_snip=""):

    nest.set_verbosity("M_WARNING")
    #nest.set_verbosity("M_ALL")

    nest.ResetKernel()
    nest.SetKernelStatus({'resolution': resolution})

    wr = nest.Create('weight_recorder')
    nest.CopyModel(synapse_model_name, "stdp_nestml_rec",
                {"weight_recorder": wr[0],
                 "w": 1.,
                 "delay": delay,
                 "d": delay,
                 "receptor_type": 0,
                 "mu_minus": 0.,
                 "mu_plus": 0.})

    # create spike_generators with these times
    pre_sg = nest.Create("spike_generator",
                         params={"spike_times": [pre_spike_time, sim_time - 10.]})
    post_sg = nest.Create("spike_generator",
                          params={"spike_times": [post_spike_time],
                                  'allow_offgrid_times': True})

    # create parrot neurons and connect spike_generators
    pre_neuron = nest.Create("parrot_neuron")
    post_neuron = nest.Create(neuron_model_name)

    spikedet_pre = nest.Create("spike_recorder")
    spikedet_post = nest.Create("spike_recorder")
    #mm = nest.Create("multimeter", params={"record_from" : ["V_m"]})

    nest.Connect(pre_sg, pre_neuron, "one_to_one", syn_spec={"delay": 1.})
    nest.Connect(post_sg, post_neuron, "one_to_one", syn_spec={"delay": 1., "weight": 9999.})
    nest.Connect(pre_neuron, post_neuron, "all_to_all", syn_spec={'synapse_model': 'stdp_nestml_rec', "post_ports": ["post_spikes"]})
    #nest.Connect(mm, post_neuron)

    nest.Connect(pre_neuron, spikedet_pre)
    nest.Connect(post_neuron, spikedet_post)

    nest.Simulate(sim_time)

    # rest is retrieving results from spikes
    ...
```

Listing 3.10: Creating the network with JIT

Chapter 4

Vectorization

To mitigate the storage of duplicated data in both the *PyNEST* and *NestKernel* sides, we thought about decomposing the models into segments that are simple to generate from the perspective of the code generator and yet independent, which makes it more efficient in the *JIT* context. In computer science, we refer to this decomposition as *modularity*. The idea behind it is to consider the model as an aggregation of blocks, and these blocks are assembled during the progress of the simulation script upon calling the `nest.Create`, `nest.Connect` and `nest.Simulate` functions. Each model will start with a *dummy* instance of the `Node` class in the *NestKernel* that only has pointers to these blocks, which they will be assigned separately and independently during the simulation. Such blocks may extend the *dummy* instance by new attributes like new *Parameters* and new *States*, or even new functions related to updating the state of the model, delivering emitted events and registering connections. Basically, a valid perfect partition of the model into segments is the partition that aligns with the `nest.Create`, `nest.Connect` and `nest.Simulate`. When calling the `nest.Create`, we only require the *Parameter* and *State* blocks to be assigned to the model. On `nest.Connect`, we extend the model by new functions that support the connectivity of the neuron with the synapse. Finally, by calling `nest.Simulate`, we append the `update` block. This partition might not be always perfect and depending on the chosen model, we may require a different segmentation and a more detailed analysis to create the blocks.

A problem might rise with this approach is that when creating n instances from the model. In the context of modularity and as depicted in the **??** in the left part, it means that each *State* block must be constructed separately, and each block must be assigned to each instance of the model. So assuming the time for creating a *State* block takes c times and biding each instance of the model with the block takes a times and l is the time needed for creating each node separately. Therefore, if we have n instances, then overall we will spend $n \cdot (c + a + l)$ connecting each block with its corresponding instance.

Here where the *Vectorization* comes into play. Instead of having n instances of the *State* block, we only use one block that is *shared* among all other instances of the model. Thus, the *State* block is now vectorized, which means it has a vector of attributes where each entry in the vector corresponds to an instance of the model at the position i. Again, if we have n instances then the overall time needed for creating the instance is $n(\cdot a + l) + c + r$, where r is the overhead for resizing the vector and allocating space for it. It is also possible to reduce the $n \cdot a$ part in which we just have a central entity that has direct access to the blocks and instances of the model. Therefore, the a term will completely vanish, and the overall time required for the `nest.Create` to finish will be $\approx n \cdot l + c + r$. In **??**, we illustrate the best way how the modularity and vectorization may work together to achieve a simple and flexible data structure for handling models in the *NestKernel*. The *central entity* in this context may be the `Model` class in the `NestKernel`, and it only exists once per thread. The instances of the `Node` class are depicted in circles, and they all share the *central entity* which has direct access to the *State*, *Parameter* blocks. Each node has a `local_id` and with it only it can retrieve information about the current stored values of the *State* and the *Parameter* at the corresponding positions.

In this thesis, we will only focus on implementing the *Vectorization* concept for the models and show that it is possible to integrate the *Vectorization* design in the *NestKernel* without drastically breaking its workflow.

In this chapter, we introduce a naive solution and discuss its drawbacks, and then we show a better solution that respects certain requirements coming from the *NestKernel* and finally discuss the current limitations and how is it possible to mitigate them.

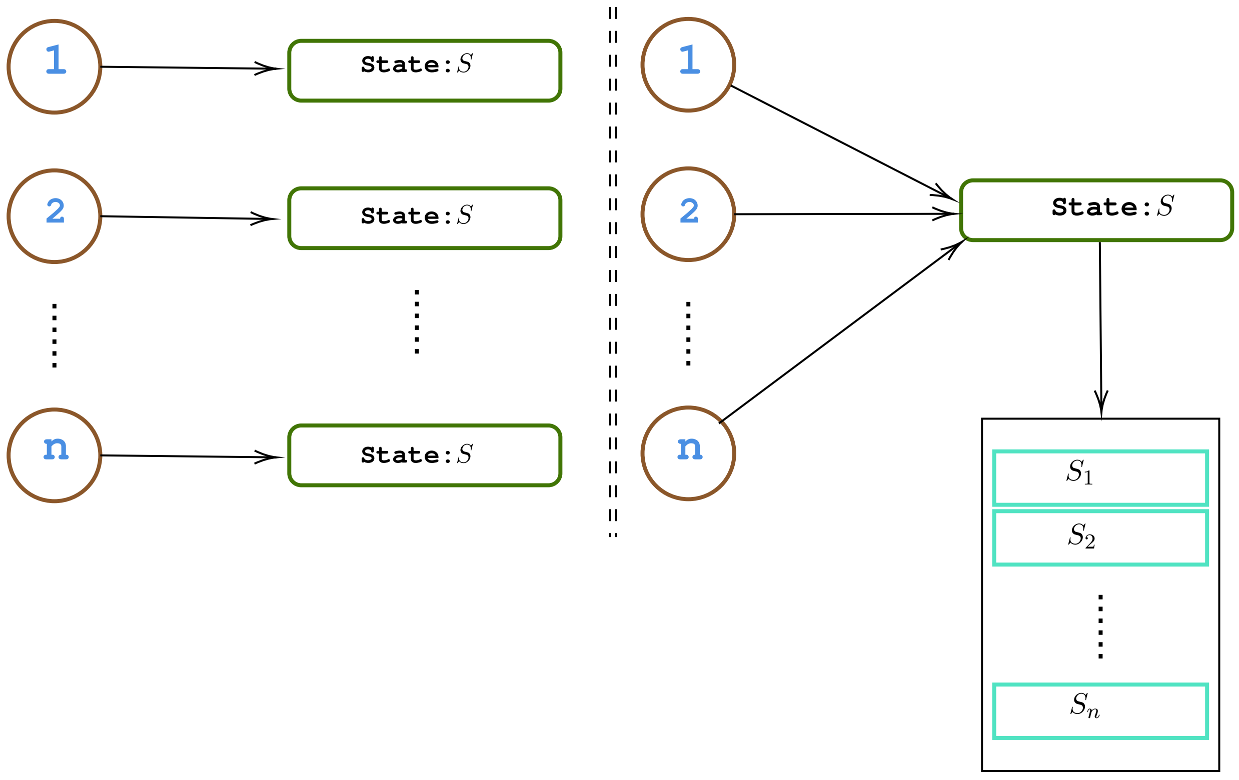

Figure 4.1: Modularity and Vectorization: The left part represents the current implementation of the models in the *NestKenel*. We have n instances of the model and each has its own *state* block with an attribute S. In the right, we have the suggested idea using *vectorization*. All instances share the same *state* block, and instead of having n separated attributes, the attributes are aggregated in a single array with size n.

4.1 The Data Structure

Vectorization in a plain language is the process of rewriting an instruction that is repeating n times over single entities to an instruction that is processing k entities simultaneously.

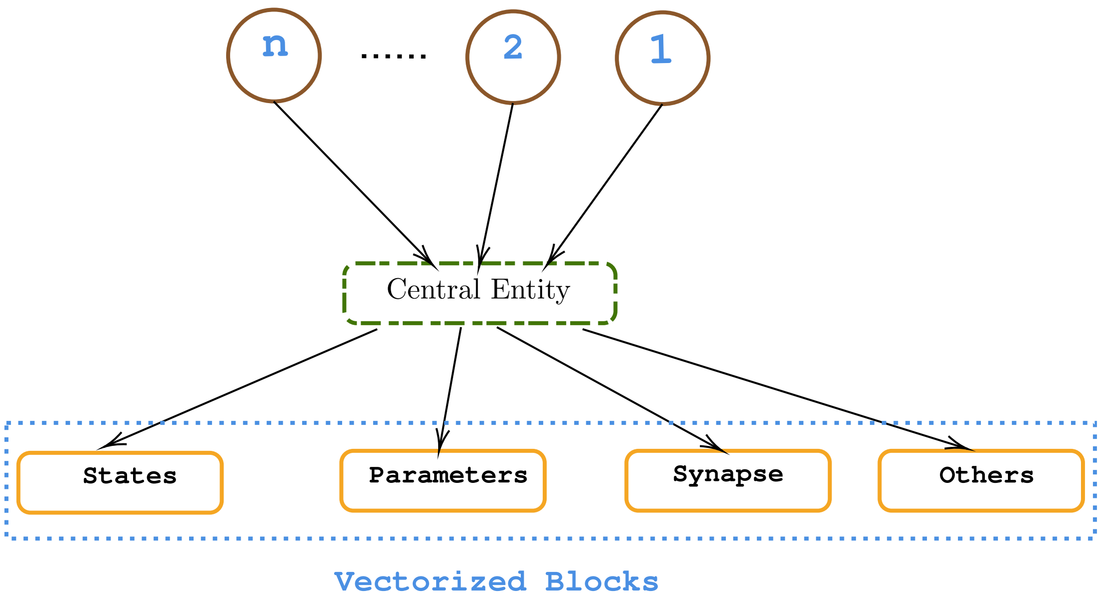

Figure 4.2: Modularity with a central entity: The model is segmented into different blocks that are assembled together during the simulating. The nodes share the *central entity* that manages the state of the blocks. Each of the blocks support vectorization and each of the attributes has a vector form with the size equal to the created population in the network from the specified model.

Usually, the term *vectorization* overlaps with the *SIMD* term. The *SIMD* is short for *single instruction/multiple data*, and it refers to the capability of the hardware components to perform the same operation on multiple data concurrently. The **??** illustrates a simple example of a *SIMD* executing four operations in parallel. Another term strongly related to high performance computing and optimization is the term *speedup*. The term *speedup* in the context of *vectorization* is the ratio of time required by the *non-vectorized* operation $T(1)$ to the time required by the *vectorized* operation, executing k operations at the same time noted as $T(k)$. The formula of the *speedup* is the defined as

$$S(k) = \frac{T(1)}{T(k)}$$

. Assuming that a single instance of the operation takes t units to complete and also neglect the overhead of memory access, then applying the operation on n elements we have $T(1) = n \cdot t$ and $T(k) = \frac{n}{k} \cdot t$, which should imply that

$$S(k) = \frac{T(1)}{T(k)} = \frac{n \cdot t}{\frac{n}{k} \cdot t} = k.$$

A *speedup* of k means that we load k elements in the register of the processor and execute the operation k times in parallel. In reality, the larger k gets, the less speedup we can reach, as it strongly depends on the bandwidth of transferring the data from the main memory to the register and depending on the *CPU* design and architecture, we may end up in a memory bandwidth issues, and therefore we have to be careful in which extent we should use *Vectorization*.

In order to exploit the performance of *SIMD* instructions, we have to use an adequate data structure that is more cache friendly and works perfectly well with the *vectorization*.

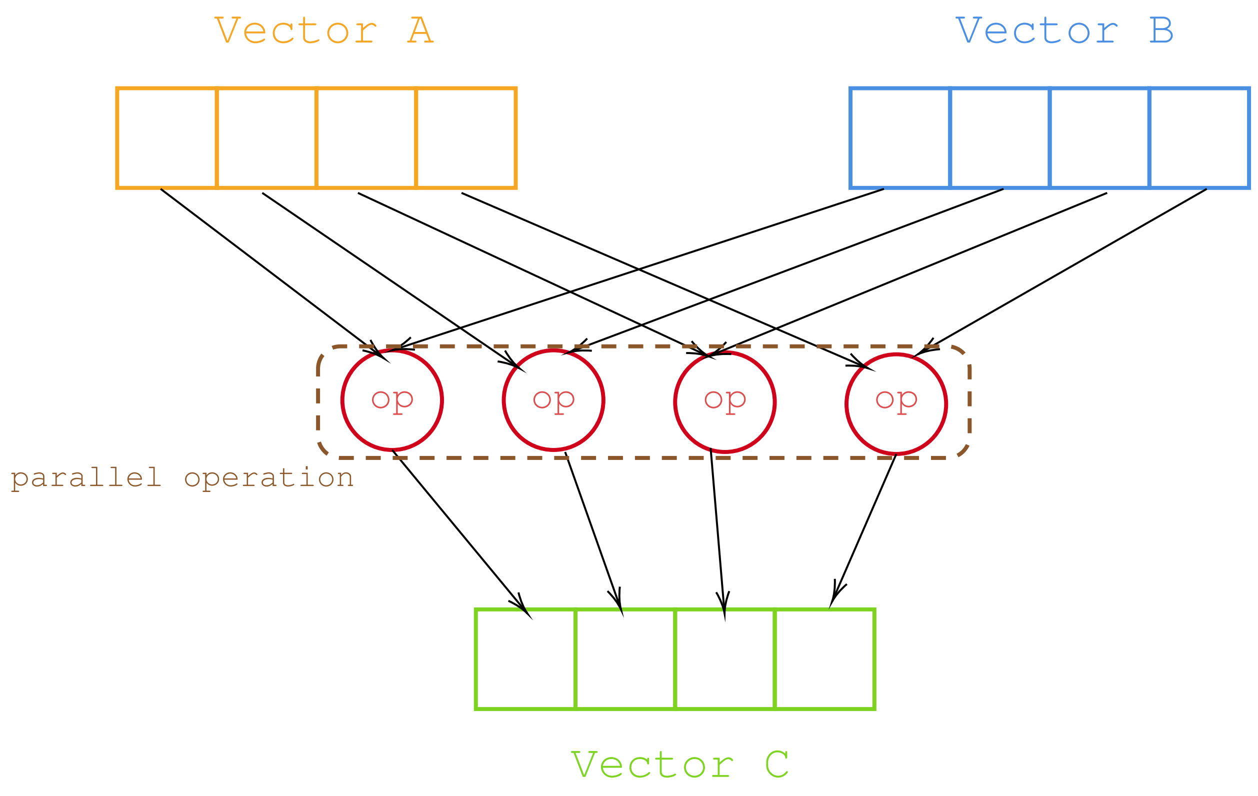

Figure 4.3: Example of SIMD operation: A parallel operation is applied to the single entries of the *Vector A* and *Vector B*. Instead of applying the operation in sequences by iterating over the items, a modern processor with SIMD instructions can execute the operation for k entries in parallel.

In **??** and in **??**, we depict the difference in performance between two data structures and explain why we need to use the one that is more cache friendly.

