Automation of Massive Modelica Simulations using Multithreading and Multiprocessing in Python

Sergio A. Dorado-Rojas Luigi Vanfretti

Department of Electrical, Systems and Computer Engineering
Rensselaer Polytechnic Institute
Troy, NY, USA
{dorads, vanfrl}@rpi.edu

Abstract

In this paper we present a Python-based approach to execute Modelica simulations in Dymola based on concurrency (multithreading) and parallel computing (multiprocessing). This data-based parallelism approach is employed to reduce the total execution time to perform 20,000 simulation scenarios that consider different transmission line contingencies in a power system. We show quantitatively the value of multithreading and multiprocessing with respect to the traditional sequential approach with metrics of CPU utilization, processor temperature and VRAM usage for each of the proposed methods. Computation time is substantially reduced by the use of concurrent and parallel computing techniques thanks to the more efficient use of CPU resources in comparison to the traditional sequential workflow. Regarding energy consumption, we verified that the concurrency-based solution tends to employ more power than the parallel multiprocessing approach. Finally, we evaluate the performance of each proposed solution, thus providing a clear reference for the technique that best fulfills the needs of a particular simulation-based study.

Keywords: big data, concurrency, linearization, Modelica, multithreading, multiprocessing, parallelism, Python-Dymola Interface

1 Introduction

Motivation

Power system studies rely heavily on simulations. Computer-based analyses have been performed in the power system industry since the early '60s. A power system simulation, however, is a computationally expensive task, even considering today's computing power (Milano, 2010). The more complex the system is, the more resource-intensive a simulation will become.

The inherent complexity of power systems is increasing dramatically due to renewable energy integration, which introduces a large amount of stochasticity in the analysis stage. There are two main approaches to introduce uncertainties in a power system study. One is to use a suitable probabilistic uncertainty model (Aien et al., 2016). This could be not desirable because it would result in system models with exponentially increased complexity, which will be reflected in longer simulation execution time.

The second approach is to vary systematically parameters that are likely to be changed due to an external event or uncertain phenomena. This would mean a large number of simulations of "normal" complexity that must be carried on to obtain a full picture of the possible operating conditions of a power system under uncertain conditions. Nevertheless, if the number of scenarios required to construct a comprehensive picture of the power system is too large, a bottleneck would occur due to the extensive simulation requirements and the limited available computing power.

This paper tackles the problem of optimizing execution time for a large number of simulation scenarios. The advantages of Modelica-based models for power system simulation have been previously shown by (Vanfretti et al., 2016) and (Baudette et al., 2018); a key one being the ability to linearize complex models, which is usually not available in most production grade power system simulators.

Thus, we aim at linearizing an electric grid model around a fixed equilibrium point for 20000 different contingency conditions (referred to as scenarios) generated thanks to a Python routine. The output of the program is the **A**, **B**, **C** and **D** matrices resulting from a Jacobian linearization. This information can be used to assess the small-signal stability of the power system in the given operating condition (Milano, 2010). For this work, we pass each scenario to a Dymola instance within Python where we compute a linearization of the model thanks to the so-called Python-Dymola Interface (PDI) (Dassault Systèmes, 2018).

We need to perform multiple times the same simulation under different conditions. Given the significant amount of scenarios under consideration, we take advantage of Python parallel computing libraries such as `Threading` and `Multiprocessing` (Python Software Foundation, a,b) to reduce execution time by optimizing the use of the available computing resources. In other words, we simultaneously start/run more than one simulation to minimize the overall computing time. This kind of needs also appears in other areas (e.g., to generate realistic cyber-attack data (Filonov et al., 2017), and to produce data to train a

neural network for fault and anomaly detection (Lee et al., 2015)). We benchmark our approach against the traditional sequential execution method to show the remarkable advantages of the concurrency/parallel computing approach.

Contribution

As mentioned before, our work is relevant from a power system perspective as it reduces drastically the required time to analyze automatically a large number of simulation scenarios. There are many applications, however, that can be benefited by replicating our approach (e.g., applications where synthetic large-scale data from simulation needs to be generated for Machine Learning applications). Therefore, the main contributions of this paper are the following:

- Implementation of code routines to perform Dymola simulations from Python using multithreading and multiprocessing approaches.
- Comparison of concurrent, parallel and sequential execution approaches in terms of CPU usage, processor heating, and memory utilization.

Paper Organization

This paper is structured as follows: Section 2 reviews the simulation process using Dymola from Python. In Section 3, we detail the main differences between multithreading and multiprocessing simulation. The principal results of our experiments are summarized in Section 4. Finally, the paper is concluded in Section 5.

2 Automatic Scenario Simulation

Our main goal is to reduce the computational time of massive scale small-signal analysis in a power system subjected to line contingencies. The base power system is the IEEE 14 bus system shown in Figure 1. A simulation scenario consists of simultaneously opening of k number of elements connecting two buses, shown in blue.

