Master in High Performance Computing

Calibration of the GEOtop model using evolutionary algorithms on supercomputers

Supervisor(s):
Giacomo Bertoldi,
Alberto Sartori

Candidate:
Stefano Campanella

6th edition
2019–2020

CONTENTS

1 Summary and Outline **2**

2 Acknowledgements **3**

3 GEOtop calibration **4**
3.1 Brief Introduction to GEOtop 4
3.2 Model Calibration 5

4 The need for HPC **7**
4.1 The Curse of Dimensionality 7
4.2 Scalability of Evolutionary Algorithms 8
4.3 Scaling in Theory and in Practice 8

5 Derivative-free Optimization **10**
5.1 Random Search 10
5.2 Evolutionary Algorithms 11

6 Tools **12**
6.1 The Python Programming Language 12
6.2 The Jupyter Ecosystem 13
6.3 GEOtoPy 15
6.4 Nevergrad 17
6.5 High-Performance Computing in Python using Dask 18

7 Implementation **21**
7.1 Model and Objective Function 21
7.2 A Closer Look at the Optimization Loop 22

8 Example GEOtop Calibration Report **26**

9 Scaling Analysis and Modeling **33**
9.1 Scaling and Efficiency 34
9.2 Linear Scaling Model 39
9.3 Models for Large Numbers of CPUs 43

10 Conclusions **49**

11 Final Thoughts Beyond the Scope of the Thesis **50**
11.1 Apology of the Thoughtful Dabbler 50
11.2 Rage Against Machine Learning 51

11.3 Math is the Ultimate Javascript Framework 52

Bibliography **54**

Master in High-Performance Computing, 2019/20 edition

Author: Stefano Campanella

Supervisors: Giacomo Bertoldi, Alberto Sartori

SISSA and ICTP

CHAPTER

ONE

SUMMARY AND OUTLINE

This thesis addresses my final project for the 2019/20 edition of the Master in High-Performance Computing at SISSA and ICTP.

Earth-system and environmental models calibration is a complex, computationally intensive task. At present, there is no general theory of model calibration, but instead a large collection of methods, algorithms and case studies. As a result, calibration is often more an art than a science: one must make several discretionary choices, guided more by his own experience and intuition than by the scientific method.

One of the challenges is the large number of parameters involved. For this reason, preliminary sensitivity analysis may be used to reduce this number and select the relevant parameters. Still, the computational load of sensitivity analysis and calibration is high.

In this work I used High-Performance Computing solutions to calibrate GEOtop [RBO06][EGDallAmicoR14], a complex, over parameterized hydrological model. I used the derivative-free optimization algorithms implemented in the Facebook Nevergrad Python library [RT18], and run them on the Ulysses v2 HPC cluster, thanks to the Dask framework [Tea16].

GEOtop has been used to simulate the time evolution of variables as soil water content and evapotranspiration of mountain agricultural sites in South Tyrol with different elevations, land cover (pasture, meadow, orchard), and soil types. In these simulations GEOtop solved the energy and water budget equations on a one-dimensional domain, i.e. on a thin column of soil and neglecting the lateral fluxes. Even in the simplified case of homogeneous soil, one has tens of parameters. These parameters control the soil and vegetation properties, but only a few of them are experimentally available, hence the need for calibration.

The computational aspects of GEOtop calibration have been examined, and the important issue of robustness against model convergence failures has been addressed. Finally, the scaling of calibration time has been measured up to 1024 cores.

The outline of the thesis is the following:

1. **Introduction and motivations.** Where I introduce relevant information about the GEOtop model. I also discuss the problem of GEOtop calibration, and the need for High-Performance Computing.
2. **Problem, methodology and implementation.** Where I state the problem in mathematical terms, but without mathematical rigour. Afterwards, I discuss the tools and implementation details of calibration.
3. **Results and conclusions.** Finally, I present the results and scaling of calibration, focusing on the HPC content.

CHAPTER

TWO

ACKNOWLEDGEMENTS

I would like to thank Giacomo Bertoldi, for giving me this opportunity, Alberto Sartori for his help during the whole master, Ing. Piero Calucci for his assistance, Stefano Salon for his advice, and Alessandro Vuan for his patience.

I would also like to thank Nuno Carvalhais for the conversations we had and all the good ideas he gave me (all the bad ones are mine of course).

Finally, I would like to thank Fulvio and Erica for the good times we had together during this strange, difficult, transformative year, Cosimo for his tenacious encouragement and Giorgio, aka “Il Sommo”, whose discreet example and savvy words have so much influence on me; to him goes the merit (or demerit, the future will tell) of suggesting me to apply for the MHPC.

The inspiration of this work comes from the GEOtopOptim and geotopbricks R packages by E. Cordano [CDF15][Cor15].

G. Bertoldi work on the GEOtop model was partly supported by the European Regional Development Fund, Operational Programme Investment for growth and jobs ERDF 2014-2020 under Project number ERDF1094, Data Platform and Sensing Technology for Environmental Sensing LAB – DPS4ESLAB.”

Experimental observations used in this work were collected thanks to the project MONALISA - Monitoring key environmental parameters in the Alpine Environment involving science, technology and application, financed by the Special Research Fund of the Province of Bolzano and in the framework of the LTSER Platform Matsch | Mazia, which belongs to the national and international Long-Term Ecological Research Networks (LTER-Italy, LTER-Europe and ILTER).

The research reported in this work was supported by Eurac, OGS, and CINECA under HPC-TRES program award number 2019-33.

CHAPTER

THREE

GEOTOP CALIBRATION

> When a program grows in power by an evolution of partially-understood patches and fixes, the programmer begins to lose track of internal details and can no longer predict what will happen—and begins to hope instead of know, watching the program as though it were an individual of unpredictable behavior.
>
> —Marvin Minsky, *Why programming is a good medium for expressing poorly understood and sloppily formulated ideas*

3.1 Brief Introduction to GEOtop

GEOtop is a model of the mass and energy balance of the hydrological cycle aimed for simulations of small catchments. It is a distributed model and can simulate the evolution of snow cover, soil temperature, and water content. GEOtop takes into account vegetation processes, such as evapotranspiration, to correctly describe the water and energy exchange with atmosphere.

A GEOtop simulation requires some input data, parameters and settings. The meteorological data strictly necessary to run the model consist of: air temperature, relative humidity (air water content, air vapor pressure, or dew point), wind speed, shortwave radiation, pressure, and precipitation. The meteorological time series must come from at least one station with a resolution of at least six hours.

Parameters can be divided into surface parameters, which values are single numbers for each point of the catchment, and soil parameters, that may vary with depth. The first ones are related to energy fluxes, as the albedo, and vegetation properties, as vegetation height and leaf area index, which may vary with time. The seconds can be either thermodynamical properties of the soil, as its thermal conductivity and capacity, or hydraulic properties. The latter are crucial for determination of the soil moisture content, and some of them appear in phenomenological relations which are highly non-linear, as the Van Genuchten equation [VG80]. Small changes in these parameters correspond to very different behaviours of soil retention. Putting all together, there are around thirty parameters that describe a single point of the simulated catchment. However, soil parameters are arrays since different layers of soil can have different properties. This means that the total number of values that can be used to characterize a point of a basin, including default values, can exceed one hundred.

The core of a simulation is the solution of the system of coupled partial differential equations that describes the flow and diffusion of water, and thermodynamical properties of soil. GEOtop solves a finite difference approximation of this system of equations. It uses a fixed time-step

length and a three-dimensional grid, whose upper bound is given by a digital elevation model of the catchment, and the lower one is at a specified varying depth.

The two main equations are the water and energy balance equations with appropriate source and sink terms. The water balance equation also includes a diffusive term, as the soil is a porous medium. At each time-step, GEOtop uses an iterative method to solve these discretized equations. The number of iterations is determined by the residual, which must be under a user-defined threshold. Since the number of iteration is not fixed, the number of CPU cycles required to simulate the same time interval can vary and even diverge. GEOtop has several settings to limit the number of iterations of its internal routines. Nonetheless, it may be necessary to use an external timeout when dealing with batch executions of GEOtop.

For more information on the GEOtop model please refer to the paper by S. Edrizzi et al [EGDallAmicoR14] and to the documentation available at the model homepage www.geotop.org. In this work, I will refer to the version 3.0, which has been written in C++ and is available as a branch of the main project on GitHub. More information on GEOtop code refactoring and benchmarks can be found in [B+18].