```
struct Point
{
    double x;
    double y;
    double z;
}

struct Point pts[N];

for(int i = 0; i < N; i++)
{
    pts.x[i] = update_x()
}
```

Listing 4.1: Array of Structure (AoS)

```
struct Point
{
    double x[N];
    double y[N];
    double z[N];
}

struct Point pts;

for(int i = 0; i < N; i++)l
{
    pts.x[i] = update_x()
}
```

Listing 4.2: Structure of Arrays (SoA)

The first data structure in **??** is known as the *Array of Structure (AoS)*. It is simply an array containing instances of a *struct* object in *C++*. The plain representation of the array in the memory is a sequence $(x_1, y_1, z_1, x_2, y_2, z_2, \ldots, x_n, y_n, z_n)$. The second data structure in **??** is basically a rearrangement of the first representation. We still have a sequence, but the position of the items is different. The representation is knows as the *Structure of Arrays (SoA)*, where we have all $(x_i)_{i\in[1,n]}$, $(y_i)_{i\in[1,n]}$ and $(z_i)_{i\in[1,n]}$ close to each other, creating a contiguous space for accessing the elements x_i and x_{i+1}. Mainly, the sequence is given as $(x_1, x_2, \ldots, x_n, y_1, y_2, \cdots y_n, z_1, z_2, z_n)$. For both configurations, we split the sequences in a group of k elements and when accessing an item in one of the group, the whole group is transferred from the main memory to the registers. In computer science, the group of k elements is called a cache line and when an item is requested we first check if the item is already loaded in the register. If the item exists already, then we have a *cache hit*, otherwise, we have a *cache miss* and the item must be retrieved from the main memory.

We assume now then that the cache line loads three entries from the main memory to the registers. Thus, reading x_1 in the *AoS* data structure loads also y_1 and z_1. Whereas reading x_1 in *SoA* also loads x_2 and x_3. Both the **??** and **??** have the same *for-loop* that updates only of the x elements in each of the arrays. In the *AoS* configuration, each update causes a *cache miss*, and for n iterations we have n *cache misses* in total. Whereas in the *SoA* configuration reading x_1, also loads x_2 and x_3, and therefore when reading three items we have only one *cache miss* and two *cache hits*. By updating n elements, we have $\frac{N}{3}$ *cache misses* and $N \cdot \frac{2}{3}$ cache hits. In general, for an arbitrary k, we will have $\frac{N}{k}$ *cache misses* and $N \cdot \frac{k-1}{k}$ cache hits. In the following, we define the *hit rate* β as the percentage of memory accesses that can be resolved from cache because there was a recent load to the same address and the *miss rate* as $(1 - \beta)$. Additionally, we have T_m the time required to access the main memory and T_c the time required to access the cache. Further, we define the *average access time* as $T_{av} = \beta \cdot T_c + (1 - \beta) \cdot T_m$ and $\tau = \frac{T_m}{T_c}$ as the speedup due to using caches, then the *average access performance gain* can be expressed as

$$G(\tau, \beta) = \frac{T_m}{T_{av}} = \frac{\tau \cdot T_c}{\beta \cdot T_c + (1 - \beta) \cdot \tau T_c)} = \frac{\tau}{\beta + \tau(1 - \beta)}$$

As β only takes values between $[0, 1]$, we easily see that for *beta* $= 1$, the *average access performance gain* $G(\tau, \beta) = \tau = \frac{T_m}{T_c}$, which the maximum speedup that can be reached in the system. For $\beta = 0$, $G(\tau, \beta) = 1$, which corresponds to a speedup of one and therefore no utilization of the cache access at all. The *AoS* corresponds to the case where $\beta = 0$ and the *SoA* for the case where $\beta \approx 1$.

It should be clear by now why we should prefer the *SoA* data structure over the *AoS*. The next step will be how to adjust the *NestKernel* and make it support *Vectorization* without deeply affecting its workflow, and make the transition to the *vectorized models* as smooth as possible.

4.2 The Naive Solution

In **??**, we showed the main important components for registering new external models and how to create independent instances from them. As a central component in the *NestKernel*, we have the `Node` class as the *abstract class* providing the necessary functions required for the simulation to work. The `Model` and the `GenericModel` manage the implemented models and act as a *factory* for creating instances of the derived classes from the `Node` class. Finally, we have the `ModelManager` that is responsible for registering new models and making them accessible during the simulation and the `NodeManager` with the role of managing the states of the created `Node` instances.

A direct naive solution to integrate *Vectorization* in *NestKernel* is to replace all fields in the `Node` class and its derived classes with `vector<double>` instead of having single double attributes. But with the current functions signatures in the Node class we can nor update or inspect the state of the single instance separately, and therefore we have to extend the signature of these functions with a new parameter indicating the entry index in the *vectors* that must be updated. However, with this approach we have the problem that we are missing the *context*. It means when calling the update or get_status function in the *NestKernel*, we do not know which entries to update because the Node class represents now a collection of nodes. To solve that issue, we need only to have a Wrapper class that has a *pointer* to the Node class, and as a class member of the Wrapper, we add an attribute representing the *local_id* of the node as the position of the entry in the Node class. But in order to make the Wrapper class able to communicate with the other *Nestkernel* components, the Wrapper must inherit from the Node class.

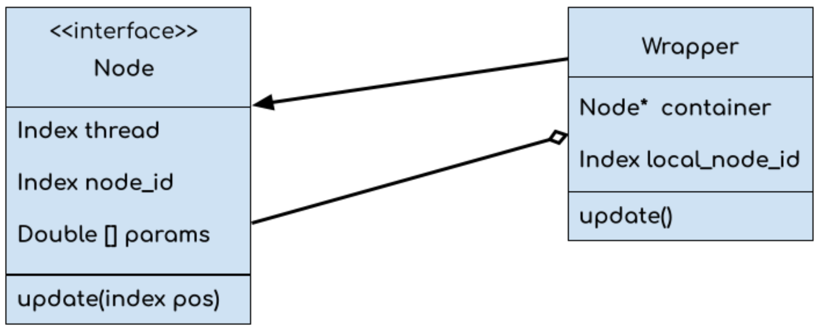

Figure 4.4: The naive solution for Vectorization: The Node class in the *NestKenel* is adjusted by extending all its functions by a new parameter indicating the local_id of the node that we want to apply the function on. Additionally, the attributes in the Node class are modified to have a vector from. The Wrapper class inherits from the Node class and at the same time has an attribute with a Node type. This makes the Wrapper class have an instance of itself, and we should definitely avoid using this design.

The **??** shows the relation between the Node and Wrapper classes. Here we have a cyclic dependency between the Wrapper and the Node class, as the Wrapper inherits from the node and at the same time it stores an instance of the Node class, which forces us to change the signature of the functions in the Node class and makes the Wrapper have a twice the number of function as in the Node class. Those functions are with the extended local_id parameter, and those without it. Definitely, this solution suffers from design problems, and it could make it hard for developer to maintain the logic and ensure a sound and a correct implementation.

4.3 The Container

To solve the issue in the naive solution, we have to separate between *vectorized* models and the already existing models. The context of which nodes must be retrieved and updated should be executed in blocks and in parallel without dependency between the nodes.

Additionally, we have to avoid changing the declaration of the `Node` class functions to avoid the overhead of propagating these changes to the other derived classes from the `Node` class and make the implementation of the *vectorized* models transparent to the *NestKernel* components.

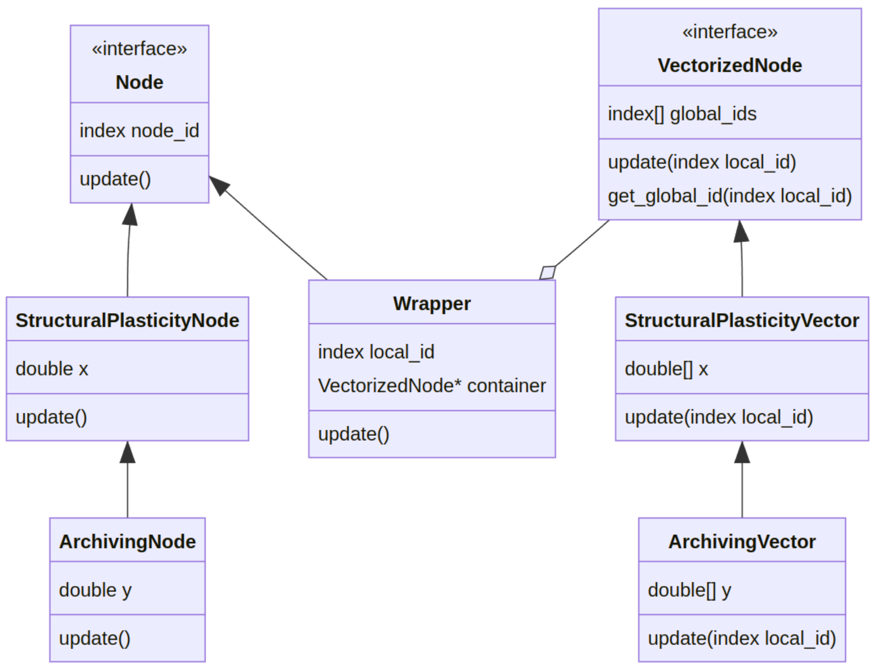

Figure 4.5: The Wrapper and the VectorizedNode: The new design after considering the drawbacks of the naive solution. The `Wrapper` still inherits from the `Node` interface, but instead of owning an attribute of the `Node` class, it owns an attribute of the `VectorizedNode`. The `VectorizedNode` class is the twin of the `Node` class, with the only difference that supports vectorization.

In the **??**, we illustrate the derived solution from the naive solution. The idea behind this solution is to split the `Wrapper` from the naive solution into two different classes. We have a new `Wrapper` class that inherits from the `Node` class, and it has a member variable of a `VectorizedNode` type. The `VectorizedNode` is an interface that has the same exact functionality as the `Node` interface, with the only difference that all its member variables are vectors and all functions have an extra parameter which represents the position of the entry that we want to retrieve or update. Instead of having n instances of the `Node` class from any arbitrary model, we have a single container instance from the `VectorizedNode` that has vectors of size n and each of these entries are represented by the *Wrapper*. Since the `Wrapper` inherits from the `Node` class, the other component in the *NestKernel* can then interact with the `Wrapper` without any further adjustments. Each call to the functions inside the `Wrapper` are forwarded to the `VectorizedNode` that exactly call the corresponding function with the `local_id` as the context coming from the `Wrapper` instance. It is also important to notice that for each of the `StructuralPlasticitiyNode` and the `ArchivingNode` we have the equivalent vectorized class derived from the `VectorizedNode` that has the same functions but with the support for vectorized models.

As depicted in the **??**, registering new external models in the *NestKernel* starts with calling the `register_node_model_vectorized` function in the `ModelManager`. The

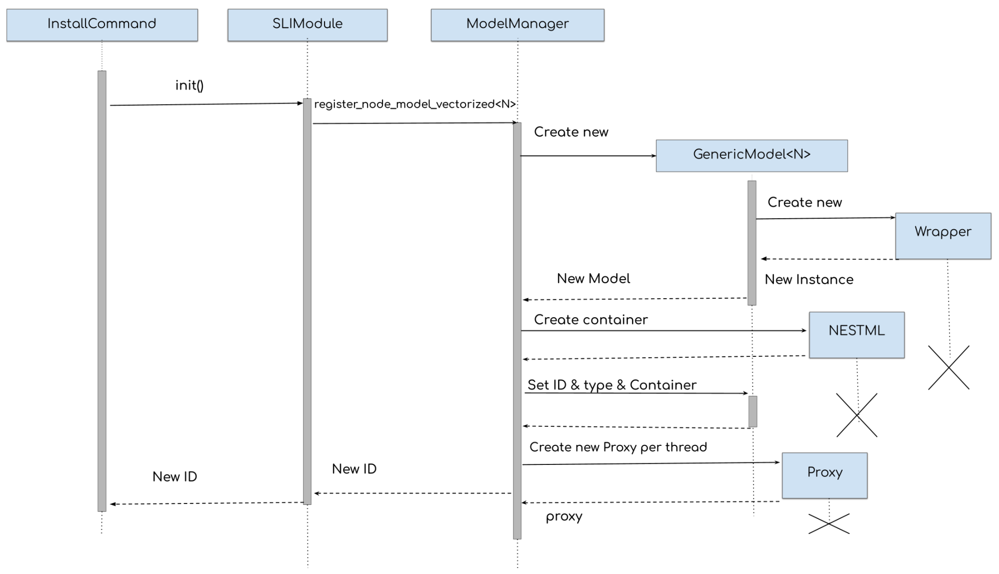

Figure 4.6: Registering a Vectorized model: The message sequence chart depicts the interaction between the components in the *NestKernel* responsible for registering the vectorized new custom models.

function creates then a new instance of the `GenericModel`, but instead of providing the model class type as the value of the template `T`, we set `T` to the `Wrapper`. In other words, we create an instance from the `GenericModel<Wrapper>`. An instance of the `Wrapper` is created and stored locally in the `GernericModel`. Next, once the constructor of the `GernericModel` finishes, the control returns to the `register_node_model_vectorized`, we proceed by creating an instance of the `VectorizedNode` which is indicated as `NESTML` in the figure. We create the container from the `VectorizedNode` and then update the created model by setting its `id`, `type` and `container`. Afterwards, as in the normal registration workflow, we just create the `Proxy` models for each thread which is not affected by the vectorization. In short, the main difference between registering a normal model and vectorized model is the in the provided class type `T` in `GenericModel` with the extra step of creating the container as an instance of the `VectorizedNode` class, otherwise everything stays the same.

In the typical way of running a simulation with external simple models without the *vectorization* feature, there is am important condition that must be satisfied between the created nodes and the current number of used threads. The condition states that each node exactly belongs to one thread and threads can not modify the state of nodes coming from different threads. Since the `VectorizedNode` behaves the same as the `Node` class, then the same must hold for containers. Each container must belong exactly to one thread, and threads can only modify the containers they possess. In order for the `VectorizedNode` to fulfill this requirement, we have slightly modified the `Model` Class and the `ModelManager` in the `add_neurons_` function.

The **??** shows the inserted new functionality in the `add_neurons_` function for creating new `Node` instances from the registered models. We first start by checking if the model class supports *vectorization*, and if it is the case, then we ask the model instance if it had seen the current running thread t or not. If the model knows the thread t, then an instance of the `VectorizedNode` was already created. In that case, we simply ask