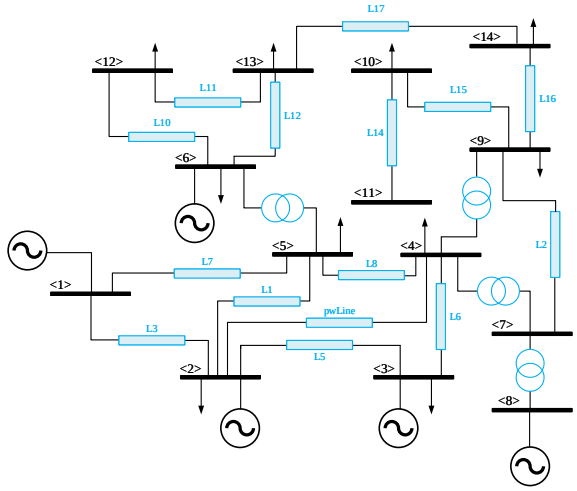

Figure 1. IEEE 14 bus model.

The power system is recreated in Dymola using the OpenIPSL library (Figure 2). Each contingency is simulated by increasing the impedance of the element to a very large value ($X_i \approx 10^{12}$ pu), not by removing the element from the model. This approach guarantees that the system is topologically the same. In other words, the dimension of the mathematical representation of the system is the same for all cases. A total of 20000 scenarios are considered.

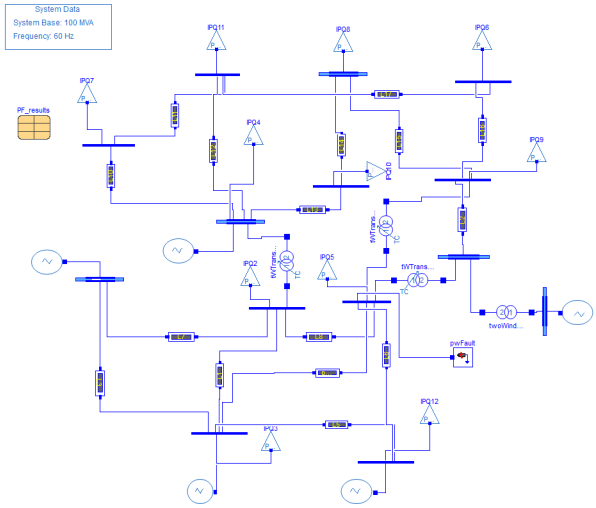

Figure 2. Implementation of the IEEE 14 bus model in Dymola using OpenIPSL.

Simulations are carried out using the Python-Dymola (PDI) Interface (Dassault Systèmes, 2018). This is an API that enables the execution of Dymola commands from Python. The first building block to use the PDI consists of instantiating a `DymolaInterface` object, passing the path of the dymola executable as an argument, loading the library required for the simulation, and opening the model itself.

```
# Path of the Dymola executable
path = "/opt/dymola-2019-x86_64/bin64/
    dymola.sh"
# Instantiating dymola object
dymola = DymolaInterface(path)
# Changing working directory
dymola.cd(".../Documents/DymolaLibraries")
# Loading OpenIPSL library
dymola.openModel("/OpenIPSL-1.5.0/OpenIPSL/
    package.mo")
# Opening IEEE 14 bus model
dymola.openModel(".../IEEE_14/IEEE14/
    package.mo")
```

After instantiating a `DymolaInterface` object in Python, we set up the simulation parameters. Each scenario is described by a list with strings that correspond to the name of the impedance parameter in the model that will be tripped. For example, the simulation scenario in which L1 and L7 are opened is specified by `['L1.X', 'L7.X']`. We generate the scenarios using an ad-hoc Monte Carlo two-stage sampling procedure which is out

of the scope of this paper.

The next step is to perform a linearization of the system around an equilibrium point specified by a fixed power flow condition. This is carried out by the Dymola built-in function `linearizeSystem`. The parameters of the simulation are passed in-line when calling the function in Python. The lines whose impedances are changed are indicated by a string generated from the scenario using a simple function called `trip_line`. The resulting `.mat` file containing the **A**, **B**, **C** and **D** matrices corresponding to a Jacobian linearization of the nonlinear model is saved, and opened if the linearization is successful.

In small-signal analysis, the eigenvalues of the system matrix **A** are of interest. For this reason, the matrix **A** is read from the `.mat` file, their eigenvalues are computed and saved into a Python data structure for further post-processing. Notice that these steps only take place if the linearization is successful as shown below:

```
def trip_line(model_name, scenario):
    return str(model_name + "(" + "=1e12, "
        .join(scenario) + "=1e12" + ")")

scenarios = [['L1.X', 'L2.X'],
        ['L1.X', 'L10.X', 'L15.X']]
model_name = "IEEE_14_Buses"
path_wd = ".../home/Tests/
    ModelicaConference/IEEE14"

# Counter for the number of simulations
a = 0

for scenario in scenarios:
    try:
        result = dymola.linearizeModel(
            trip_line(model_name, scenario)
            ,
              startTime = 0.0,
              stopTime = 0.0,
              fixedstepsize = 0.001,
              method = "Dassl",
              tolerance = 0.0001,
              resultFile = path_wd + "{}".
                  format(a))
        if result:
            A_dim = dymola.readMatrixSize(
                path_wd +              "{}.
                mat".format(a), "ABCD")
            A = dymola.readMatrix(path_wd +
                 "{}.mat".format(a),      "
                ABCD", A_dim[0], A_dim[1])
            eigs_model.append(sl.eig(A)[0])
            a += 1
            print("Linearization OK")
        except:
            print("Linearization failed")
```

This fundamental building block can be used to construct an automatic loop to perform an arbitrary number of linearizations. A direct implementation in this form, however, would be a sequential execution approach since each linearization is carried out after another one has been completed. The purpose of the next section is to discuss two more efficient ways of performing the task.