3.2 Model Calibration

As discussed in the previous section, GEOtop has many free parameters. Their large number is the result of assembling many different submodels of various nature. Not all of them can be directly measured with an instrument or inferred in some other way than calibration. This fact is especially actual for the ones found in phenomenological relations, think for example to the Van Genuchten equation for soil retention. Furthermore, even when there is a straightforward experimental procedure to determine them, their value is affected by uncertainties, which could be large.

One could think that default values of the parameters exist, which describe the average system, and that any further adjustment of the parameters describes small deviations from it. However, because of the strong non-linearity of the associated processes, this is not the case, and parameters do not simply need fine-tuning.

Indeed, the outputs of a simulation can change wildly, and without carefully chosen values of the parameters, the results are entirely meaningless. Hence, the predictive power of GEOtop is strictly related to good calibration.

What it means to calibrate the model? The intuitive idea of “finding the values of the inputs with best outputs” contains a certain degree of hand waving. Practically, one should answer the following questions:

1. How do I compare the time series produced by the model with the experimental data?
2. How do I choose which input parameters I need to calibrate?
3. Which optimization algorithm and hyperparameters should I choose?
4. How much computational resources do I need?
5. How long should I wait for the results?
6. Are they meaningful?
7. Finally, is there a way to get the same or better results with fewer resources, i.e. waiting less or using less computational power?

The task of model calibration is more an art than a science. Still, it would be useful to have hints and heuristics. The analyses in this work answer only a few of the previous questions. However, the developed code enable making further experiments and empirical studies on the subject. Let's consider the challenges that calibration entail.

The multitude of parameters translates into dimensionality curse, that is the exponential growth of the volume of the search space with respect to the number of parameters. However, the dimensionality curse is not an obstacle per se, take for example neural networks. Yet, neural networks have two peculiar properties. On the one hand, the derivatives of the objective function with respect to the model parameters can be easily calculated. On the other, it turns out that wherever you start from in the search space, there is always a good set of parameters nearby[1]

Leaving out categorical parameters, in the case of GEOtop, the first point is not a real impediment. In principle, we could use a numerical differentiation scheme (supposing that there are no threats of numerical instability and that the objective function is smooth enough). However, the second point has no similar in GEOtop.

In pictorial form, if one could put himself in the parameters space of GEOtop, and look at the cost function, he would see hills and canyons, craters where the model crashes, swamps where`it doesn't converge, mirages of oasis with unphysical parameters, and deadly desert plateaus where one moves from meaningless outputs to equally meaningless outputs. In this lumpy and bumpy landscape, moving towards the direction of the steepest descent would lead, in the best scenario, to a useless local optimum[2].

Hence, if we want to find the holy grail of global optimum, we need to roam and wander, jumping here and there, with increasing confidence on our next guess as we grasp some (statistical) knowledge of the shape of the objective function. However, this process is very time-consuming: for the kind of simulations with which we are involved, each sampling takes about one minute, hence serial computations are not feasible.

The results of calibration strongly suggest that a global optimum does not exist. Instead, the objective has a plethora of equivalent local minima. This is an interesting feature of the model, and a strong indication that it is over-parametrized. Indeed, this open the possibility for further analysis of the data collected during calibration. It could allow getting useful information for a better understanding of the model behaviour and future models improvements.

[1] It is implausible that local optima exist in such high dimensional spaces. Given the gargantuan dimensionality of models like GTP3, the concept of direction, or distance assume a statistical meaning: just by chance, there will always be a direction along which you could move to smaller values of the cost function.

[2] However, at the end of a good calibration, we might have a good prior. In this case, it would make sense to perform a local optimization search. In principle, this would boost the performance of the calibration strategy. Unfortunately, there was no time to develop this idea.

CHAPTER

FOUR

THE NEED FOR HPC

> The old joke is "HPC is the art of taking a CPU-bound computation and making it I/O bound"
>
> —dnautics, comment on Hacker News

4.1 The Curse of Dimensionality

Let's consider an imaginary calibration. The first problem is the discretization of the parameters space: i.e. establish what is the scale at which changes of a parameter produce relevant variations of the output of the model. Note that this quantity may not be constant, meaning that small changes can produce huge differences in a region of the parameters space and negligible in another. Furthermore, the parameters might take values on a domain with complicated shape. Paradoxically, even before doing the calibration, we would need precise knowledge of the objective function to design calibration.

Anyway, in the simplest possible model, each parameter can be quantized with one bit, and it is enough to sample just two of its values. The reader will readily recognize the resemblance with the wheat and chessboard problem or viral disease spread. With only 20 parameters (a perfectly reasonable number for GEOtop, if not small), a brute force search would need $2^{20} \approx 10^6$ evaluations of the objective function. Hence, a calibration of this toy model with 20 parameters and one minute per objective evaluation would take two years on a single CPU, and 2000 years with 30 parameters.

The moral of the story is twofold:

1. whatever algorithm we choose, it must significantly outperform grid search, and
2. this algorithm must be executed in parallel to some degree.

Ironically, grid search algorithms is a truly embarrassingly parallel algorithm. Still, even on large supercomputers with hundreds of thousands of CPUs, the volume of the search space is just too large to use it.

4.2 Scalability of Evolutionary Algorithms

The evolutionary algorithms used for calibration basically use random search at each iteration, making increasingly educated guesses. Within each iteration, one can perform massively parallel computations with very little communication among processes. However, since there is data dependency between one iteration and the following, these algorithms cannot scale indefinitely. Nonetheless, the upper limit for scaling within an iteration usually depends on a free parameter.

Having access to a supercomputer, one could think to tweak this parameter and scale up the number processing units with impunity. In this way, he could keep the same number of iterations and find "better" results in the same amount of time, or use fewer iterations and find the old results in shorter time. Unfortunately, this is not the case.

As a first approximation, we can decompose calibration in two tasks. The first is locating the region of the search space containing the global minimum, and the second is exploring this region to locate its exact value. Increasing the number of guesses each iteration, which determines maximum scaling, increases the chances to escape from local minima, and move the exploration and refinement phase at later iterations. We can say that the convergence of the evolutionary algorithm slows down. Conversely, the fewer the guesses at each iteration, the sooner the local-search behaviour kicks in.

It is reasonable that a sweet spot for the number of guesses exists, providing acceptable solutions in the least amount of time. The optimal combination of algorithm, number of concurrent guesses and iterations must be determined by empirical studies, in lack of a general theory. These empirical studies may use optimization algorithms as well, instead of trying each possible combination, as done in grid search. Since the degree of concurrency is multiplicative under composition, optimization of calibration hyperparameters could easily exceed a large supercomputer capacity.

4.3 Scaling in Theory and in Practice

> In theory there is no difference between theory and practice - in practice there is
>
> —Yogi Berra

The remarks on scaling of the algorithms sketched in the previous section do not apply to their implementation. For example, they don't take into account finite data transfer bandwidths and latencies among CPUs. Also, they neglect the CPU cycles needed by the optimizer itself and consider only the load due to objective function evaluations. Therefore, the scaling of real calibration is a different matter.

A crucial difference from the ideal case, is that the objective function is not a total function. There are values of the parameters for which GEOtop crashes, immediately or at later times, or it does not converge, and the computation takes forever. This occurrence has an impact both on the implementation and scaling. On the one hand, the implementation must have some form of resiliency against objective function failure. On the other, it motivates speculative execution: it is convenient to evaluate the objetive function more times than needed, using all the available CPUs, because some of them will fail. The consequences of objective function failure will be investigated in later chapters.

In the general case of black-box optimization, the specific bottlenecks that an implementation might find depends on the characteristics of the objective function. Placing the objective function in one of the four quadrants of a plane where the axes are the boundedness (CPU, IO) and cost (cheap, expensive) is a good indication of what to expect.

For example, the execution of cheap CPU-bounded functions may need a very responsive scheduler, as you may receive too many requests on a distributed system or even waste too much time on forking and joining threads. An expensive IO-bound function will require almost certainly a distributed file system.

This kind of optmization problems are common and already cited examples are earth-system model calibration and hyperparameters optimization in machine learning models. The application of HPC to these problems is an active reseach topic and a vast, rapidly evolving field.

CHAPTER

FIVE

DERIVATIVE-FREE OPTIMIZATION

> My children, the only true technology is nature. All the other forms of manmade technology are perversions.
>
> —Ralph Bakshi, Wizards

5.1 Random Search

In this chapter, I would like to pinpoint in mathematical terms the ideas behind the numerical experiments with which this thesis is concerned. The following is far from a rigorous treatment, may seem pretentious or naive, and probably it is. Nonetheless, I think it's helpful to have a mental model of the computations that are going to be performed.