```
if ( model.get_uses_vectors() )
    {
      if ( model.has_thread_assigned( t ) )
      {
        index t_container_pos = model.get_node_pos_in_thread( t );
        t_container = vectorized_nodes.at( t ).at( t_container_pos );
      }
      else
      {
        t_container = model.get_container()->clone();
        t_container->set_thread( t );
        vectorized_nodes.at( t ).push_back( t_container );
        index t_container_pos = vectorized_nodes.at( t ).size() - 1;
        model.add_thread_node_pair( t, t_container_pos );
      }
    }
```

Listing 4.3: Assigning containers to threads

the model to give us the position of the container in the thread t and we retrieve it in order to resize it and adjust it accordingly. If the model has not seen the thread yet, then we simply *clone* the container that is stored inside the model. Once the container is cloned, we update the container by providing the thread t that is owning it and also insert the container in the thread context. To simplify the matching between containers and threads, we simply have an attribute called `vectorized_nodes` in the shape of `vector<vector<VectorizedNode>>`, so that the first dimension is occupied with the `thread_id` and the second dimension with the created containers. The created container c within the thread t is then stored as `vectorized_nodes.at(t).push(c)`, and the thread is registered in the container by simply calling `container.set_thread(t)`. Once the matching between the container and the thread is completed, we then register the thread t in the model instance, so that by the next call, the model knows which container to retrieve and at which position. For models that do not support *vectorization*, the whole new inserted block is ignored and the function `add_neurons_` continues to execute without further changes. Since the `Wrapper` inherits from the `Node` class, the `create` function inside the model always returns an instance from the provided class `T` inside the `GenericModel<T>` class, which can be either the registered model without *vectorization* or the new `Wrapper` class. As depicted in **??**, we almost change nothing in the `while-loop`, and instead of updating the container step by step, we simply wait until the while loop finishes and then update the container at once to save us the overhead of repeatedly calling the `resize` function.

4.4 Limitation

So far, we only made *Vectorization* work with *multithreading*, and we are still required to completely integrate it into the *NestKernel*. We have implemented the `VectorizedNode` class that represents a container holding *vectors*, and the size of these vectors is the number of created `Wrapper` instances pointing to the container. For simplicity, we can consider the container as an *array* and the `Wrapper` instances represent each entry in the array. Thus, each `Wrapper` can only modify the entry is pointing to. Right now, when we are updating the `Node` instances, we are iterating over each one of them individually. For

```
while ( node_id <= max_node_id )
      {
        Node* node = model.create( t );

        node->set_container( t_container );
        node->set_node_id_( node_id );
        node->set_nc_( nc_ptr );
        node->set_model_id( model.get_model_id() );
        node->set_thread( t );
        node->set_vp( vp );
        node->set_initialized();

        local_nodes_[ t ].add_local_node( *node );