3 Multithreading and Multiprocessing

The purpose of this section is to introduce some concepts of the Computer Science jargon that are important to understand our simulation proposal. We begin by defining threads and processes, and then introduce the related concepts of concurrency and parallelism. Finally, we discuss briefly how both techniques are implemented in Python.

Threads and Processes

A *process* is a program that is in execution. In other words, it is the basic abstraction form in an operating system. Any program that is running can be understood as a process. On the other hand, a *thread* can be thought of as the minimal computational unit spawn in a processor. It is an entity that lives inside a process. Henceforth, all threads in the same process share the same data and system resources of the process.

Two different processes are independent of each other in the sense that they do not share memory addresses. Since threads are alive within a process, they are "cheap" to create in terms of resources and time. Processes, on the other hand, require more time and resources to be created, but its independence could be advantageous for some applications.

Multithreading and Multiprocessing

Two or more tasks are said to be *concurrent* if they start at the same time but are not executed simultaneously. Likewise, two programs are said to be in *parallel* if they are being executed at exactly the same time.

Multithreading consists of the concurrent execution of algorithms in several execution threads. In other words, if one thread is waiting for the completion of some calculation or for some input/output prompt, another thread can be executed while the first is paused. Hence, multithreading is a form of concurrent computing. *Multiprocessing*, on the other hand, is the simultaneous execution of processes.

In Python, multithreading can be carried out using the class `Thread` or the library `Threading`. Multiprocessing benefits from the function `Pool` of the `Multiprocessing` library and the `Process` class. A comparison between both approaches is presented in Table 1.

4 Results

In this section, we present a performance evaluation of three methods for massive scale model linearization simulations for the IEEE 14 bus model under different contingency scenarios. The execution procedures that will be benchmarked against each other are sequential, multithreading (concurrency) and multiprocessing (parallel).

In the sequential or serial fashion, each scenario is passed to a Dymola instance, which performs a linearization routine. The next scenario is fed when the simulation

	Definition	Creation	Execution
Multithreading	Creation of multiple threads inside a single process to increase computing power by concurrency	Economical (regarding time and resources)	Concurrent. Several threads in a single process
Multiprocessing	Addition of independent processes in CPUs to increase computing power by simultaneous (parallel) execution of tasks	Intensive (regarding time and resources)	Parallel. Multiple processes in different CPUs

Table 1. Comparison between multiprocessing and multithreading.

is completed. On the other hand, both multithreading and multiprocessing are data-based parallelism techniques. It means that data containing the information of the different scenarios is divided into as many batches as threads/processes will be created. Then, each thread/process is responsible for carrying out a smaller amount of simulations. The execution of threads is concurrent, not parallel, while processes run simultaneously.

The number of mini-batches depends on the number of threads/process that would be created. For multiprocessing, a rule of thumb is to create as many processes as physical processors are available minus one. That is, if a computer has 12 physical cores, we shall create 11 processes. For multithreading, even though the maximum number of threads is theoretically limited by the available computing resources, an extremely large amount can slow down execution process and make multithreading slower than a sequential approach (what is known as *parallel slowdown*). We select 11 threads to be able to compare this approach with multiprocessing in almost the same conditions.

In Table 2, we list the characteristics of the computer used to run the simulations.

	Characteristic
Operating System	Ubuntu Server 16.10
RAM	128 GB
Processor	Intel(R) Xeon(R) CPU E-1650 v4 12 Cores @ 3.60 GHz 15 MB Cache
Storage	1 TB SSD
Graphics Cards	4 x NVIDIA GTX 1080 Ti (CUDA Capable) 11 GB GDDR5X (each)
Dymola Distribution	Dymola 2019
Python Release	3.6.8

Table 2. Characteristics of the computer employed to run the linearization task.

The metrics we used to evaluate the performance are execution time, CPU utilization, processor temperature and virtual memory (VRAM) usage. This was determined by a Python script that measures and registers every second the metrics related above. We perform a detailed analysis of each feature later in this section.

In Table 3 we present a summary of the most relevant statistics. We see that execution time is drastically reduced by the use of concurrent and parallel strategies. Multithreading approach, however, raises the temperature of the processor the most, implying that it requires the largest amount of energy. We see as well that virtual memory usage is almost the same for all methods. The code to replicate this experiment can be found in GitHub[1].

4.1 Execution Time

Execution time was measured by registering the time it takes for the program to complete all requested linearization routines. Results are shown in Figure 3. Note how the use of parallel computing techniques reduces the execution time in approximately 25 h. This is a reduction of 63% taking the sequential procedure as a reference.