As more thoroughly explained in the previous chapters, we are concerned with the calibration of the GEOtop hydrological model. Calibration means to find the values of the input parameters that one has to set to obtain the best possible overlapping between the outputs of a simulation and the experimental data.

"Overlapping" is not a precise term, but for the moment let's suppose that we have a pure function, the cost function, which takes the value of the parameters as arguments, runs a simulation, and returns a real number representing the discrepancy with observations: higher values mean worst overlap, and 0 means perfection. We will consider only continuous parameters, and suppose that you can get arbitrarily small differences by evaluating the cost function with " close enough" arguments, which we can image as points of the parameter/search space.

Therefore, we can model our cost function as lower bounded, continuous function $f : D \subseteq \mathbb{R}^n \mapsto [0, +\infty)$, where the search space D is compact such that we know that there exist one or more global minima. We should consider an extension of $[0, +\infty)$ that includes failing computations, when the model crashes or fails to converge, something like bottom in Haskell. Let's forget about it though, and consider this information encoded in the domain D, within which our computation acts like a real mathematical function.

A black box optimizer is an iterator that at each step returns the approximate location of the minimum (referred to as recommendation or candidate), with increasing precision as the iterations go by. In this innocent statement lingers the assumption that it exists only one global minimum, assumption which for the moment we will ignore. In particular, we will consider only randomized algorithms which return a different sequence of points at each execution.

Therefore, we can define an optimizer as a random process $X_i : \Omega \mapsto D$ that gives better and better recommendations, that is $\forall \omega \in \Omega.\ f(X_i(\omega)) \leq f(X_{i-1}(\omega))$.

The first optimizer we consider is random search, in which we sample the search space uniformly at each step and recommend the best result to that point. Let $U_i : \Omega \mapsto D$ a sequence of iid random variables uniformly distributed on D (that is compact, hence has finite Lebesgue measure), then we can define the random search X_i as follows

$$\forall \omega \in \Omega. X_0(\omega) = U_0(\omega), X_i(\omega) = \begin{cases} U_i & \text{if } f(U_i(\omega)) \leq f(X_{i-1}(\omega)) \\ X_{i-1}(\omega) & \text{otherwise} \end{cases}.$$

In this way, the sequence of losses $Y_i = f(X_i)$ (which are random variables themselves) is point-wise monotonically decreasing, so it converges point-wise to a random variable $Y = \lim_{i \to \infty} Y_i$. It is easy to show that $Y = \min_{x \in D} f(x)$ almost everywhere.

5.2 Evolutionary Algorithms

Random search doesn't perform poorly at all in the long run, and, if we could wait forever and sample infinite points of the search space, it would be a good option. Therefore, when we ask another optimizer X_i' to "outperform random search" what we mean is that, on average, we want smaller losses Y_i' after a finite number of steps: $\mathbf{E}[Y_i'] \leq \mathbf{E}[Y_i]$.

To this purpose, other optimizers use a more refined strategy. At each step, they sample from random variables U_i' whose distribution is inferred from previous steps, and which usually converge to some a posteriori U'. In this way, these algorithms super-sample the region of the search space where, based on their assumptions, it is more probable to find a global minimum and accelerate the convergence of the best candidate X_i'. It is crucial to note that while they focus on a particular minimum (global or local), they subsample the rest of the search space.

In other words, if a heuristic algorithm converges, usually it does rapidly to a minimum and then sits there, with minimal chances to discover different minima. Therefore, these algorithms typically have a parameter that can be tweaked to explore more the search space but which slows convergence: exploration vs exploitation.

Evolutionary algorithm is a broad term applicable to a large collections of heuristic optimization algorithms inspired by biological evolution or other biological processes, as in Particle Swarm Optimization. An evolutionary algorithm starts by drawing a set of random points of the search space, the so-called individuals, with some a priori distribution, and then repeat the process using a modified distribution, which takes into account the losses of the individuals. Each iteration is called a generation, and the individuals of a generation are the population.

In general, the population size behave as the parameter described above, allowing more or less exploration of the search space. The main difference between an evolutionary algorithm and another is how random variables from which individuals of a generation are drawn depend on the ones of the previous generation. The crucial point is that individuals of the same generation are independent one another.

CHAPTER

SIX

TOOLS

> Code reuse is the Holy Grail of Software Engineering
>
> —Douglas Crockford

6.1 The Python Programming Language

Python is an interpreted, duck-typed language[1] with a terse syntax, allowing for fast-paced development. Python programs can easily integrate code written in other languages, such as C (the language of the reference implementation CPython), and for this reason, it is said to be a glue language. Moreover, it has a rich standard library and a full gamut of third-party libraries and tools. In 2020 Python has been the third language in the TIOBE index and its popularity has grown steadily in the past years, especially in machine learning and data science.

The usage of Python in scientific computing is nowadays a consolidated practice. Among its reasons, there is the existence of mature, fully-featured, and open-source libraries like Numpy, Matplotlib, and Pandas, respectively, for numerical computing, plotting, and manipulation of tabular data. Thanks to these libraries, Python became a popular alternative to MATLAB and R.

Indeed, Python is a right candidate when choosing a programming language in which writing general and extensible code, yet to be used by scientists without formal training in computer science. In that sense, Python can be viewed as a glue language also in terms of the developers' and users' levels of expertise. However, when it comes to the typical workloads found in numerical computing (the so-called number-crunching), Python shows the same shortcoming of MATLAB. Due to the dynamic nature of the language, repeated floating-point operations on contiguous memory or other kinds of tight loops, which can be very fast on modern CPU architectures, take far longer in Python than in other compiled languages. Although comparing different programming languages' performance is often unmeaning, a quick look at the computer language benchmarks game [BFG01] tells us that a pure Python implementation of these CPU-bounded algorithms should be expected to be between ten and one thousand times slower than the same in C. So how come that Python is widely adopted in numerical computing?

The answer is similar to the one for the case of MATLAB. In both cases, the dynamic programming language overcomes these shortcomings by wrapping and calling some more efficient code, usually written in Fortran, C or C++. It is possible to do so because the distribution of the

[1] Optional type annotations are available from Python 3.6. During execution, Python will ignore the type of objects anyway, and code defines behaviour using only the methods that objects implements. However, when annotations are available, types can be calculated and used for static analysis and debug purposes.

workload in number-crunching codes is often peaked: most of the time is spent on a small fraction of the code. Hence it is possible to optimize the whole program by rewriting a few routines in another language. In Python, optimized mathematical routines are available in the Numpy library, and it is often possible to obtain a sensible speedup by rewriting unoptimized code in terms of these functions acting on arrays and avoiding loops (a paradigm sometimes called array programming). In worst cases, a large part of the program needs to be rewritten, and Python serves just as a prototyping language.

This problem is typically referred to as the two-language problem, and a new programming language named Julia has been recently developed to solve this problem. There is a shared opinion that Julia will probably become the dominant language in scientific computing in the future. However, this will not happen too soon because of code and users inertia.

The reasons why the Python programming language has been chosen for the project reflect the general facts listed above: the need for a simple language known by the scientists who will work and use the code in the future, the capability of interfacing with the operating system and external processes natively, the availability of libraries for derivative-free optimization and distributed computing.

I will discuss the points above more thoroughly in the following sections.

Finally, almost all of the calibration time is spent executing the model and the overhead caused by a suboptimal optimizer[2], for example, due to the programming language used, is negligible. This statement, which has been discussed in the previous chapters, will be given a quantitative meaning in the next ones.

6.2 The Jupyter Ecosystem

The Jupyter Notebook is an open-source, interactive web application that allows writing executable documents, called notebooks, containing rich text, code and visualizations.

Jupyter Notebook is language-agnostic and can include code written in various languages, such as Python, R, Julia or C++. The code is executed on a language-specific kernel, an instance of an interpreter connected to Jupyter via ZeroMQ, an asynchronous messaging library.

Notebooks are stored as text files, and the Jupyter Notebook file format is defined via a JSON schema. For historical reasons, notebooks extension is `ipynb` (the Python kernel is called IPython). Notebooks are divided into cells, which can be of different types. Code cells can be evaluated and the value yielded by the last statement of a cell is captured by Jupyter Notebook, which by default then renders it as HTML, together with the standard output and error collected during execution. Usually, the output of a cell is stored within the notebook. Also, the order of execution of a notebook's code cells is arbitrary, but the execution count of a cell is stored within its metadata.

Since late 2018, the Jupyter Notebook has been integrated into JupyterLab, an extensible IDE-like interface combining the Jupyter Notebook with a terminal, a text editor and a file browser. However, this new interface is entirely backwards compatible.

It is crucial to notice that Jupyter Notebook and JupyterLab define both an execution model and a file format.