        node_id += num_vps;
        has_at_leat_one_node = true;
      }

      local_nodes_[ t ].set_max_node_id( max_node_id );
      if ( t_container and has_at_leat_one_node )
      {
        t_container->resize( max_new_per_thread, t );
      }
    }
```

Listing 4.4: Creating nodes

the normal models without vectorization, we can not do anything about it, but for the vectorized models, we should aggregate the `Node` (i.e., `Wrapper`) instances that share the same container and then only apply the function we want on the container. Assume we have n containers and each container has k `Wrappers` pointing to it, and we have a function f that is applied to these items separately. In total, the function will be called $n \cdot k$ times, and for a large k that might be a huge overhead. Instead, we can aggregate the items and only call the function f n times. Thus, each container can fully utilize the *vectorization* by applying internally the function f in parallel to the k items.

Also, the `get_status` and `set_status` functions do not fully utilize the implemented new data structure for *Vectorization*. Assume the container has n vectors of size k (k is the number of attributes in the model). The normal execution of these functions takes each `Node` instance individually and iterates over the k attributes and returns a *dictionary* containing the values of the k attributes. This workflow does not work well with *vectorization*, because if each `Wrapper` within the same container executes the same logic, then we might have a high number of *cache misses*. Assume we have two consecutive access x_i and x_{i+1}, and we have k large enough so that not all vectors can be loaded into the registers. The `Wrapper` at position x_i loads its necessary data and returns the *dictionary*, the second `Wrapper` x_{i+1} on the other hand must load the first k vectors again. What really should happen is that each of the l vectors must be loaded once in the registers. Since we have a consecutive access, all the entries must be loaded and ready to be used for constructing the result. Better yet, we simply again group the `Wrapper` instances by the container type and let the container call these functions and treat each of these k vectors individually and in parallel.

Chapter 5

Performance

The major goal of this work is to have a fully automated model generation in *PyNEST* without drastically degrading the performance of the simulation script and encouraging users to use the new implemented *JIT* module. In order to test the impact of the using *JIT* in the simulation script, we need to test different configurations that might eventually appear when running the simulation with *JIT*. Firstly, we test the neuronal network with the script explicitly calling *PyNESTML* module and the `nest.Install` function and consider its running time as our base model. Secondly, we run the same neuronal network, but this time everything is controlled by the *JIT* module. Thirdly, we repeat the same experiment, but this time we choose a neuronal network that includes using a custom implemented synapse.

As the *NestKernel* does not completely utilize the new data structure behind *vectorization*, we can not really make meaningful evaluation of its impact on the performance. Therefore, we only decided to compare single functions between simple and *vectorized* models. Such functions are `nest.Create` and `nest.GetStatus`.

From the *JIT* perspective, we do not really care about the size of the network or its behave, because we only simply retrieve the model and make it available to the simulation script. In that case, any neuronal network should be fine to test with. In the first experiment, we choose a random balanced network as described by *Brunel* (**?**). For the second experiment with synapse, we build a network with a spike-timing dependent plasticity (STDP) synaptic type.

For *vectorization*, we only test the `nest.Create` and `nest.GetStatus` functions by creating instances of the *Izhikevich* (**?**) with modifying the number of instances and the number of used threads.

5.1 The JIT Module

Running The JIT Module Without An external Synapse Type

In the first experiment in testing the *JIT* module, we run the *Brunel* network by explicitly using the *PyNESTML* to generate the code for the required model, and then we run the same network by using the *JIT* module. The result of the experiment can be found in the **??**.

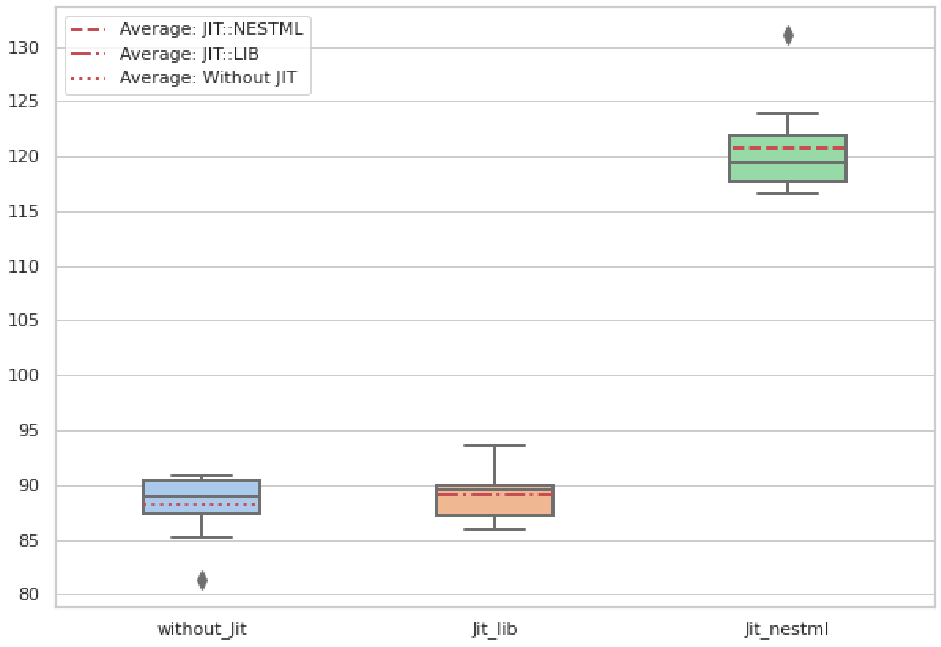

Figure 5.1: The max-average-min runtime in seconds

In the **??**, we show the runtime of three possible cases. The first is when we are explicitly calling the *PyNESTML* module, and it is labeled with *Without_Jit*. The second case is when running the simulation with an external model using the *JIT* module, and it is labeled with *Jit_Nestml*. The last case is when using *JIT* with an external library, and usually it is the result after generating the code for the *nestml* model. The last case is labeled with *Jit_Lib* in the plot. In each of the three cases, we show the *maximum*, *average* and *minimum* required to run the simulation.