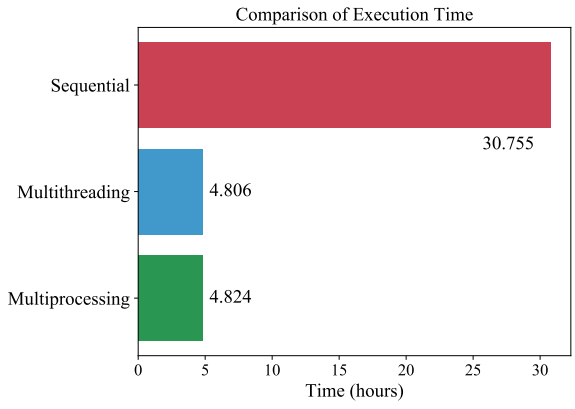

Figure 3. Comparison of execution time (in hours) of the different execution methods.

4.2 CPU Utilization

CPU utilization is the percentage of the CPU that is being employed in a specific core at a given time. Mean CPU utilization is the average of all cores. This plot is presented in Figure 4.

[1] https://github.com/ALSETLab/Massive-Simulation-Automation-for-Modelica-Dymola

	Sequential	Multithreading	Multiprocessing	
Execution Time	30.755 h	4.806 h	4.824 h	
CPU Utilization (%)	12.00%	89.52%	89.05%	Mean
	9.97%	89.79%	89.31%	Median
	3.976%	3.83%	3.53%	SD
Temperature	47.01°C	61.61°C	59.06°C	Mean
	47.00°C	62.00°C	59.00°C	Median
	1.87°C	1.51°C	1.48°C	SD
Virtual Memory (%)	10.79%	12.08%	11.83%	Mean
	10.7%	12.2%	11.80%	Median
	1.13%	1.09%	1.15%	SD

Table 3. Summary of statistics for the different performance metrics (mean, median and standard deviation (SD).

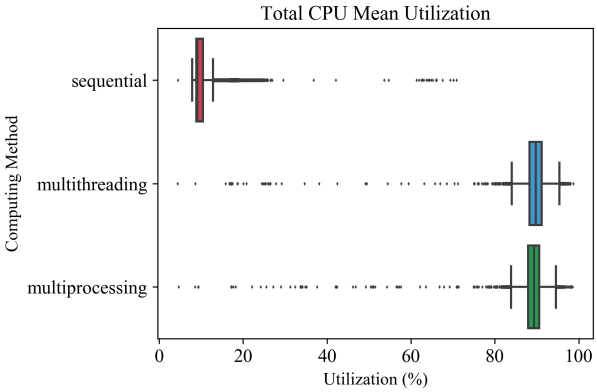

Figure 4. Mean CPU utilization in each method.

Multithreading and multiprocessing make better use of the resources, reaching mean utilization close to 90%. The sequential approach, on the other hand, does not employ all the available computing capacity of the machine, reaching an average usage of 12%. We conclude that multiprocessing is the best since mean usage is the highest, meaning that the use of CPU resources is maximized.

Now, we analyze the performance with respect to each core for each technique. We begin with the sequential approach (Figures 5 and 6).

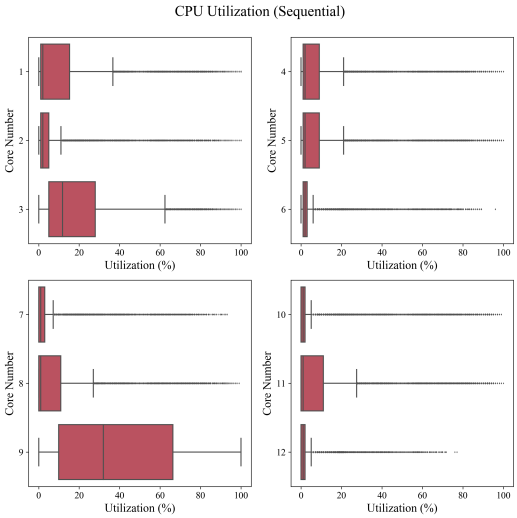

Figure 5. CPU utilization per core for the sequential execution approach.

In serial execution, we see that most of the computation is carried out by core 9 with an average use of 38.45%. The remaining cores are mostly unused.

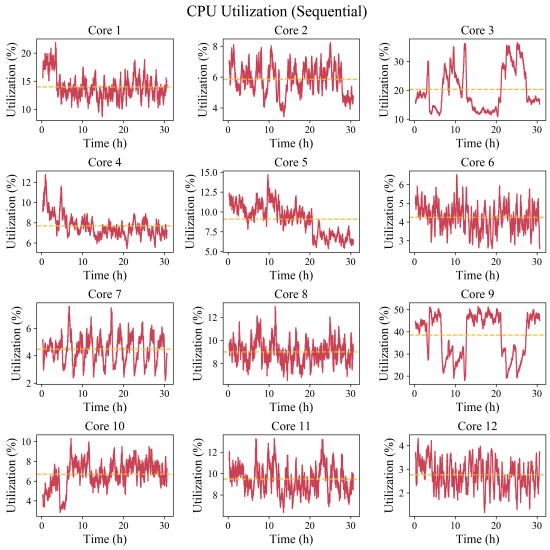

Figure 6. Time profile of CPU utilization for serial execution.