A notebook's execution is stateful: the same notebook can be executed twice by the same kernel and obtain different results because the kernel's internal state can change between

[2] Sorry for the wordplay.

executions. Indeed, for this reason, users often need to restart the kernel and re-evaluate the code cells. The hidden state problem has been criticized, and other implementations of reactive notebooks exist (such as Pluto for Julia).

Since the file format is a JSON dictionary, binary data must use a binary-to-text encoding, which introduces an overhead (of both memory and CPU cycles). Furthermore, notebook files poorly integrate with version control systems (another problem solved by Pluto).

Notwithstanding these and other problems, Jupyter has become the de-facto standard for interactive computing and visualization, and nowadays it is used by a vast community of scientific programmers, data scientists, and educators.

More recently, some initiatives within the broader context of reproducible research started targetting specifically at Jupyter [RBZ+19]. Various projects integrate and extend JupyterLab, for example, to validate notebooks, to parametrize their execution (Papermill), to retrieve data from them (Scrapbook), or produce complex, publication-quality documents (Jupyter Book).

6.2.1 Papermill and Scrapbook

These two lightweight Python libraries belong to the Nteract organization and have orthogonal purposes: Papermill allows to parametrize and execute notebooks, while Scrapbook allows to record data produced during execution.

Papermill can be used via the Python API or its command-line interface. In both cases, the user must provide the path to the input notebook, the path to the output notebook, and a dictionary of parameters. The paths can be on the local filesystem or remote ones; Papermill currently can handle HTTP, HTTPS and supports additional protocols for working with the major cloud storage providers. Also, it can read and write to the standard input and output, respectively. When using the CLI, the dictionary of parameters is specified using a YAML string, a YAML file path, or the value of single parameters. When using the API, Papermill can also use a Python dictionary.

Before executing the input notebook, Papermill looks for a cell with the `parameters` tag containing the default values. It adds a new cell just below (tagged with `injected-parameters`) overwriting the value of the variables contained in the parameters dictionary. Finally, it executes the notebook and saves a copy at the output path location. The output notebook also contains Papermill execution metadata, such as the injected parameters' value and each cell's execution time. As a notebook file is a JSON, one can easily retrieve these metadata afterwards.

The same can be obtained in other ways, for example, using NBclient. However, Papermill is more featured and integrates more easily in a data analysis pipeline. For example, it is currently employed at Netflix to schedule notebooks [SKU18].

A cell's output is usually rendered using HTML and embedded into the notebook file as a JSON string as sketched above. However, this is only a first approximation. The actual output is a JSON dictionary with a specific structure, and its content can have an arbitrary MIME type. Nonetheless, there is no easy way to serialize an object created during execution and then include it in the cell output using Jupyter Notebook only. However, the Scrapbook library does precisely that.

The Scrapbook library introduces a few names, quoting from the documentation page of the project scraps: serializable data values and visualizations such as strings, lists of objects, pan-

das dataframes, charts, images, or data references, notebook: a wrapped nbformat notebook object with extra methods for interacting with scraps, scrapbook: a collection of notebooks with an interface for asking questions of the collection, encoders: a registered translator of data to/from notebook storage formats. Notice that Scrapbook, which was initially part of Papermill, share with the previous the capability to work with remote file systems. Also, the Scrapbook library is easily extensible. Indeed, it is possible to register new encoders for serializing Python objects, possibly using high-performance formats, such as Apache Arrow.

The combined usage of Papermill and Scrapbook allows batch processing of notebooks, persistent storage of the results, and retrieving them, for example, for ensemble analysis. Of course, one could obtain the same in other ways, such as using Python scripts and a database. However, this way, it is possible to use the same tools for prototype and production code.

6.2.2 Jupyter Book

Jupyter Book is an open-source project which leverages the Sphinx documentation system to build publication-quality documents from Jupyter notebooks and Markdown sources. It can execute and cache Jupyter notebooks, and use its outputs (including interactive widgets). Furthermore, it supports a Markdown flavour called Markedly Structured Text, providing margin notes, blocks, panels, dropdowns, etc. This document is built using Jupyter Book.

6.3 GEOtoPy

GEOtoPy is a small Python package without external dependencies developed for this project. It contains a single module exporting a single base class `GEOtoPy.GEOtop`, which must be derived to have a functioning GEOtop wrapper.

GEOtop allows running different types of simulations using a flexible configuration file. The model's inputs and outputs can be very diverse, and the same is true for the workflow in which one runs model. If one wants one tool able to encompass all the different scenarios, he has two options. The first is writing a very complex wrapper introducing an abstraction layer capable of handling all particular cases. The second is writing a barebone wrapper, lacking part of the implementation, and let the user adapting it to his use case. GEOtoPy chooses this second option.

Indeed, the purpose of GEOtoPy was more about documenting and allowing code reuse than being a full-fledged wrapper.

The central assumption of GEOtoPy is that the user workflow consists of the following steps:

1. check minimal preconditions on the inputs,
2. preprocess the inputs,
3. run the model,
4. postprocess the outputs.

The `GEOtoPy.GEOtop` constructor takes care of the first step. The other three are encapsulated in the `run_in` method of the object.

Let me recall that the GEOtop executable takes one argument, which is the path to a directory containing a `geotop.inpts` file. The `GEOtoPy.GEOtop` class constructor takes only one posi-

tional argument: the path mentioned above, which can be a string or a PathLike object. It then checks that this path points to a readable directory and contains a readable `geotop.inpts` file. It also checks that a `geotop` executable exists. If all the preconditions above stand, it reads and parses the `geotop.inpts` file. Otherwise, it throws an error.

The constructor does not ensure that the `geotop.inpts` file is a valid configuration file. Indeed, in the case of malformed `geotop.inpts`, it just warns the user. A valid line in the configuration file can be either a comment, identified by the regex `\s*!.*\n|\s+`, or a setting, identified by the Python regex `\s*(?P<keyword>[A-Z]\w*)\s*=\s*(?P<value>.*)(?:\n|\Z)`. Values associated with given keywords can be read from and print to strings thanks to a JSON dictionary containing the type associated with each keyword.

Correctly parsed settings are stored in a Python dictionary within the object. The constructor can also keep the whole content of the inputs directory in memory, archived into tar, using `BytesIO`.

The `run_in` method executes the model within a working directory `working_dir` with additionally provided arguments. Its purpose is to execute in sequence the preprocessing, running and postprocessing steps. It has more or less the following implementation

```
import subprocess

def run_in(self, working_dir, *args, **kwargs):
    self.preprocess(working_dir, *args, **kwargs)
    subprocess.run([self.exe, working_dir], **self.run_args)
    output = self.postprocess(working_dir)
    return output
```

It is worth noticing a few design choices.

First, the data flow via IO. The `postprocess` method has no arguments other than the working directory. The choice of the `postprocess` signature follows from the assumption that the derived class implements the wrapper for a specific type of simulation, with a particular shape of the inputs and outputs. However, the `preprocess` method can take additional arguments to change the values of (some of) the inputs and run different simulations.

Second, since we want to run the model multiple times in a concurrent fashion, it is fundamental that different runs do not interfere with one another. If `run_in` does not change the global state but change the internal one of the object to which it belongs, it is possible to run the method on multiple copies of the same object simultaneously without data races. If we want to avoid duplicates, the `run_in` method must be a pure function: it must not have side effects. However, strictly speaking, both scenarios are impossible since GEOtop works on files, and the running step is guaranteed to do IO.

Nonetheless, we can avoid data races by requiring that `preprocess` and `postprocess` satisfy some conditions. Unfortunately, there is no way of expressing this behaviour in Python. Hence it is the responsibility of the user to implement these methods such that they fulfil them. The methods shall not change the state of the object, shall not write on the inputs directory. Indeed, it is also essential to assure that the inputs do not change from one run to another. The `run_in` method checks that `working_dir` points to a different location from the inputs directory.

By combining the previous gimmicks, we can run in parallel the GEOtop model from Python multiple times, using only one instance of the `GEOtoPy.GEOtop` class.

A minimal functioning implementation is

```python
from geotopy import GEOtop

class Model(GEOtop):

    def preprocess(self, working_dir, *args, **kwargs):
        self.clone_into(working_dir)

    def postprocess(self, working_dir):
        return None
```

which populates the working directory with the input files, and always returns `None`.

Indeed, the `GEOtoPy.GEOtop` class also provides some helper methods to implement the `preprocess` and `postprocess` ones, like `clone_into`.