The first case (i.e., *Without_Jit*) has the lowest runtime, and it took on average 90 seconds to finish the simulation. In the other hand, the average runtime for the *Jit_Nestml* took around 130 seconds, and the *Jit_Lib* took around 90 seconds. Both the first case and the last case have almost the same behavior, with the exception that the *minimum* runtime for the last case was a bit higher. This can be explained with the introduced overhead in the *JIT* module, and it is most significant in the second case. All the three values of the *Jit_Nestml* are the highest among the other cases, and it can be explained by the time required for searching the model and maintaining `NodeCollectionProxy` by converting the `JitNodeCollection` to the real `NodeCollection` instances and copying the parameters from one instance to the other.

Running The JIT Module with An external Synapse Type

In the second experiment, we ran the *STDP windows* simulation script from the `NESTML` tutorial, and again we compared the simulation runtime with using *JIT* against the original script without *JIT*. In this tutorial, we have a function that is responsible for creating the instances of the models and creating the network, and it is called multiple times in a *for-loop* (exactly 151 times). In the *JIT* settings, it means that the model will be searched many times, and we would expect that only the first iteration will take most of the time in comparison to the subsequent iterations.

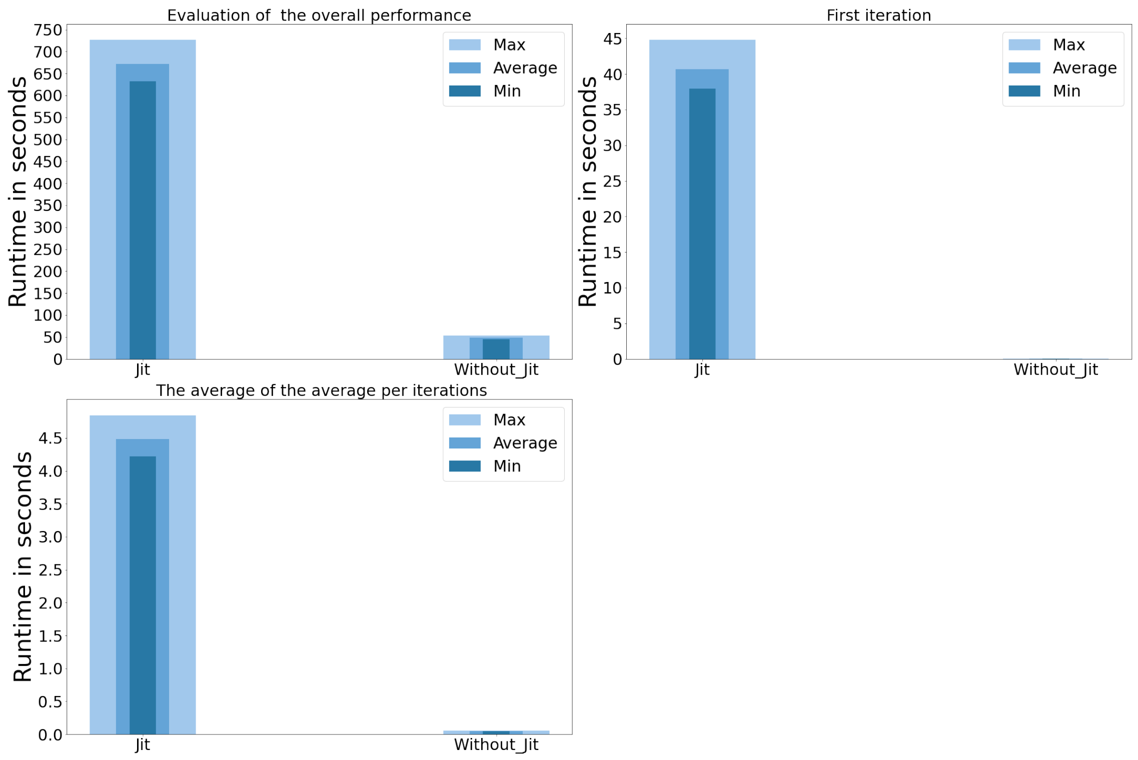

Figure 5.2: The overall runtime

In the **??**, we analyze the runtime of the whole script, and then we take a look at the runtime of the function responsible for creating the network in each iteration. At the top left in the figure, we can observe the significant effect of using the *JIT* module with an external synapse type. The *JIT* module requires on average more 700 seconds, whereas the normal execution took around 50 seconds to finish. The minimum runtime of the *JIT* is also very high in comparison to the normal mode. If we observe the second plot at the top, we can straightforward see the reason behind the 700-second runtime. The first iteration alone took more than 40 second to finish, whereas the normal execution took less than one second to finish. This difference is caused by the same overhead discussed above, plus now we have to trigger the code generation pipeline again and replace the *source/target* elements in the network with the new correct neuron model and update its state along the synapse instance.

In the third plot at the bottom left of the figure, we show the average time needed by all the 151 iterations in each run. As expected, the result shows that mostly only the first iteration takes the longest running time. The average is around 4.5, and it is way smaller than the average of running the first iteration. The results seem to be consistent as we have 4.5×151 is around 700 seconds. On the other hand, the average runtime of the 151 iterations without *JIT* is significantly lower. As the external models will be available after the first iteration, which is the main reason why the left bottom plot shows better runtime than the first iteration, we still have the overhead of going through the `Wrapper` to execute the *PyNEST* functions, which introduces a great overhead.

5.2 Vectoriztation

As *NestKernel* does not completely support *vectorization* yet, we restrict our tests to the direct functions involved with the vectorized the model. We tested both the `nest.Create` and `nest.GetStatus` functions. Both experiments share the same configuration, we mainly can either change the number of created nodes and keep the number of threads constant or vice-versa. The experiments are executed with a *Ryzen 7 3700u* processor with four cores and eight threads. For the number of the nodes, we have picked {1, 10, 30, 50,100,300,500,2000,5000,8000,1000} and for the threads we have {1, 2, 4, 8, 16, 32}. Each sub-experiment is run five times, with storing the *Wall* time in a *csv* file containing the runtime of the *vectorized* and *non-vectorized* model. In each sub-experiment, we have three plots. In the first plot on the left, we show the *max-average-min* runtime of the vectorized model. The middle plot shows the *max-average-min* runtime of the simple *non-vectorized* model. Finlay, the last plot on the most right, overlays the average runtime for both models in a single plot. The *y-axis* in the first two plots shows the values after applying the *logarithmic* function with base 10 on the runtime values, which were stored in microseconds, whereas the values in the last plot are only in microseconds.

5.2.0.1 The Create function

Varying the number of nodes with a constant number of threads

In the first configuration in testing the `nest.Create` function, we fix the number of threads and vary the number of nodes. In the **??** until **??**, we show the results of the experiments. For each number of threads, we display the performance of the `nest.Create` function with the vectorized model against the simple models.

In the **??**, we observe that both types of models behave the same until reaching a number of nodes larger than 300 ($10^{2.5}$). After that point, the vectorized model starts to become a bit slower on average, but with few microseconds. Another observation is that the *min-max* area of the vectorized model is a bit fat and gets more narrow after reaching 10^3, whereas the *min-max* area is almost always narrow, and it gets more narrow after reaching 300 nodes.