For multithreading, we get the data shown in Figure 7. Core 12 is supposed to take care of the operating system, while the remaining 11 cores are assigned by Python to run simulation threads.

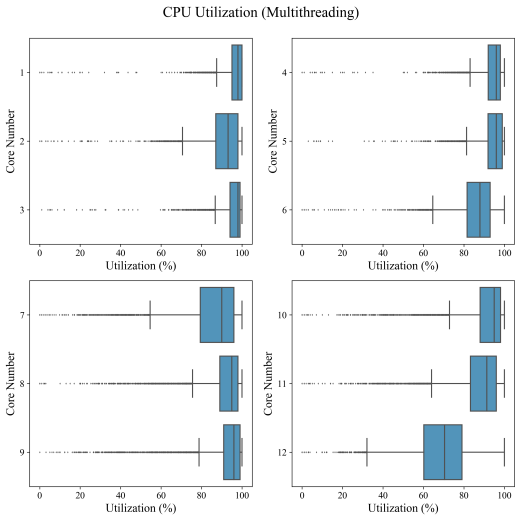

Figure 7. CPU utilization per core for the multithreading approach.

In Figure 8 below, we see the time behavior of the uti-

	Core						
	1	2	3	4	5	6	
Sequential	13.96%	5.85%	20.31%	7.68%	9.07%	4.25%	Mean
	2.00%	2.00%	11.80%	2.00%	2.00%	2.00%	Median
	25.41%	25.41%	21.52%	13.02%	16.21%	8.22%	SD
Multithreading	96.44%	91.03%	95.91%	94.21%	94.19%	86.27%	Mean
	98.00%	93.10%	98.00%	96.00%	96.00%	87.90%	Median
	5.71%	8.89%	5.69%	6.17%	6.30%	9.18%	SD
Multiprocessing	95.86%	87.26%	93.80%	94.02%	94.03%	84.50%	Mean
	97.50%	89.00%	96.00%	96.00%	96.00%	86.00%	Median
	5.71%	9.81%	7.12%	6.08%	6.40%	9.46%	SD
	7	8	9	10	11	12	
Sequential	4.47%	9.00%	38.45%	6.69%	9.47%	2.76%	Mean
	1.00%	1.00%	32.0%	1.00%	1.00%	1.00%	Median
	9.75%	18.65%	31.42%	17.22%	19.09%	6.18%	SD
Multithreading	85.33%	91.15%	92.29%	91.03%	87.57%	68.86%	Mean
	90.00%	94.90%	96.00%	95.00%	91.30%	70.4%	Median
	14.66%	10.53%	11.09%	11.15%	12.20%	14.23%	SD
Multiprocessing	85.95%	92.71%	92.25%	90.29%	88.37%	69.55%	Mean
	90.00%	95.10%	96.00%	94.00%	92.00%	71.00%	Median
	13.93%	8.67%	9.84%	11.09%	11.43%	13.73%	SD

Table 4. CPU usage statistics per core (mean, median and standard deviation (SD) from top to bottom, respectively).

lization of each core. The signals show that there is no symmetric distributions among all cores. Core 12, with an average utilization of 71.0%, is responsible for the operating system and register the performance data. It does not run any simulation thread.

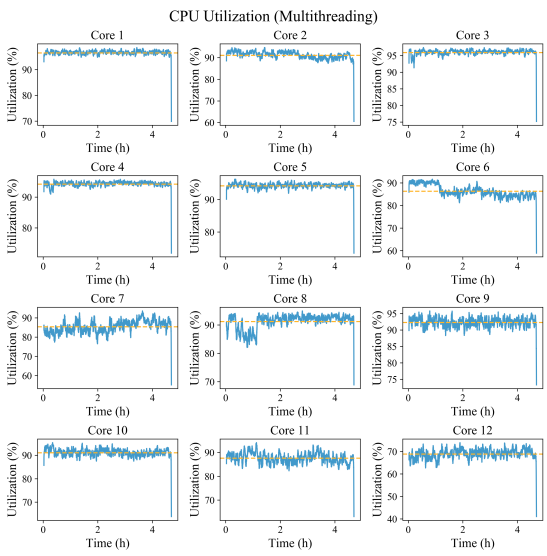

Figure 8. CPU utilization per core for the multiprocessing execution approach.

Finally, in Figure 9 we show the performance plot for the multiprocessing approach in regards to CPU utilization. The usage per core is more or less the same as in the case of multithreading (see Table 4) and significantly higher in comparison to the serial method. Core 12 is not running any Dymola process but rather is taking care of the operating system and the measurement program like happens with multithreading.

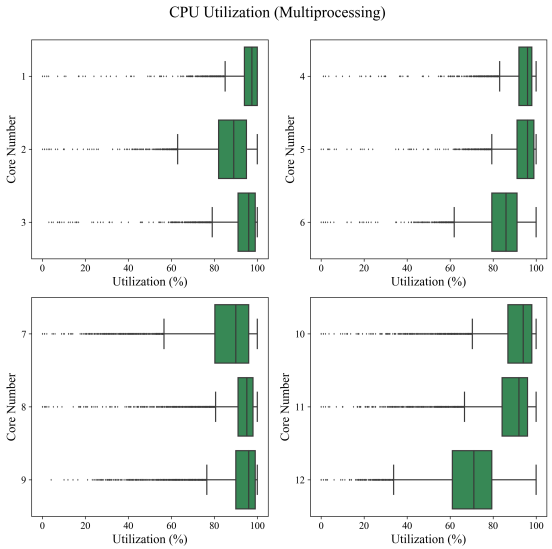

Figure 9. CPU utilization per core for the multiprocessing execution approach.