6.4 Nevergrad

The panorama of existing Python libraries for derivative-free optimization is varied, as different libraries account for different needs. However, the vast majority of these libraries are designed for hyperparameters optimization of machine learning models. Hyperparameters optimization in machine learning is a vast topic, which is difficult to summarize in a few words. Still, the main idea is to find the optimal values of the parameters that control the learning process. For different reasons, derivative-based algorithms, such as gradient descent or BFGS, are usually not suited for searching the optimal values of model hyperparameters (for example, because there is a mixture of continuous and discrete parameters). An interesting class of algorithms is the early stopping one, especially when model training is computationally expensive. Indeed, the application of early stopping algorithms to the calibration of earth-system and environmental models might be a good research topic.

In general, derivative-free optimization libraries consist of two pieces:

1. one to model the search space, and
2. one to select the algorithm and perform the optimization.

Also, these libraries typically assume that the interface with the objective function is a callable object.

Nevergrad is a Python library for derivative-free optimization not explicitly targeted at hyperparameter optimization and focusing on evolutionary algorithms [RT18]. It can handle continuous and discrete parameters, and Python containers, such as tuples, lists (arrays) and dictionaries. It has a wide range of preconfigured optimization algorithms, and it offers both a high level `minimize` function, and a lower level ask-tell interface. Notice, that the `minimize` function is able to evaluate the objective function in parallel using the `concurrent.futures.Executor` interface. The ask-tell interface [CHP+13] is an algorithm-agnostic, object oriented programming interface for implementing the optimization loop, and will be discussed in the next chapter.

The Nevergrad library implements several evolutionary algorithms, such as Particle Swarm Optimization (PSO), Covariance Matrix Adaptation - Evolutionary Strategy (CMAES) [HMullerK03]. It also contains one-shot algorithms, i.e. algorithms where the points of the search space which will be sampled are known from the beginning. Finally, it has two meta-algorithms,

Shiva and NGO, which select an algorithm among the available ones based on the available information using empirical rules.

The algorithms implemented in Nevergrad follow the same philosophy as CMA-ES: the choice of the hyperparameters of the optimizer should be part of the algorithm's design (although it is possible to tweak and configure the optimizers if needed). The only parameters that the user must specify are the `budget` and `num_workers`. The first is, simplifying a bit, the number of allowed calls to `optimizer.ask()`.

The first (the budget), is significant for some one-shot algorithms, where the optimizer must generate a low-discrepancy sequence of a given length apriori. The second, `num_workers`, is the number of objective function calls that can be evaluated in parallel, i.e. the number of CPUs. In evolutionary algorithms, the latter maps naturally to the number of individuals in a generation, the population size. In the next chapter, we will see how and why these two are involved in failing objective function evaluations.

6.5 High-Performance Computing in Python using Dask

The previous sections leave open the question of whether it is possible to do High-Performance Computing using Python. The answer to this question is related to performing parallel computations on distributed systems using this language. It turns out that Python is indeed a useful tool for this purpose, and it is capable of scaling on large supercomputers: it was able to scale up to 921 nodes on Summit [Col20], still leaving room for improvement (M. Coletti, personal communication, 28 January, 2021).

> As you've no doubt observed, there hasn't been activity of late, but it does reference at least one other group that has modified the heartbeat implementation that could be used to overcome scaling problems we encountered on Summit. That is, I was able to successfully run dask on Summit up to 921 nodes, with six dask workers per node, but couldn't scale beyond that; but there was a comment in that github thread about tweaking the heartbeats to possibly overcome that. In any case, since 921 nodes is almost a quarter of the machine, that may actually be enough for most scientific tasks on that platform. However, I'm confident that, again, with some configuration tweaking that dask could be pushed to support more nodes.

Still, before moving to distributed systems, it is interesting to examine the topic of parallel computing in Python on a single shared-memory system.

6.5.1 Parallel computing in Python

The reference implementation of Python, the CPython interpreter, compiles the Python code into an intermediate representation called bytecode, which runs on a virtual machine. Also, CPython uses reference counting for garbage collection. This means that there is a counter for each object created during the execution of a Python program: when a variable is bound to the object, the counter of the latter is increased, when a reference is deleted, the counter is decreased. Once the counter reaches zero, the object is deallocated, or, in C++ parlance, the destructor is called. In multi-threaded code, this form of garbage collection requires some mechanism to avoid data races. CPython opted for a global lock on the interpreter, hence called the Global Interpreter Lock or the GIL. External dependencies might or might

not acquire the GIL, but when they do the interpreter cannot move to the next bytecode instruction until the GIL is released, even if the dependency spawns new threads. The reason is that Python includes several C dependencies, not all of them thread-safe.

The Python standard library provides the threading module to work with threads. However, one must always consider that the GIL prevents the interpreter to be executed by more than one thread at a time. In other terms, code written in pure Python will be executed as if there is only one CPU core, even if multiple ones are available. Of course, if the Python code running on a thread reach a C extension which releases the GIL or spawn its own threads, then the interpreter can move to the next instruction. In this way, we can have a speedup with multi-threaded code. This approach may be beneficial, for example, with Numpy routines or IO code.

The Python library also provides the multiprocessing package to do multi-processing (the clue is in the name). This module side-steps the GIL by launching several processes, each one being a full-fledged Python interpreter. The objects defined in the main process will be serialized and sent to the others. The serialization happens via the Pickle module from the standard library. However, Pickle has some limitations. For example, it is not able to serialize lambdas. If one wants to use multiprocessing he has to build his code around these limitations. Finally, multiprocessing has larger overhead than multithreading due to serialization and, on a much smaller degree and depending on the operating system, to system calls. Notice that the multiprocessing contains also the `shared_memory` module to provide direct access to shared memory across processes.

The standard library also provides some abstractions to work with these low-level tools. Among them, there is the `concurrent.futures` module. Promises and futures are constructs used in some concurrency models. They represent values that will eventually become available and are usually the result of a remote computation. They can be viewed as queues of size one: producers make promises while consumers wait for futures. These constructs are supported by many languages. The `concurrent.futures` module defines the `Executor` abstract class that provides methods to execute calls asynchronously, such as `submit` and `map`. The `submit(fn, *args, **kwargs)` method returns immediately a `Future` object. Afterwards, it is possible to call the `result` method on a `Future` object, which will block and return the value of `fn(*args, **kwargs)` as soon as it has been evaluated. The concrete classes `ThreadPoolExecutor` and `ProcessPoolExecutor`, derived from `Executor`, use a pool of threads and processes respectively to evaluate the result (again, the clue is in the name).

Finally, it should be noted that Python supports non-preemptive threading natively with `asyncio`, and, while `asyncio.Future` objects are awaitable, `concurrent.futures.Future` ones are not.

6.5.2 Distributed Computing with Dask

Dask is a library for parallel computing in Python which consists of two components: a scheduler and a collection of data structures with an interface familiar to Numpy and Pandas users. Indeed, it is possible to enable parallelism simply using Dask arrays and dataframes as a drop-in replacement for their Numpy and Pandas equivalent. Also, Dask can do out-of-core computations and exploit distributed systems. Since the application we are concerned with is CPU bounded, I will focus on the second.

In Dask, operations on the above-mentioned data structures are divided into smaller tasks, and the dependency relations among the tasks is encoded into a task graph. Afterwards, the scheduler use the task graph to distribute the work among different processing units.

Moreover, the intermediate representation of a computation as a graph allows for some optimizations.

Therefore, Dask fits naturally the domain of computations that can be efficiently expressed using array programming and involving large amounts of data. However, it also provides interfaces to interact directly with the scheduler, adapting to more general cases. One of them is the futures interface, which extends `concurrent.future` from the standard library. I briefly described the futures abstraction in the previous section and some further details will be discussed in the next chapter on implementation.

Dask has different scheduler implementations, and it's responsability of the user to choose and setup the right one. All the implementations share the concept of worker: a piece of software that takes a task, performs some computations and returns the results. Tasks are scheduled for execution on a pool of workers. Different workers might be executed concurrently on different CPU cores, thus achieving parallelism (or serially for debug purposes).

Currently, there are four scheduler implementations available.

1. Local Threads. Tasks are executed on separate threads, and the implementation internally uses `multiprocessing.pool.ThreadPool`. Given the above discussion on multi-threading in Python, this scheduler must be chosen only when the tasks release the GIL, such as for Numpy or Pandas. Also, the overhead per task is small.
2. Local Processes. Tasks are executed on separate processes, and the implementation internally uses `multiprocessing.Pool`. There is a large communication overhead compared to threads, but pure Python tasks can be executed concurrently.
3. Single thread. Tasks are executed in the local thread, for debug purposes.
4. Dask Distributed. Tasks are executed by workers that runs as server applications.