In the **??**, we observe the same performance for both types of models, even the *max-min* area looks almost the same. As in the previous experience, the performance of the vectorized model becomes a bit slower after reaching 300 nodes with a few milliseconds in comparison to the simple model. In **??** with four threads, we have the same observation as in the **??** with two threads.

In the rest of the figures with threads in {8, 16, 32}, we observe that the vectorized model gets slower and the gap between both models gets wider but still in the range of a few milliseconds.

Varying the number of threads with a constant number of nodes

In general, the plots between the **??** and **??** show that the `nest.Create` function has a consistent behavior. In all the plots, we can observe the discontinuity that happens around the points 4, 8 and 16. At these points the slope changes significantly, and the

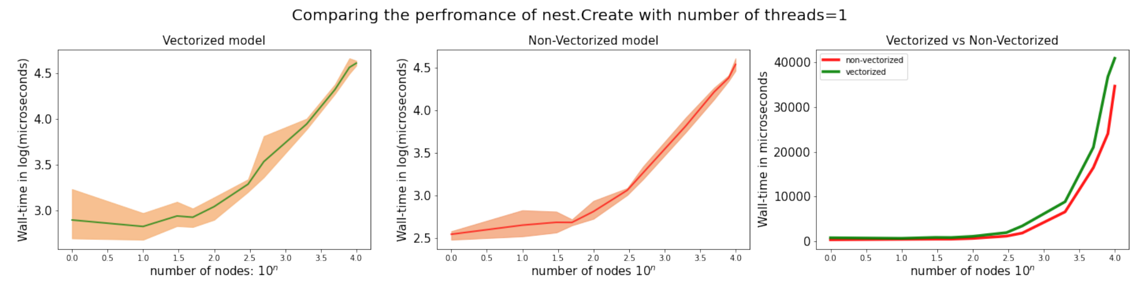

Figure 5.3: Performance of `nest.Create` function with the number of *threads* = 1

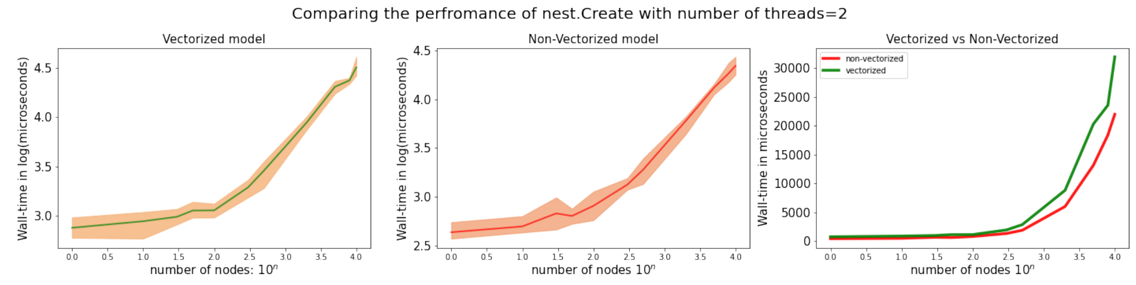

Figure 5.4: Performance of `nest.Create` function with the number of *threads* = 2

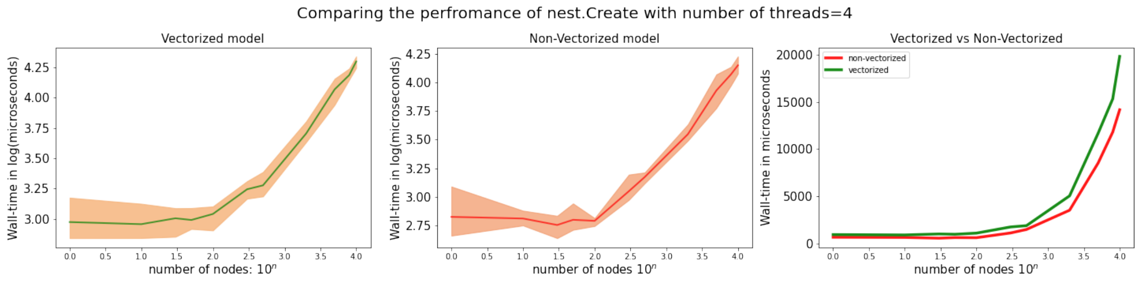

Figure 5.5: Performance of `nest.Create` function with the number of *threads* = 4

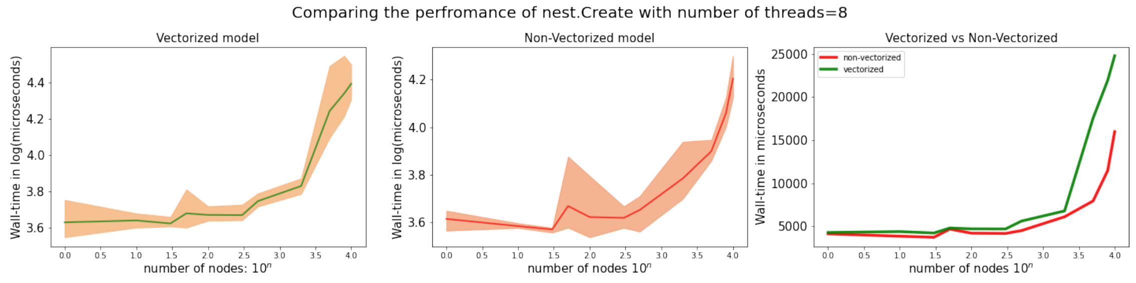

Figure 5.6: Performance of `nest.Create` function with the number of *threads* = 8

reason behind it is the processor specification. Recall that we ran the experiments on a four-core processor. In general, the vectorized model is a bit slower than the simple model and the largest difference can be observed in the **??** with 5000 nodes. Compared to the previous experiment with fixed number of threads, the *min-max* area seems to be more a bit fat, which might implicate that the current average results might be a bit noisy.

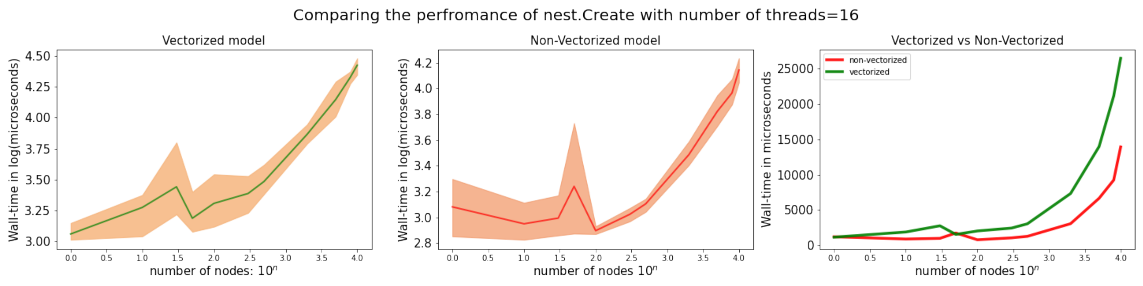

Figure 5.7: Performance of `nest.Create` function with the number of $threads = 16$

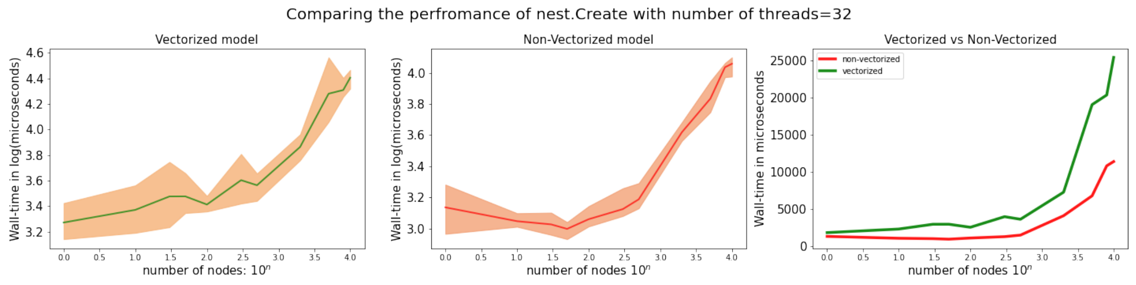

Figure 5.8: Performance of `nest.Create` function with the number of $threads = 32$

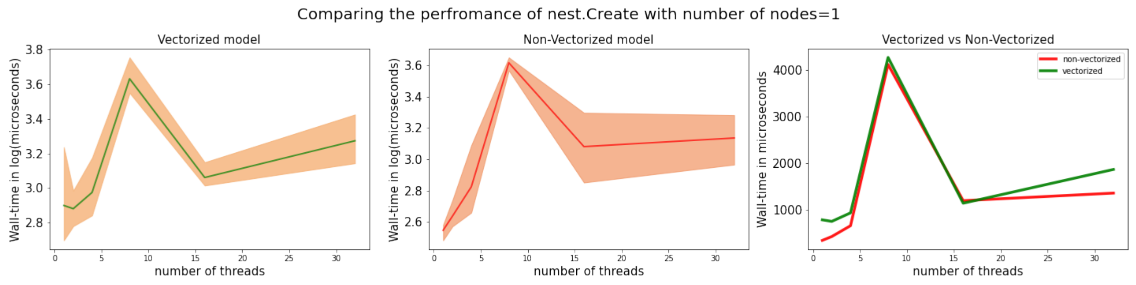

Figure 5.9: Performance of `nest.Create` function with the number of $nodes = 1$

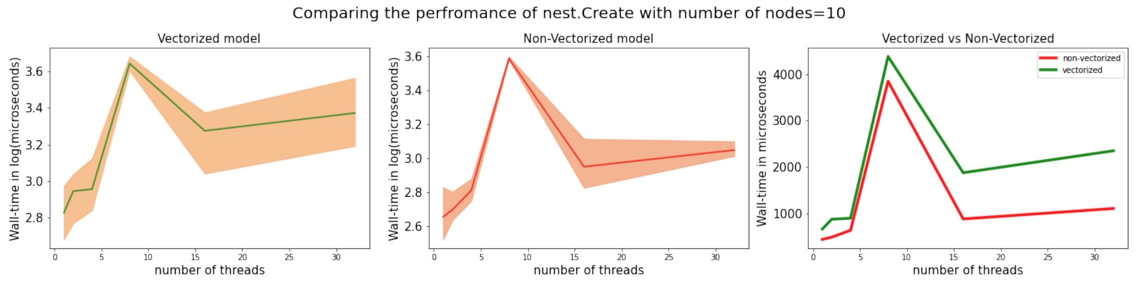

Figure 5.10: Performance of `nest.Create` function with the number of $nodes = 10$

5.2.0.2 The GetStatus function

Varying the number of nodes with a constant number of threads

Even with the `nest.GetStatus` not fully being optimized to support vectorization, both the vectorized model and the simple one has the same performance, with the vectorized model being slightly better with few microseconds.

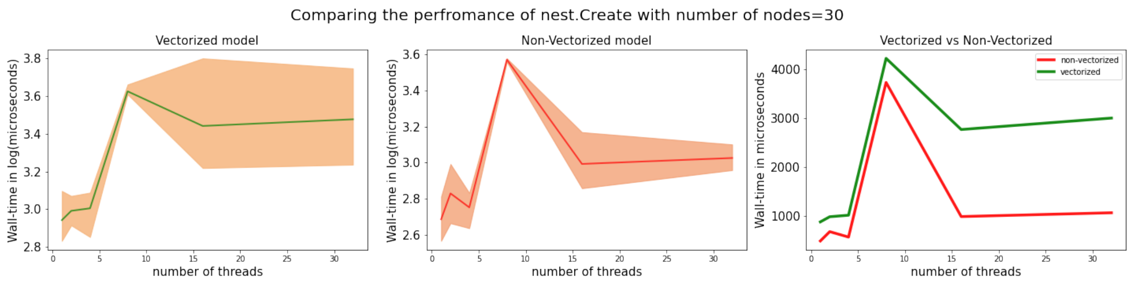

Figure 5.11: Performance of `nest.Create` function with the number of *nodes* =30

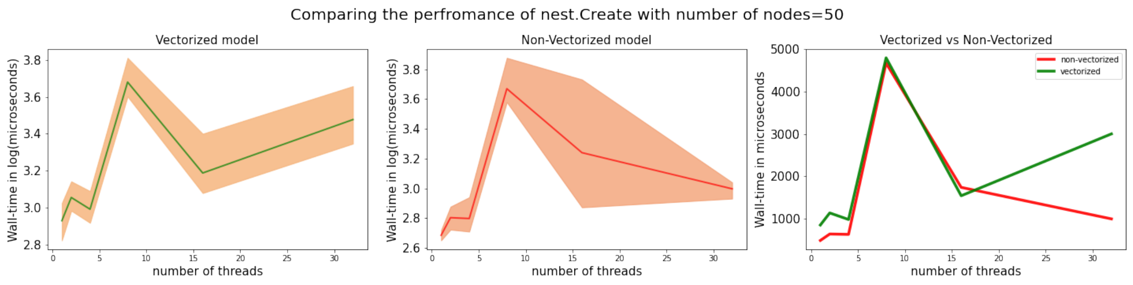

Figure 5.12: Performance of `nest.Create` function with the number of *nodes* =50

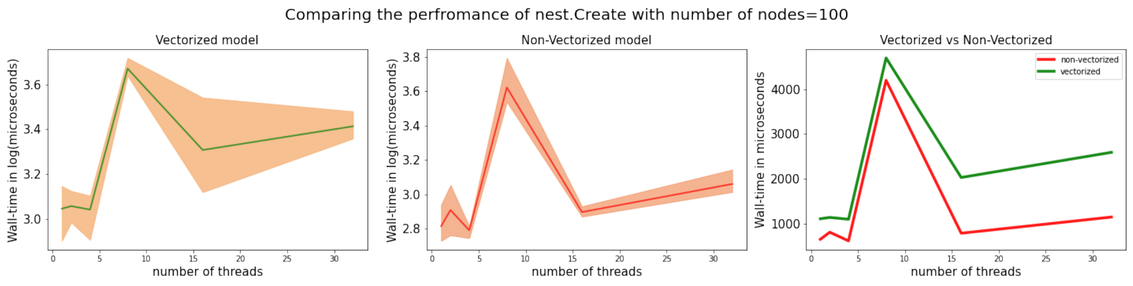

Figure 5.13: Performance of `nest.Create` function with the number of *nodes* =100

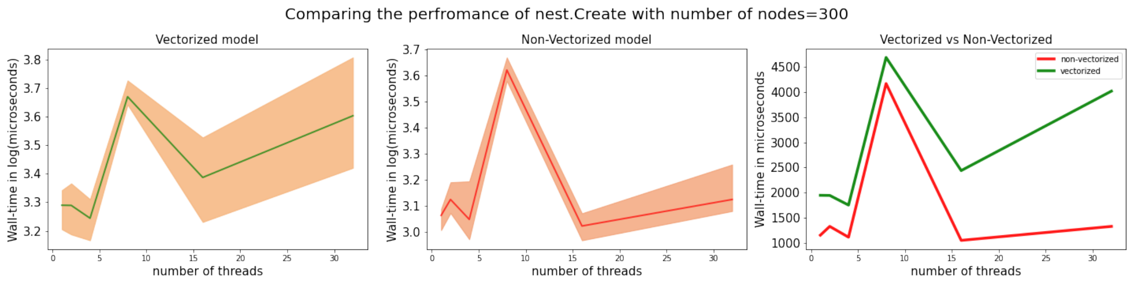

Figure 5.14: Performance of `nest.Create` function with the number of *nodes* =300

Varying the number of threads with a constant number of nodes

The `nest.GetStatus` with varying the number of threads has a very different behavior. In the **??**, both models have the same direction of variation with the only difference that at *threads* = 16 we observe that the vectorized model is becoming a bit faster. The *min-max* area in both models is also almost the same. With the number of nodes equal to ten in **??**, we observe that between the points 6 and 16 the vectorized model is significantly faster,

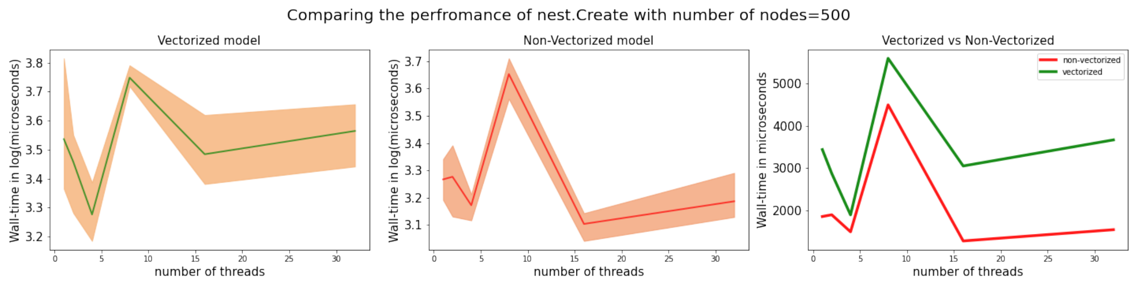

Figure 5.15: Performance of `nest.Create` function with the number of *nodes* =500

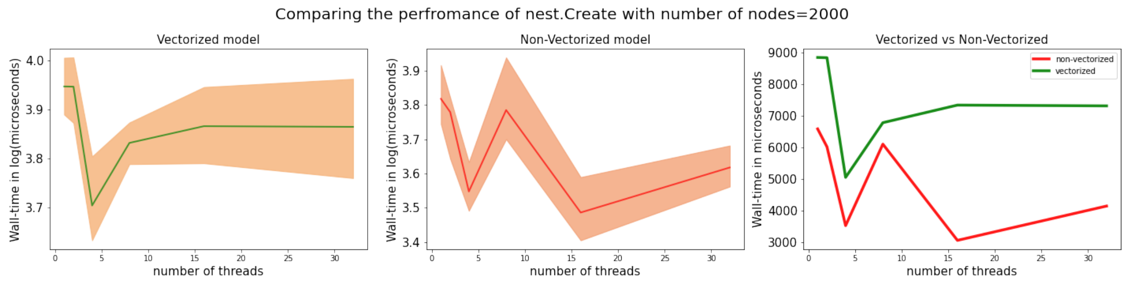

Figure 5.16: Performance of `nest.Create` function with the number of *nodes* =2000

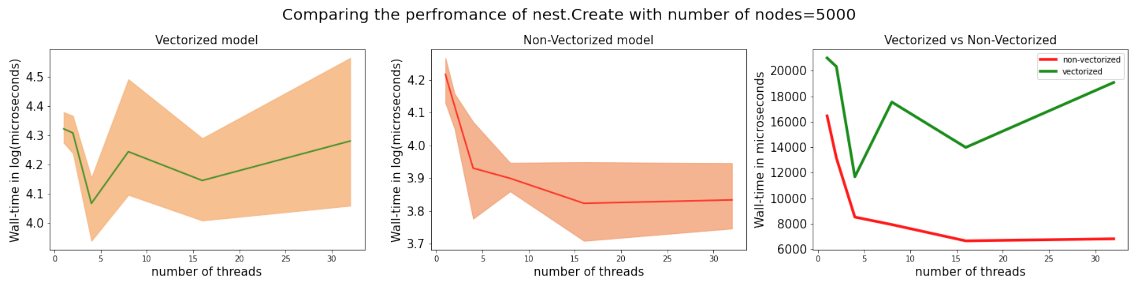

Figure 5.17: Performance of `nest.Create` function with the number of *nodes* =5000

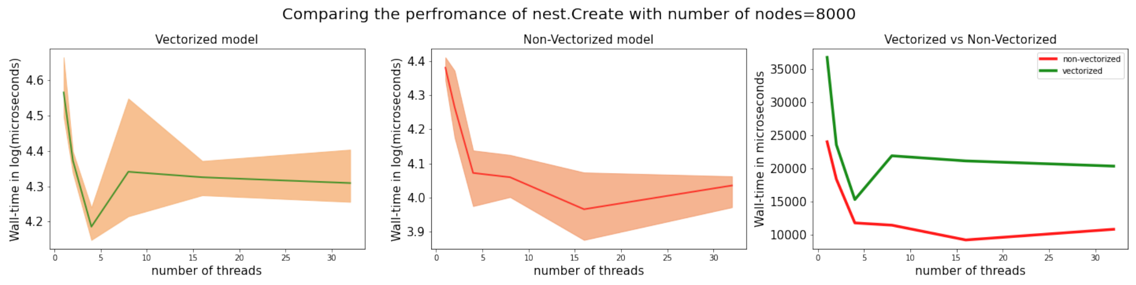

Figure 5.18: Performance of `nest.Create` function with the number of *nodes* =8000

but then starting from 16 the vectorized model becomes slower again. In both plots, the *min-max* area is very noisy.

In **??**, both models have the same behavior, with the vectorized model being a bit faster between the points 3 and 14 and then again after 20. Both *min-max* areas have almost the same shape. In **??**, the average performance between the models is significantly different with 1500 microseconds on average. Again, we directly observe the points where the slope of the curves changes. Those points are at position 4, 8 and 16. **??** with 100 nodes shows

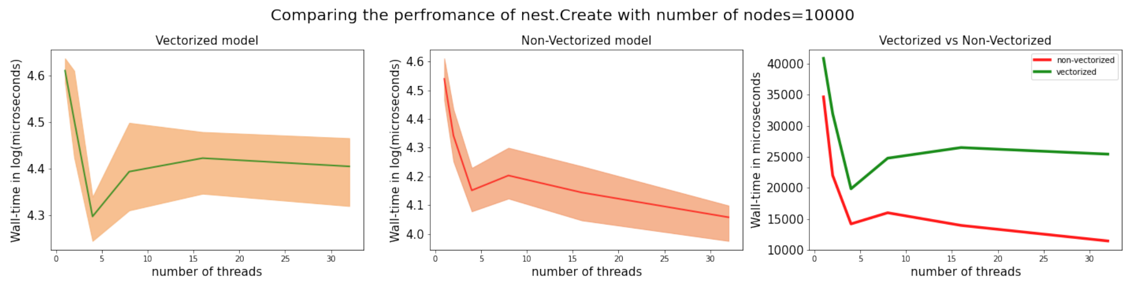

Figure 5.19: Performance of `nest.Create` function with the number of *nodes* =10000

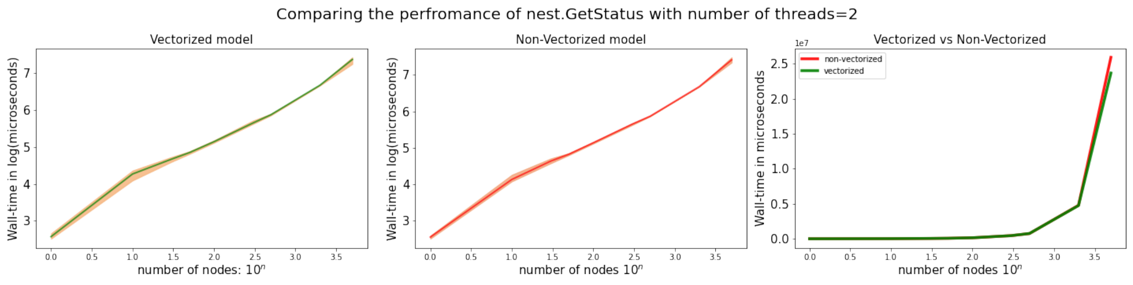

Figure 5.20: Performance of `nest.GetStatus` function with the number of *nodes* =2

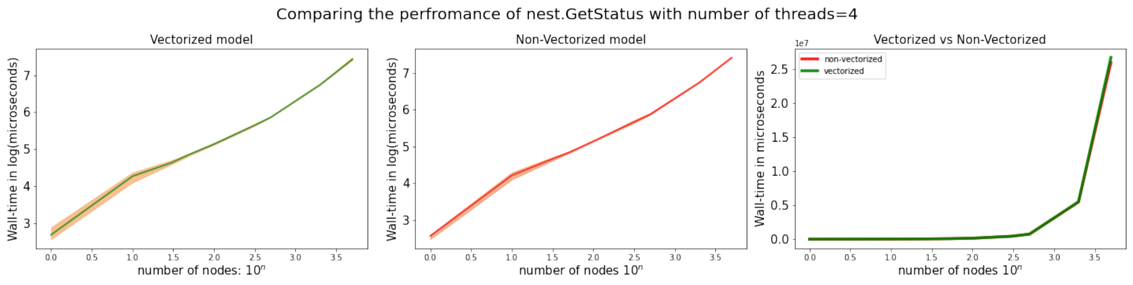