The time series data describing the CPU usage in the multiprocessing solution (Figure 10) shows a similar behavior than that of the multithreading approach. The average utilization, however, is much larger than in the serial approach which, as expected, is due to a more optimal usage of the available computing resources.

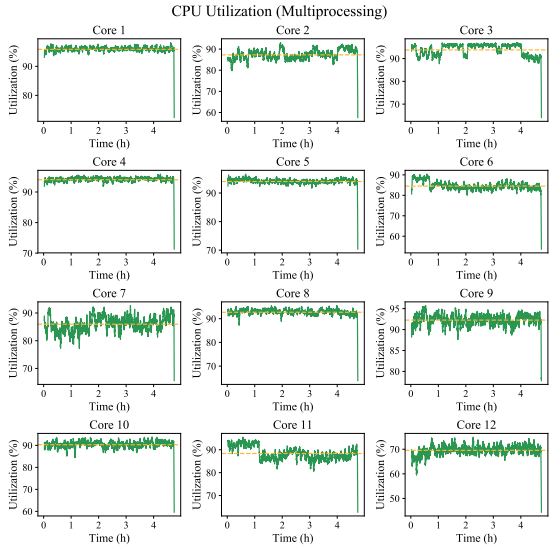

Figure 10. CPU utilization per core for the multiprocessing execution approach.

4.3 Power Consumption

Power consumption is related to the CPU's temperature. The higher the temperature of the cores, the more power it is consumed throughout the execution of an algorithm.

Temperature is measured using the embedded analog sensor in the Intel Processor. An important assumption is that we neglect the effect of the room temperature, since it is maintained at a (roughly) constant value by an air conditioning system that helps keeping in a normal operating condition the rest of the hardware in the laboratory. The most important statistics are resumed in Table 5

	Temperature Statistics	
Sequential	47.01°	Mean
	47.00°	Median
	1.8756°	SD
Multithreading	61.61°	Mean
	62.00°	Median
	1.5122°	SD
Multiprocessing	59.06°	Mean
	59.00°	Median
	1.4855°	SD

Table 5. Most relevant temperature statistics.

The sequential procedure keeps the processor temperature at its lowest. This is reasonable since it is the method that forces the less the computer to its capacity boundary, hence requiring less energy in execution.

The most interesting detail, however, is that there is a difference multithreading and multiprocessing showing that multithreading warms up more the processor chip (around 3°C). This fact leads us to conjecture that multithreading execution drains more power than the parallel-ing approach. We can detail this further in Figure 11.

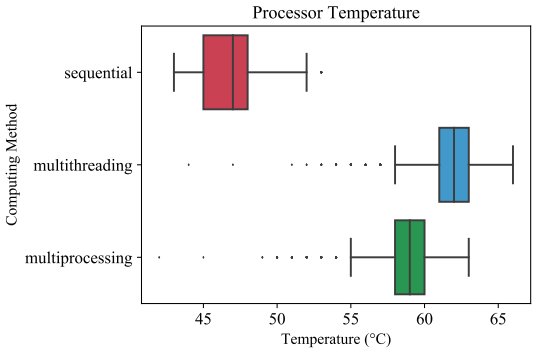

Figure 11. Boxplot for temperature measurements.

The time series of the temperature measurements in Figure 12 show how the cooling system of the CPU (Corsair) tries to regulate a temperature for both the concurrent and the multiprocessing techniques. The values, however, are more spread for multithreading which may take into account the fact that concurrent execution can be less effective than parallel execution since only one thread is being run at a given instant, and the CPU has to manage all threads (i.e., determine what is to be run next) simultaneously.

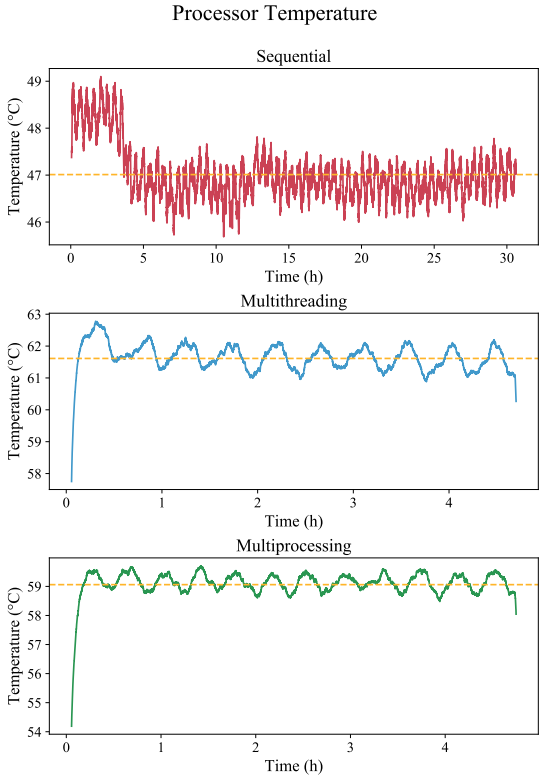

Figure 12. CPU utilization per core for the multithreading execution approach.