Dask Distributed requires a runtime both for the scheduler and the workers, both running as separate daemon processes, and called `dask-scheduler` and `dask-worker`. It bundles an API to connect to `dask-scheduler` from Python via the `dask.distributed` module. However, it is possible to use this implementation on a single machine in the same fashion of the implementations, avoiding the setup and letting Dask Distributed manage the runtime. Using Dask Distributed on a single machine offers more advanced features than using local processes, such as profiling. For this reason, it is the recommended way of running Dask in most of the situations where using local threads is not.

In a Dask Distributed setup, there can be N `dask-worker` processes, each internally running a pool of T threads using `multiprocessing.pool.ThreadPool`. This means that at most $N \times T$ tasks can be executed in parallel, if there are enough CPU cores are available. Choosing the right combination of processes and threads is part of performance tuning, and it is specific to the kind of workload considered.

In a Dask cluster, there is one `dask-scheduler` and (possibly) multiple `dask-worker` processes. These processes do not need to be executed on the same machine, however, they must be on the same network, since the need to communicate one another. Communications among processes happens via TCP/IP, but UDP protocol is available and there is experimental support for Nvidia UCX. When starting the cluster the user must provide to each worker the address of the scheduler. The scheduler takes care of collecting and distributing the addresses of the workers on the network, so that point to point communications are possible without passing throught the scheduler.

CHAPTER

SEVEN

IMPLEMENTATION

> Talk is cheap, show me the code.
>
> —Linus Torvalds

7.1 Model and Objective Function

As discussed in the previous chapter, in order to have a functioning interface to the GEOtop model using GEOtoPy it is necessary to specify how to do preprocessing of the data (basically meteorological forcings and input parameters) and postprocessing of simulation results. The following is the implementation used for simulations with uniform soil parameters.

```python
import json
import importlib.resources as resources
from geotopy import GEOtop
from mhpc_project.utils import postprocess_full

class UniformSoilModel(GEOtop):
    with resources.open_text('mhpc_project', 'uniform_defaults.json') as file:
        default_settings = json.load(file)

    def preprocess(self, working_dir, *args, **kwargs):
        settings = self.settings.copy()
        settings.update(self.default_settings)
        settings.update(args[0])
        self.clone_into(working_dir)
        self.patch_inpts_file(working_dir, settings)

    def postprocess(self, working_dir):
        settings = self.read_settings(working_dir / 'geotop.inpts')
        return postprocess_full(settings, working_dir)
```

In this case, preprocessing is trivial and consist of copying the files to the proper directory and patching the `geotop.inpts` with the new value of the parameters. Notice that the `self.settings` dictionary of settings is copied to avoid nefarious side effects. Preprocessing also takes care of appling some defaults settings, which are assumed in the postprocessing phase.

The postprocess is just a wrapper around a utility function. This is because the same postprocessing is done also in the variable soil model, discussed below. In this way one can avoid code duplication without introducing a intermediate class in the inheritance scheme.

Once having a functioning model, the objective function can be easily implemented.

```
def objective(model, candidate, observations):
    try:
        with TemporaryDirectory() as tmpdir:
            predictions = model.run_in(tmpdir, *candidate.args, **candidate.kwargs)
    except (CalledProcessError, TimeoutExpired):
        predictions = None
    return compare(predictions, observations)
```

The `compare` function should return the loss value. A good choice is to use the Kling-Gupta or Nash-Sutcliffe efficiencies [GKYM09][NS70], which are well-suited for hydrological models, notice however that higher number of these correspond to better overlap of simulation and observations.

Using `TemporaryDirectory` from the tempfile module allows running the model in `tmpfs` (i.e. in RAM) and automatic deletion of files on exit (also in case of exceptions and canceled Dask task). The `CalledProcessError` and `TimeoutExpired` exceptions from the subprocess module must be caught since they represent routine GEOtop failures. Other exceptions will be propagated since they signal abnormal behaviours.

7.2 A Closer Look at the Optimization Loop

A wide class of algorithms can be implemented using the unique interface described in [CHP+13], and the ones sketched in the chapter *Derivative-free Optimization* belong to them. Let's say we want to minimize the objective function `f`, the optimization loop will look like the following

```
while not optimizer.stop():
    x = optimizer.ask()
    y = f(x)
    optimizer.tell(x, y)
```

The pseudocode is rather self explicative: the `optimizer.stop` method must implement some stopping criterion, the `optimizer.ask` method suggests a new point where to evaluate the objective function, and `optimizer.tell` communicates the result to the optimizer. This design allows decoupling the optimizer from the objective function.

Since the state of the `optimizer` object contains the information about the history of evaluations, different criteria are possible, such as asking a decrease of the objective above some threshold. The simplest is to use an internal counter to allow a maximum number of objective evaluations, the budget in the Nevergrad parlance. It is crucial to notice that the number of `optimizer.ask` calls can differ from the one of `optimizer.tell` calls.

As it is, the loop is fully serial. We need an optimization algorithm capable of suggesting several points at ones to make it parallel. The requirement is non-trivial, for example Bayesian optimization does not fulfill it. Evolutionary algorithms generally satisfy the requirement, since individuals of the same generation are independent one another.

A parallelizable loop using generation count as stopping criterion will look like the following.

```
for _ in range(num_generations):
    for i in range(popsize):
        x[i] = optimizer.ask()
```

(continues on next page)

(continued from previous page)

```
    for i in range(popsize):
        y[i] = f(x[i])

    for i in range(popsize):
        optimizer.tell(x[i], y[i])
```

In theory, `optimizer.ask` could be read-only, and the first loop could be executed in parallel. In practice, it must change the state of the internal pseudo-random generator. Since `optimizer.tell` changes the state of the object (with exception of random search), concurrent execution of the third loop is guaranteed to cause race conditions unless `optimizer.tell` uses some mutex. However, both `optimizer.ask` and `optimizer.tell` are not supposed to be expensive to call[1], hence their loops can be executed in serial fashion without performance degradation.

When the objective function evaluation is time-consuming, as it is the case for GEOtop, most of the time is spent in the second loop. The objective loop is embarrassingly parallel and can be executed concurrently for example using futures. Let's suppose to have a Dask Client `client`, the loop becomes.

```
for _ in range(num_generations):
    for i in range(popsize):
        x[i] = optimizer.ask()

    for i in range(popsize):
        futures[i] = client.submit(f, x[i])

    for i in range(popsize):
        y[i] = futures[i].result()

    for i in range(popsize):
        optimizer.tell(x[i], y[i])
```

The first two loops can be fused, as well as the second two. Notice that Dask assumes that `f` is a pure function (otherwise, the whole computation would not make sense anyway).

The previous design however has a serious flaw: the third loop is executed synchronously (there is no event loop) and `future.result()` is blocking. Therefore, the interpreter will wait the first future, then the second, and so go on. However, since the execution time of `f` is random (and varies in a large interval of values), some results may be ready much before their turn. Fortunately Dask Distributed as the `as_completed` class, which iterates the futures as soon as they are done.

However, since we lost the information about the order of the results, we need to use a small wrapper that keeps track of the argument and the loss.

```
for _ in range(num_generations):
    for i in range(popsize):
        x[i] = optimizer.ask()

    for i in range(popsize):
```

(continues on next page)

[1] This is true within a generation. From one generation and the next, in case of large population size, the optimizer can perform expensive computations.

(continued from previous page)

```
        futures[i] = client.submit(lambda x: (x, f(x)), x[i])

    to_tell = []
    for future in as_completed(futures):
        to_tell.append(future.result())

    for x, y in to_tell:
        optimizer.tell(x, y)
```

Such optimization loop still does not take into account objective function failures. In our implementation failing computations return nan. We can check that the result is valid using the `isfinite` function from Numpy. It is possible to elegantly solve the problem by submitting other computations to `as_completed`. Indeed, it has two methods `as_completed.add` and `as_completed.update` which allow adding one or more futures to the queue respectively, and a `as_completed.count()` method which counts how many futures are still in the queue.

```
completed_queue = as_completed(futures)
to_tell = []
for future in completed_queue:
    x, y = future.result()
    if isfinite(y):
        to_tell.append(future.result())
    if len(to_tell) + completed_queue.count() < popsize:
        new_x = optimizer.ask()
        new_future = client.submit(lambda x: (x, f(x)), new_x)
        completed_queue.add(new_future)
```

The previous code works since to exit the loop `completed_queue` must be empty. If `completed_queue` is empty, then bottom of the loop has been reached without a insertion of a new future, that is `len(to_tell) + completed_queue.count() < popsize` must have been false. But `completed_queue.count()` is equal to zero, hence `len(to_tell) >= popsize`. However, the number of failures equals the number of insertions, hence `len(to_tell) == popsize`.