Figure 5.21: Performance of `nest.GetStatus` function with the number of *nodes* =4

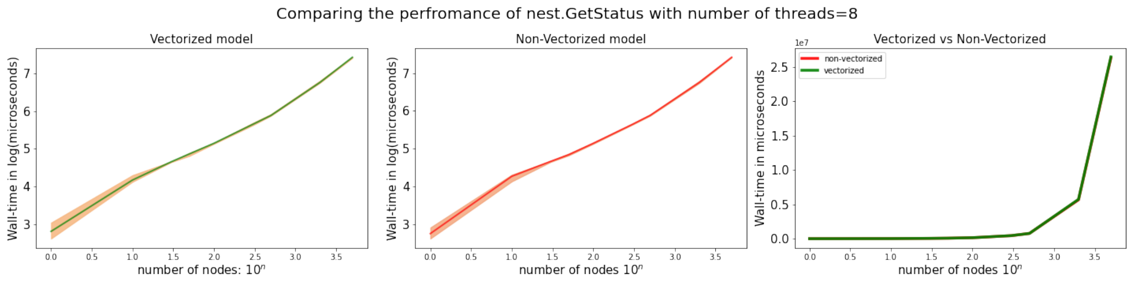

Figure 5.22: Performance of `nest.GetStatus` function with the number of *nodes* =8

also the same observations as in the **??**.

In **??** with 300 nodes, the vectorized model takes almost completely the lead in all thread values, with the only exception between position 6 and 10. Both **??** and **??** respectively with 500 and 2000 nodes show the same observations, where the vectorized model is being faster than the simple model.

Reaching higher number of nodes as in **??** with 5000 nodes in **??** with 8000 and in **??**

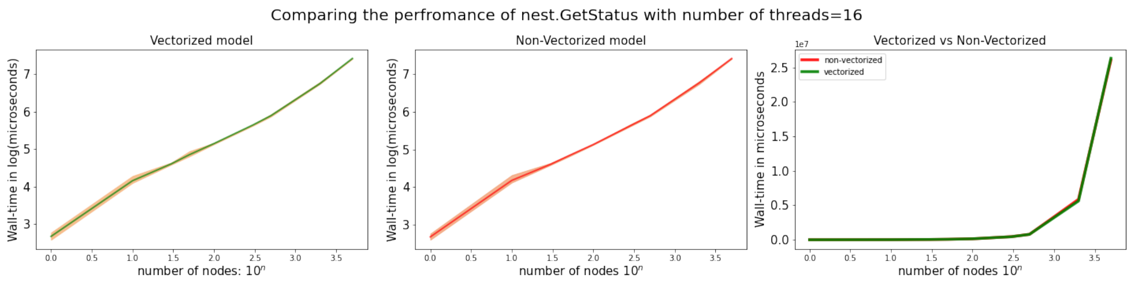

Figure 5.23: Performance of `nest.GetStatus` function with the number of *nodes* =16

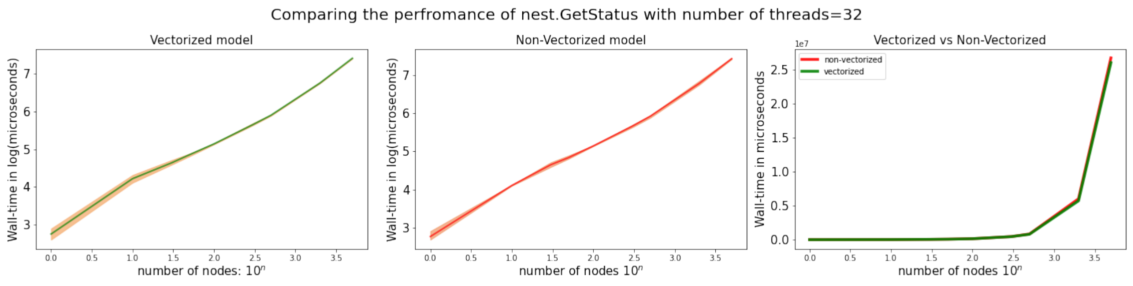

Figure 5.24: Performance of `nest.GetStatus` function with the number of *nodes* =32

with 10000 nodes, the vectorized model is either behaving the same as the simple model or performing even better.

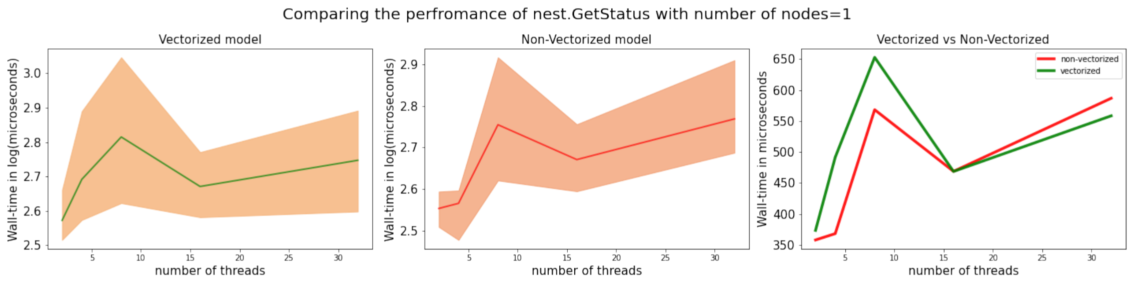

Figure 5.25: Performance of `nest.GetStatus` function with the number of *threads* =1

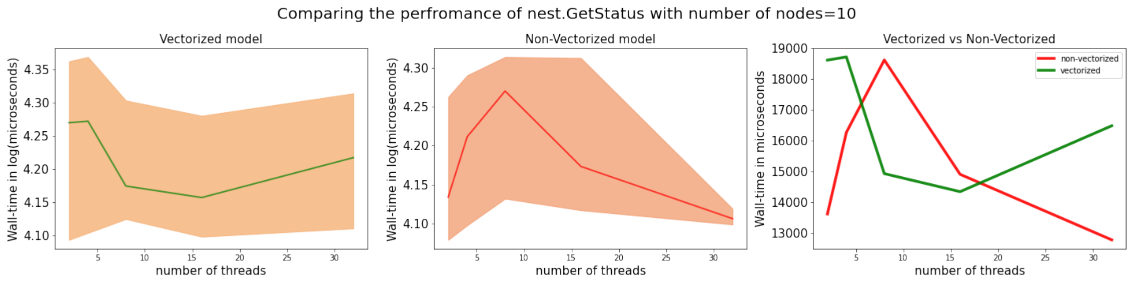

Figure 5.26: Performance of `nest.GetStatus` function with the number of *threads* =10

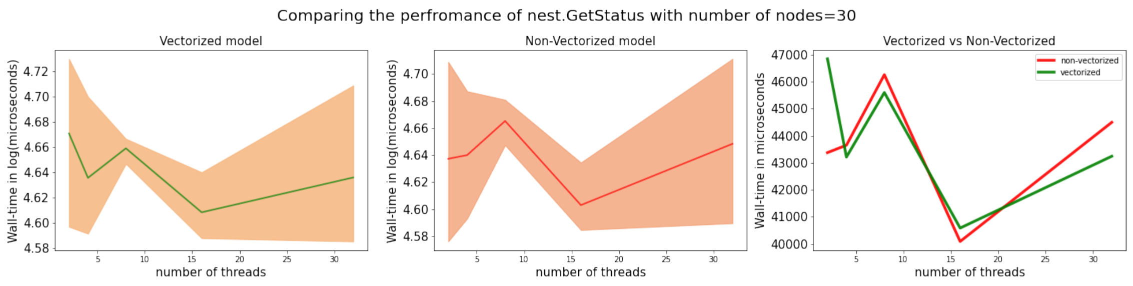

Figure 5.27: Performance of `nest.GetStatus` function with the number of *threads* =30

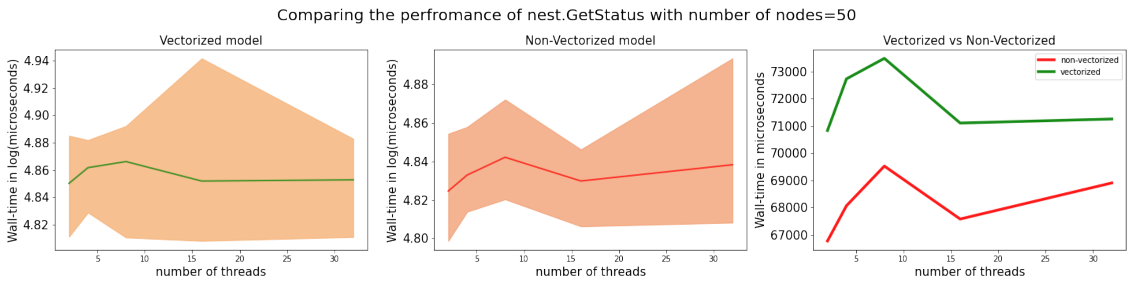

Figure 5.28: Performance of `nest.GetStatus` function with the number of *threads* =50

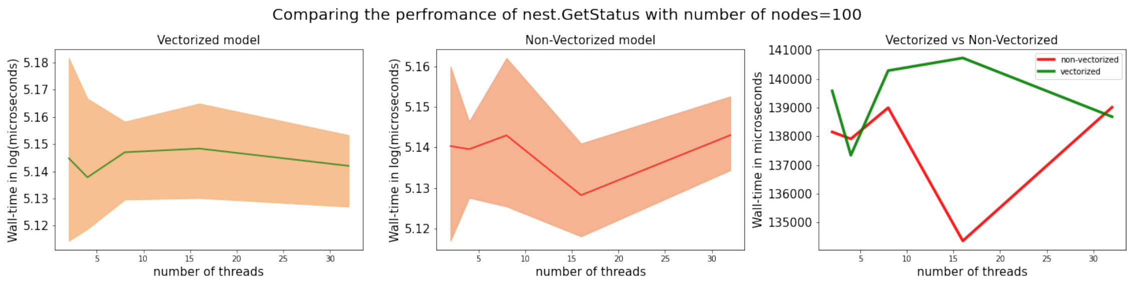

Figure 5.29: Performance of `nest.GetStatus` function with the number of *threads* =100

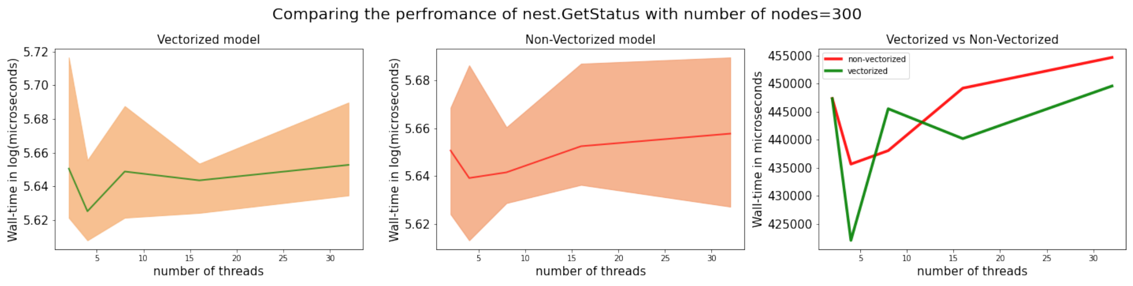

Figure 5.30: Performance of `nest.GetStatus` function with the number of *threads* =300

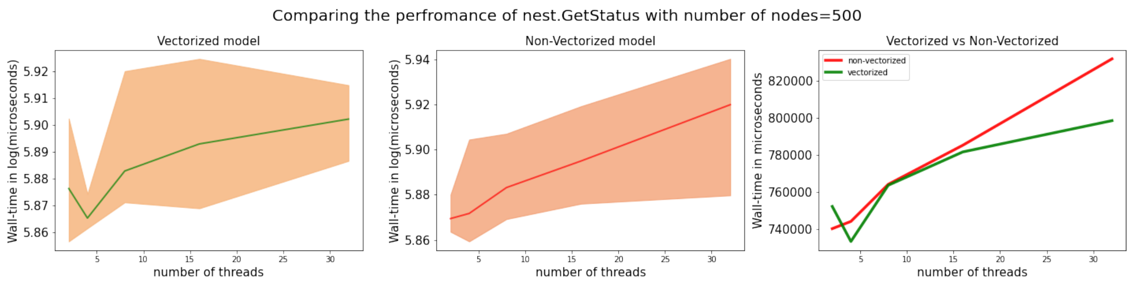

Figure 5.31: Performance of `nest.GetStatus` function with the number of *threads* =500

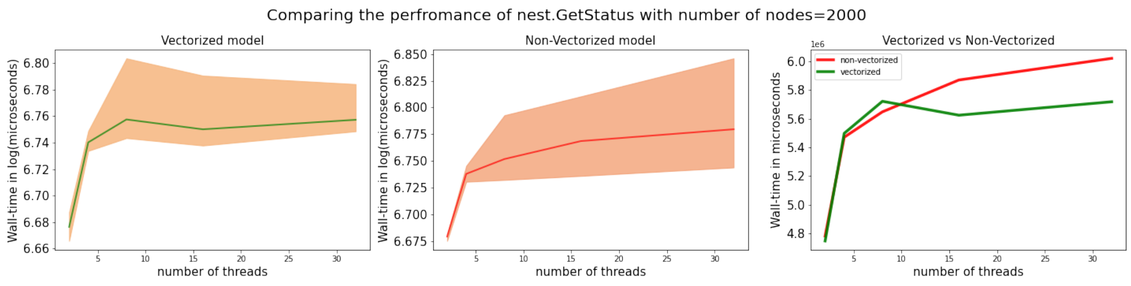

Figure 5.32: Performance of `nest.GetStatus` function with the number of *threads* =2000

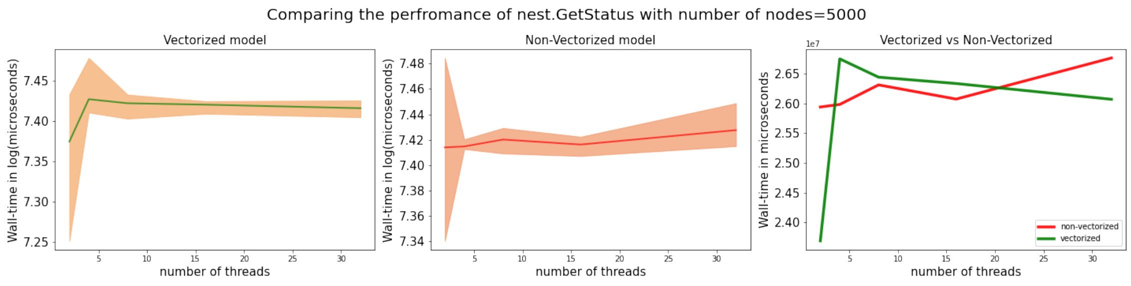

Figure 5.33: Performance of `nest.GetStatus` function with the number of *threads* =5000

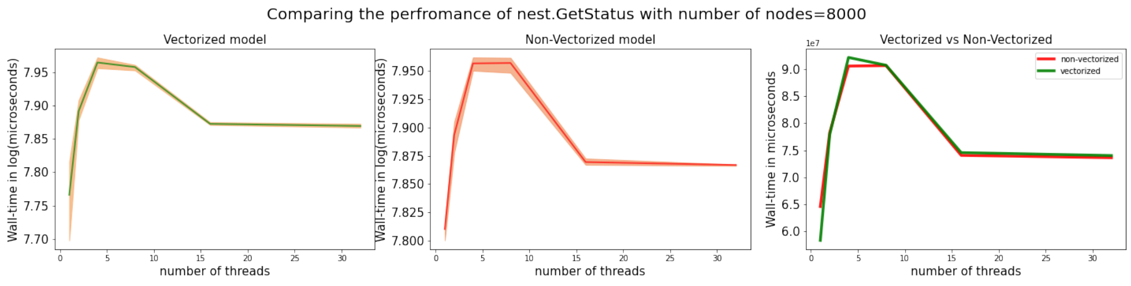

Figure 5.34: Performance of `nest.GetStatus` function with the number of *threads* =8000

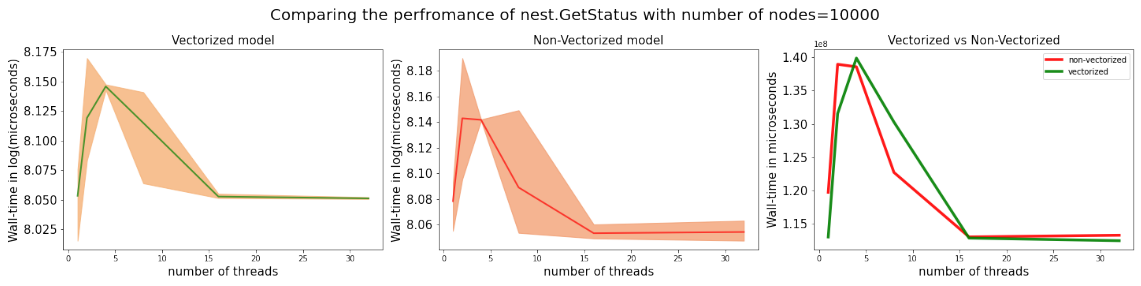

Figure 5.35: Performance of `nest.GetStatus` function with the number of *threads* =10000

Chapter 6

Discussion

6.1 Conclusion and Summary

JIT Module: The Wrapper

We have implemented a Wrapper around the *PyNEST* Python module that can intercept its function calls, and depending on specific constraints, we can either apply some pre-processing and post-processing steps before calling the function or ignore it completely. The Wrapper has a straightforward structure that renders its use simple and flexible. Each targeted function in *PyNEST* has a custom implemented wrapper, and these wrappers are registered in a *dictionary,* with the keys being the name of the targeted function and the values are the registered *Wrapper* classes. Once the *JIT* module is imported, it iterates over the *PyNEST*'s functions, methods, and classes, and if one of these objects is registered in the *dictionary*, then a wrapper instance is created and inserted in the *JIT* module. Non-targeted objects are inserted directly without having a wrapper in between. Another main feature of the wrappers is that they are implemented independently and do not influence each other.

JIT Module: The Lightweight Version

Since the implemented wrapper for the `nest.Create` function must search for the model, and in the worst case, the code of the models must be generated, we decided to split the code generation overhead. On the call to the `nest.Create`, the user will not need to wait for the whole code of the model to be generated, built, and installed, but instead we create a *lightweight* version of the model that has only the *State* and *Parameter* blocks, which are only the necessary attributes needed from the model after creating the instances, and then the real code generation of the library happens in the background without blocking the execution of the wrapped `nest.Create` function. Between the `nest.Create` and `nest.Connect`, the user can modify the attributes of the lightweight version by assigning new values, being either deterministic or random.

One drawback with this approach is that after calling `nest.Connect` or `nest.Simulate` the models will be instantiated for real, and then we need to copy the values of the attributes from the lightweight version to the real instances. Thus, we have the same amount of memory allocated twice, one in the *Python* level and the other in the *C++* level.

JIT Module: The NodeCollectionProxy