4.4 Virtual Memory (VRAM)

The last performance metric we analyzed is the percentage of virtual memory (VRAM) used. This accounts not

only for program execution but also for the data being generated as the simulation advances and more scenarios are considered. Results are shown in Figure 13 and Table 6. We see that as the simulation progresses, more memory is used. This is expected since each time a scenario is considered, more data is generated.

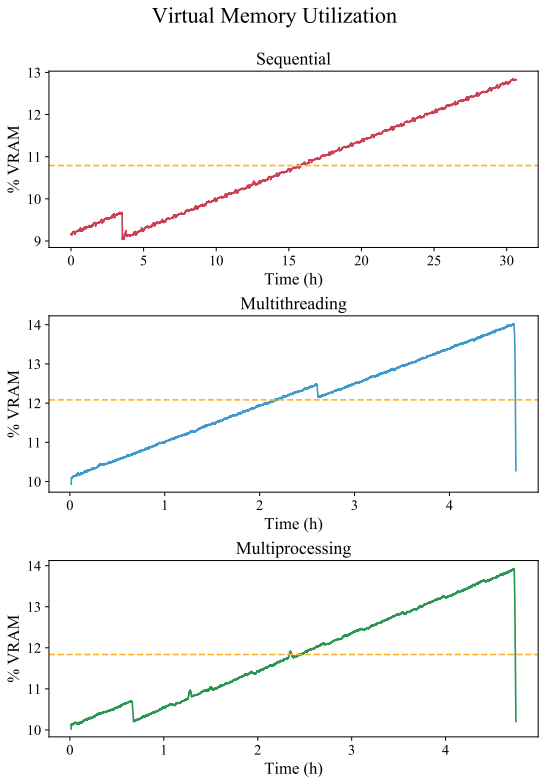

Figure 13. Virtual memory (VRAM) utilization throughout execution time.

Multithreading and multiprocessing require a slight larger amount of memory because of the number of Dymola instances existing simultaneously. The difference, however, is not that considerable compared with the sequential approach.

Note that in all cases there is a small discontinuity due to a sudden clean in memory. This might be due to a process such as a garbage collector running in the background in the Linux operating system.

	VRAM	
Sequential	10.79%	Mean
	10.7%	Median
	1.138%	SD
Multithreading	12.08%	Mean
	12.2%	Median
	1.098%	SD
Multiprocessing	11.83%	Mean
	11.8%	Median
	1.154%	SD

Table 6. Virtual memory (VRAM) statistics.

5 Conclusions

Our experiments showed that a concurrency and a parallelism-based approach improve radically the execution time for massive simulations in Dymola compared with a traditional sequential approach. The results for both the multiprocessing and multithreading solutions are quite similar in regards to CPU usage. This may be due to the fact that both solutions were coded using practically the same code syntax. The `threading` library was employed for concurrent simulations, whereas the `Pool` class was used for the multiprocessing solution. Nevertheless, an important difference between both methods is the temperature (related to power consumption) and the memory usage, which are found to be lower for the actual parallel approach.

To the best knowledge of the authors, there are not many publications computing numerical scores for benchmarking data-based parallelism approaches (see (Shapiro) for a list of common comparison metrics, and (Bienia et al., 2008) for a quantitative description of a low-level multithreading performance evaluation). For this reason, we define a comparison metric to quantify the performance of each computational strategy.

Let $\bar{\mathbf{t}}$, $\bar{\mathbf{c}}$, $\bar{\mathbf{T}}$ and $\bar{\mathbf{m}}$ be vectors containing the median values of the execution time, CPU utilization, processor temperature and VRAM usage of each of the three methods, respectively

$$\begin{aligned} \bar{\mathbf{t}} &= \begin{bmatrix} \bar{t}_\mathrm{s} & \bar{t}_\mathrm{mt} & \bar{t}_\mathrm{mp} \end{bmatrix}^T \\ \bar{\mathbf{c}} &= \begin{bmatrix} \bar{c}_\mathrm{s} & \bar{c}_\mathrm{mt} & \bar{c}_\mathrm{mp} \end{bmatrix}^T \\ \bar{\mathbf{T}} &= \begin{bmatrix} \bar{T}_\mathrm{s} & \bar{T}_\mathrm{mt} & \bar{T}_\mathrm{mp} \end{bmatrix}^T \\ \bar{\mathbf{m}} &= \begin{bmatrix} \bar{m}_\mathrm{s} & \bar{m}_\mathrm{mt} & \bar{m}_\mathrm{mp} \end{bmatrix}^T \end{aligned} \tag{1}$$

where the subscript 's' stands for sequential, 'mt' for multithreading and 'mp' for multiprocessing. We underline that we employ the median to avoid the effect of outliers in the score. Moreover, let us write the element-wise product of $\bar{\mathbf{t}}$ and $\bar{\mathbf{T}}$ as

$$\bar{\mathbf{t}} \odot \bar{\mathbf{T}} = \begin{bmatrix} \bar{t}_\mathrm{s}\bar{T}_\mathrm{s} & \bar{t}_\mathrm{mt}\bar{T}_\mathrm{mt} & \bar{t}_\mathrm{mp}\bar{T}_\mathrm{mp} \end{bmatrix}^T. \tag{2}$$