Let's consider what happen when we reach the end of a generation. There are `popsize - 1` elements in `to_tell`, and if the objective fails a single new future is added to `completed_queue`: all CPUs except one will wait in idle. A better idea is to speculatively execute more objective functions, so to increase the chances that at least one of them does not fail.

```
completed_queue = as_completed(futures)
to_tell = []
for future in completed_queue:
    x, y = future.result()
    if isfinite(y):
        to_tell.append(future.result())
    if len(to_tell) + completed_queue.count() < popsize:
        for _ in range(num_new_futures):
            new_x = optimizer.ask()
            new_future = client.submit(lambda x: (x, f(x)), new_x)
            completed_queue.add(new_future)
```

In this way, there will be a buffer of approximately `num_new_futures` extra futures. Indeed, the upper bound for the size of the queue is `popsize - len(to_tell) + num_new_futures - 1`. However, with this modification there is no guarantee that at the end of a calculation we have `popsize` elements in `to_tell`. Usually this is not a problem, since the optimizer will throw away

the worst individuals (or include them in computations, in a non-elitist fashion). However, we can enforce the old behaviour using a `break` statement and `completed_queue.clear()`. When the latter is called, the futures still in the queue are garbage collected, and Dask cancels the corresponding computations.

```
completed_queue = as_completed(futures)
to_tell = []
for future in completed_queue:
    x, y = future.result()
    if isfinite(y):
        to_tell.append(future.result())
    if len(to_tell) >= popsize:
        break
    else:
        if len(to_tell) + completed_queue.count() < popsize:
            for _ in range(num_new_futures):
                new_x = optimizer.ask()
                new_future = client.submit(lambda x: (x, f(x)), new_x)
                completed_queue.add(new_future)
completed_queue.clear()
```

However, this introduces a bias: futures that terminates earlier have more chance to be reported to the optimizer, whatever their loss is. This new behaviour is located near the end of a generation, when the computation start to consume the buffer, and it is more evident the larger `num_new_futures`.

Let's suppose that we are able to guess how many computations will be successful. In this case, we could add to the queue only the futures that will be consumed. If the fraction of valid futures in the queue is `r` then we will need to add new futures to the queue only if `len(to_tell) + r * completed_queue.count() < popsize`. Not only, we can now get rid of the free parameter `num_new_futures`. Indeed, if `r` can be expected to stay constant within a generation, a reasonable heuristic is to add each time `(popsize - len(to_tell) - r * completed_queue.count()) / r` new futures.

The actual optimization loop used implements this strategy, estimating `r` as the weighted average success rate using the following function.

```
# log is a list of triples [(individual, loss, execution time)]
def average_success_rate(log, alpha):
    if log:
        successes = [1 if np.isfinite(l) else 0 for (_, l, _) in reversed(log)]
        weights = [exp(-alpha * n) for n in range(len(log))]
        return sum(w * x for w, x in zip(weights, successes)) / sum(weights)
    else:
        return 1.0
```

In this way, the success rate looses memory of older evaluations. The timescale parameter `alpha` is chosen as `1 / popsize`.

However, it turns out that this strategy fails when the number of missing futures is small. For this reason the actual optimization loop implements also an `overshoot` parameter that allows to submit more new futures than estimated (but introducing again a bias related to execution times). Also, the actual implementation prefetches futures in batches, and pre-scatters the data across the Dask cluster.

CHAPTER

EIGHT

EXAMPLE GEOTOP CALIBRATION REPORT

What follows is an example of GEOtop calibration report.

In this report the GEOtop model has been run in 1D mode solving the water and energy balance for one site DOMEF 2000 located in a mountain grassland in the Italian Alps in South Tyrol. The model was run using in input meteorological parameters for a period of about 4 years, with about 35 months of warm up and then calibrated with respect of soil moisture observations for the last 18 months. We considered here 16 layers up to 1m depth. The chosen cost function is based on the Kling-Gupta efficiency. The model was optimized for 29 parameters, which control soil, surface and vegetation properties. The chosen optimization algorithm was CMA-ES. In this case 256 CPUs were used.

The reports show the decrease of the cost function as function of the generations and the comparison of the simulated time series with the observations after the calibration, at different time scales, aggregated hourly, daily and monthly.

```
# Defaults
model_path = '../data/testbed/inputs'
timeout = 120
observations_path = '../data/testbed/observations/obs.csv'
parameters_path = '../data/parameters/testbed.csv'
default_parameters = {}
algorithm = 'Random'
popsize = 2
num_generations = 2
scheduler_file = None
num_cpus = 2
num_workers = 2
performance_report_filename='report.html'
```

```
# Parameters
model_path = "/scratch/scampane/MHPC-project/data/DOMES/inputs"
timeout = 150
observations_path = "/scratch/scampane/MHPC-project/data/DOMES/observations/obs.csv"
parameters_path = "/scratch/scampane/MHPC-project/data/parameters/all.csv"
algorithm = "NGO"
popsize = 128
num_generations = 32
scheduler_file = "/scratch/scampane/MHPC-project/scheduler_files/scheduler-Q61.json"
num_cpus = 256
num_workers = 32
performance_report_filename = "/scratch/scampane/MHPC-project/runs/short/DOMES-NGO-4096-
↪256-DeW-performance-report.html"
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scrapbook as sb
import dask.config
from dask.distributed import Client, performance_report

from mhpc_project.utils import date_parser, kge_cmp, calibrate, delta_mim
from mhpc_project.parameters import UniformSoilParameters as Parameters
from mhpc_project.models import UniformSoilModel as Model
import mhpc_project.plots as plots
```

```
# Store dask config
sb.glue('dask_config', dask.config.config)
```

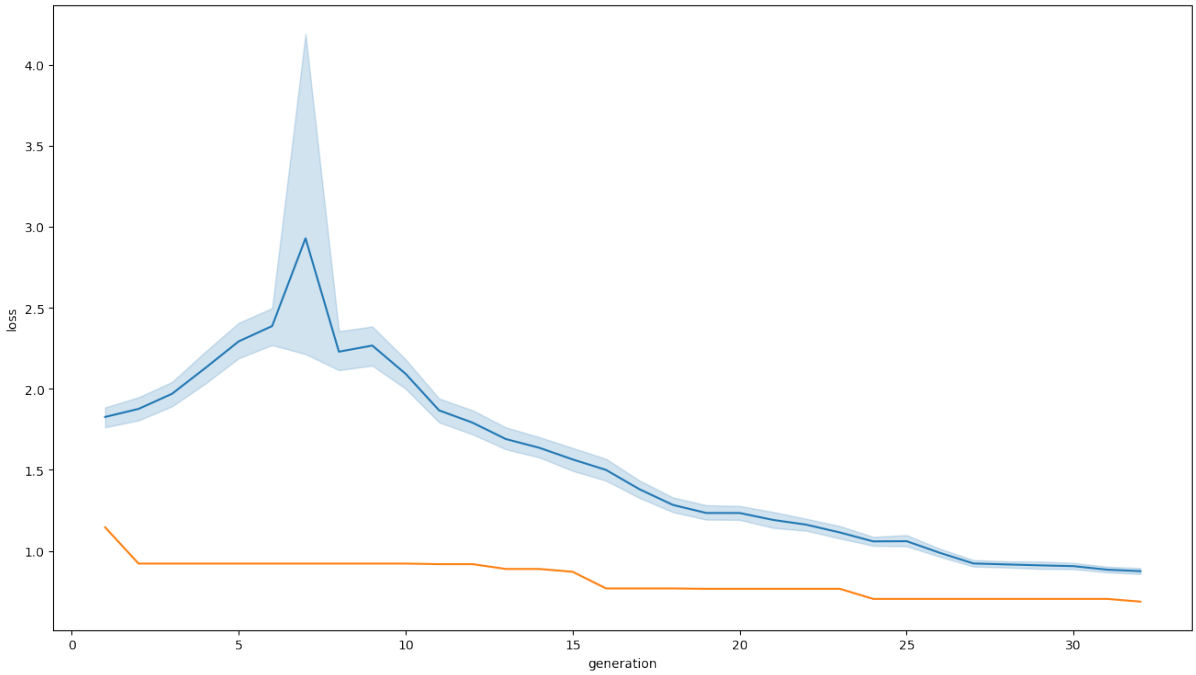
```
parameters = Parameters(parameters_path, default_parameters)
model = Model(model_path, store=False, timeout=timeout)
observations = pd.read_csv(observations_path,
                           parse_dates=[0],
                           date_parser=date_parser,
                           index_col=0)
```

```
if scheduler_file:
    client = Client(scheduler_file=scheduler_file)
else:
    client = Client()
client.wait_for_workers(n_workers=num_workers, timeout=240)
```