In order to retrieve information from the *lightweigh* version and to make handling transparent and similar to the `nest.NodeCollection`, we have implemented the `JitNodeCollection` and the `NodeCollectionProxy`. The relation between these three elements may be seen as a *tree-like* structure. The `NodeCollectionProxy` is the root element in the *tree*. It has two children, the first one is the `JitNodeCollection` and the second one is the `NodeCollection`. The `JitNodeCollection` in the other hands has an arbitrary number of children. If the *ids* of the children of the `JitNodeCollection` build a contiguous space, and they belong to the same model, then they are represented by the `JitNode`, which represents in the context of the *tree* the *leaves*. The `NodeCollectionProxy`, the `JitNodeCollection` and the `JitNode` share the same *interface* that implements the support of retrieving and setting the attributes of the instances and also the support for *indexing* and *slicing* exactly like the provided functionalities in the `nest.NodeCollection`.

Vectorization

We have introduced a new data structure in the *Nestkernel* for handling models. The new data structure is based on the *structure of arrays SoA*, where instead of having n an instance of the same model with k attributes, we have exactly one instance that has k attributes in the form of *vectors* and each of these vectors has a size of n. The *Vectorization* of the models behaves like a container of nodes, so instead of updating the nodes separately, we can update just the exciting containers which eventually updates the states of the nodes in parallel by utilizing the efficiency of *SIMD* instructions in modern processors. The *Vectorization* should also decrease the amount of *cache misses*, as updating one container and due to the spatial locality of the items in the array, we can update large number of nodes without the need of requesting the data from the main memory multiple times.

Performance

The performance of the *JIT* module without using an external synapse model does not deviate a lot from running the simulation script without *JIT*. However, when using an external custom synapse, we observe a significant difference in the performance. The difference can be explained by the overhead of running twice some steps of the code generation pipeline, maintaining the functionality of the `NodeCollectionProxy` and finally replacing the network nodes and updating their parameters.

Even though the *Nestkernel* does not fully support *Vectorization*, we observe that the performance of the *vectorized* model and the *non-vectorized* model are quite similar on average.

6.2 Outlook

Modularity

We have already raised the issue regarding the duplicate memory usage between the *Python* and *C++* side because of the *lightweight* version. Another problem is in the `NodeCollectionProxy`, each change in the `nest.NodeCollection` must also be adjusted in the `NodeCollectionProxy`. Thus, maintaining the *JIT* project might require a bit of overhead. To mitigate these problems, we can improve the concept of the *lightweight* version and make it more flexible. The solution is to adjust the code generation pipeline and make it generate and build the library in pieces. The model may be split into two pieces. The first part contains the *State* and *Parameters* blocks. These blocks will be generated and built on the call the `nest.Create` function. The rest of the code can be then generated in the background and assembled with the first part when calling the `nest.Connect` or `nest.Simulate` function. Since the co-code generation of the neuron and synapse just moves variables from the synapse to the neuron, we may then generate the code of the synapse and split it into two parts. The first part includes building the moved variables in a separate library and then append them to the neuron *State* block during the simulation. The second part takes care of the rest of the functionality in the synapse and making it available during the `nest.Connect` call.

Removal of the NodeCollectionProxy

With the *modularity* in play, we can completely drop the usage of the current *JIT* data structures (i.e., the `NodeCollectionProxy, JitNodeCollection` and `JitNode`). The *Nestkernel* infrastructure will be adjusted to support the modularity of the models and make it possible to assemble the pieces of the model at the runtime. Therefore, more sophisticated architecture of the `NodeCollection` must be designed to support modularity and make querying the model and updating their *State* and *Parameter* blocks possible even if the model is not fully built.

Vectorization: Updating the containers not the single nodes

The `update` function and other important functionalities related to the simulation do not efficiently utilize the *vectorized* models. Each of these functions handles the instances of the different models separately and does not use the created containers that represent all instances of each model. The *NestKernel* must be redesigned to use the containers as the unit for the whole simulation. Instead of storing a list of instances of the `Node` class, we store instances of `VectorizedNode` as a list of containers. Each container executes then the same instruction in parallel for all the nodes it represents.

JIT The Orchestrator

After removing the `NodeCollectionProxy` with the introduced *modularity*, the *JIT* module will have the only task of assembling the pieces of the model. It makes sure that the neuron model is connected with the correct synapse type without the necessity to run the code generation pipeline again to match the synapse type with the correct neuron model.

Appendix A

Programmdokumentation