The performance score for the ith method where $i \in \{\mathrm{s}, \mathrm{mt}, \mathrm{mp}\}$ is defined as

$$\begin{aligned} [\mathrm{Score}]_i = \Bigg[& 0.4 \left(\frac{\min(\bar{\mathbf{t}})}{\bar{t}_i} \right) + 0.3 \left(\frac{\bar{c}_i}{\max(\bar{\mathbf{c}})} \right) \\ & + 0.2 \left(\frac{\min(\bar{\mathbf{t}} \odot \bar{\mathbf{T}})}{\bar{t}_i \bar{T}_i} \right) + 0.1 \left(\frac{\min(\bar{\mathbf{m}})}{\bar{m}_i} \right) \Bigg]. \end{aligned} \tag{3}$$

The maximum value of the score is 1.0. The greater the score, the better the performance of the method. This metric intends to reward reduction of execution time (40%), optimal use of CPU resources (30%), energy efficiency understood as power per time (20%), and minimization of memory usage (10%). The results are summarized in Table 7. We conclude that multiprocessing is the best performing method with respect to this multi-criteria metric.

	Sequential	Multithreading	Multiprocessing
$\frac{\min(\bar{\mathbf{t}})}{\bar{t}_i}$	0.1563	1.0000	0.9963
$\frac{\bar{c}_i}{\max(\bar{\mathbf{c}})}$	0.1110	1.0000	0.9947
$\frac{\min(\bar{\mathbf{t}} \odot \bar{\mathbf{T}})}{\bar{t}_i \bar{T}_i}$	0.1969	0.9552	1.0000
$\frac{\min(\bar{\mathbf{m}})}{\bar{m}_i}$	1.0000	0.8770	0.9068
Total Score	0.2352	0.9787	0.9876

Table 7. Score results for each computing method.

Acknowledgments

This work was funded in part by New York Power Authority (NYPA) and the New York State Energy Research and Development Agency (NYSERDA) through the Electric Power Transmission and Distribution (EPTD) High Performing Grid Program Program under agreement 137940.

References

Morteza Aien, Ali Hajebrahimi, and Mahmud Fotuhi-Firuzabad. A comprehensive review on uncertainty modeling techniques in power system studies. *Renewable and Sustainable Energy Reviews*, 57:1077–1089, may 2016. ISSN 13640321. doi:10.1016/j.rser.2015.12.070. URL https://linkinghub.elsevier.com/retrieve/pii/S1364032115014537.

Maxime Baudette, Marcelo Castro, Tin Rabuzin, Jan Lavenius, Tetiana Bogodorova, and Luigi Vanfretti. OpenIPSL: Open-Instance Power System Library - Update 1.5 to iTesla Power Systems Library (iPSL): A Modelica library for phasor time-domain simulations. *SoftwareX*, 7:34–36, jan 2018. ISSN 23527110. doi:10.1016/j.softx.2018.01.002.

Christian Bienia, Sanjeev Kumar, and Kai Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors. In *2008 IEEE International Symposium on Workload Characterization*, pages 47–56. IEEE, oct 2008. ISBN 978-1-4244-2777-2. doi:10.1109/IISWC.2008.4636090. URL http://ieeexplore.ieee.org/document/4636090/.

Dassault Systèmes. *Dymola User Manual*. 2018.

Pavel Filonov, Fedor Kitashov, and Andrey Lavrentyev. RNN-based Early Cyber-Attack Detection for the Tennessee Eastman Process. sep 2017. URL http://arxiv.org/abs/1709.02232.

Dongkyu Lee, Byoungdoo Lee, and Jin Woo Shin. Fault Detection and Diagnosis with Modelica Language using Deep Belief Network. pages 615–623, sep 2015. doi:10.3384/ecp15118615. URL http://www.ep.liu.se/ecp/article.asp?issue=118{%}26article=66.

Federico Milano. *Power System Modelling and Scripting*, volume 54 of *Power Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-13668-9. doi:10.1007/978-3-642-13669-6. URL http://link.springer.com/10.1007/978-3-642-13669-6.

Python Software Foundation. Threading - thread-based Parallelism, a.

Python Software Foundation. Multiprocessing - Process-based Parallelism, b.

Daniel Shapiro. Recent Progress in Multiprocessor Thread Scheduling. URL http://www.site.uottawa.ca/{~}mbolic/ceg4131/Daniel{_}report.pdf.

L. Vanfretti, T. Rabuzin, M. Baudette, and M. Murad. iTesla Power Systems Library (iPSL): A Modelica library for phasor time-domain simulations. *SoftwareX*, 5:84–88, 2016. ISSN 23527110. doi:10.1016/j.softx.2016.05.001. URL https://linkinghub.elsevier.com/retrieve/pii/S2352711016300097.