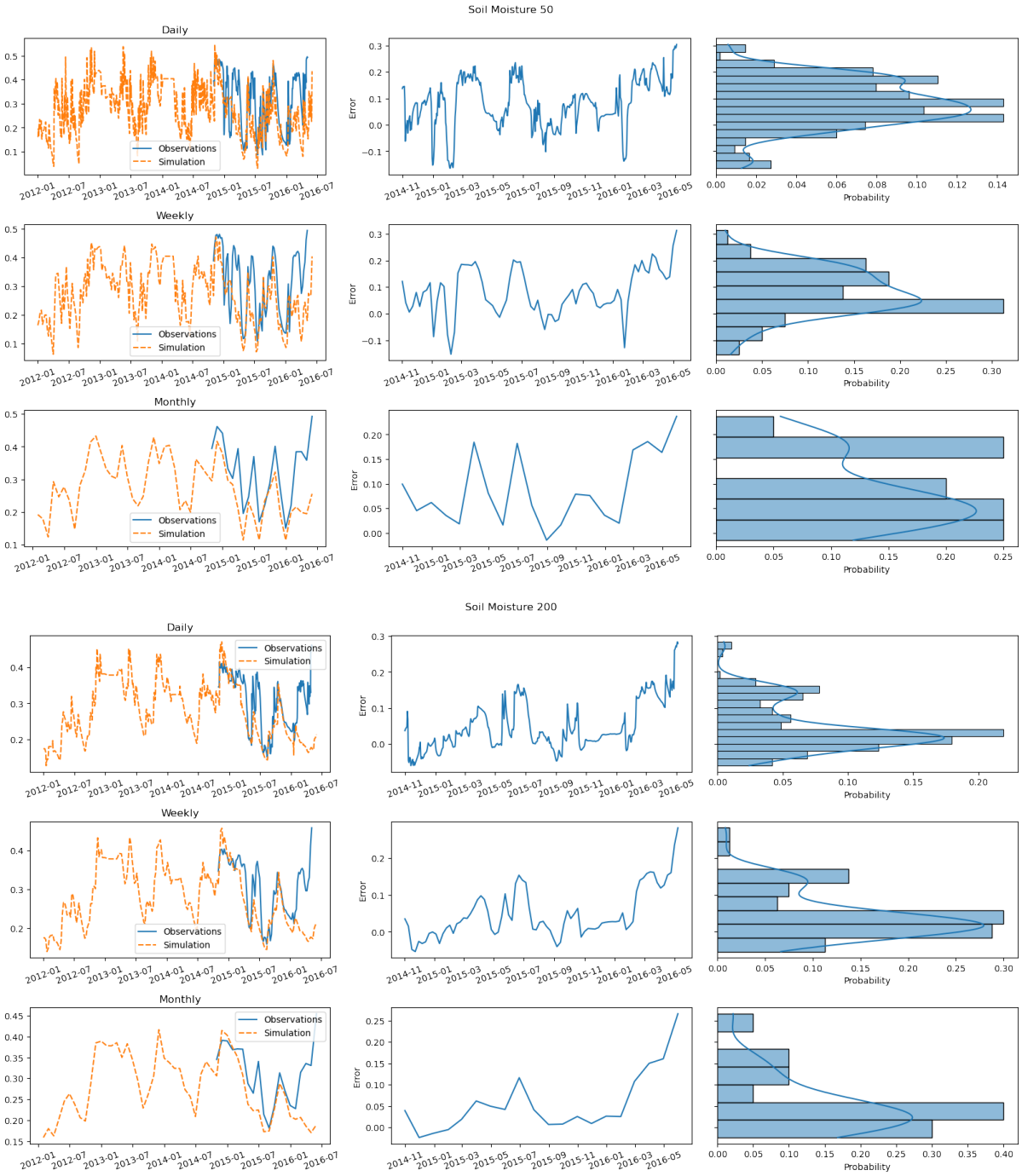
```
with performance_report(filename=performance_report_filename):
    recommendation, predictions, log = calibrate(model,
                                                 parameters,
                                                 observations,
                                                 algorithm,
                                                 popsize,
                                                 num_generations,
                                                 client,
                                                 num_cpus)

sb.glue('loss', kge_cmp(predictions, observations))
sb.glue('log', [((x.generation, x.args[0]), l, t) for (x, l, t) in log])
```

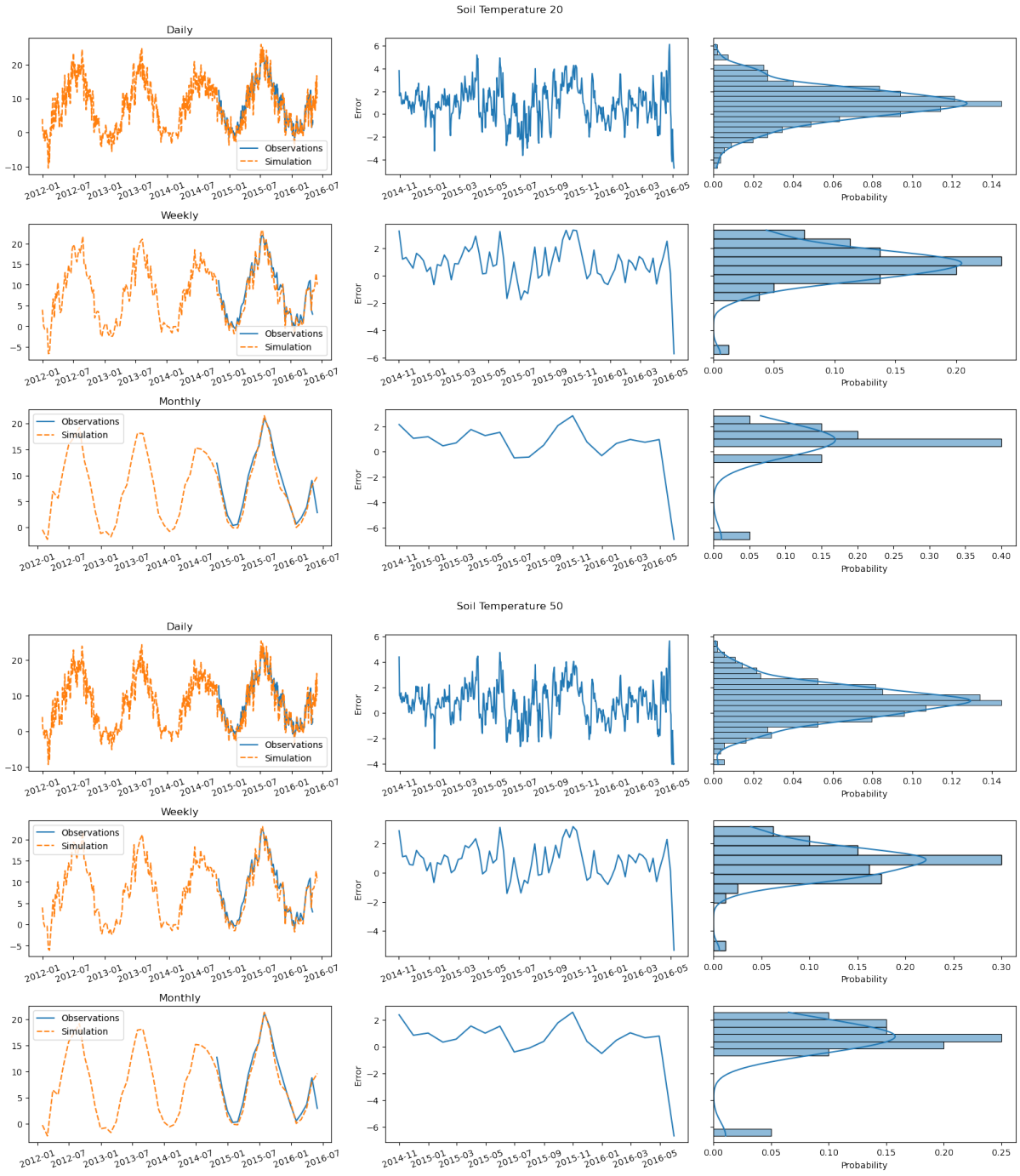
```
(64_w,128)-aCMA-ES (mu_w=34.2,w_1=6%) in dimension 29 (seed=nan, Thu Feb 18 09:36:33 2021)
```

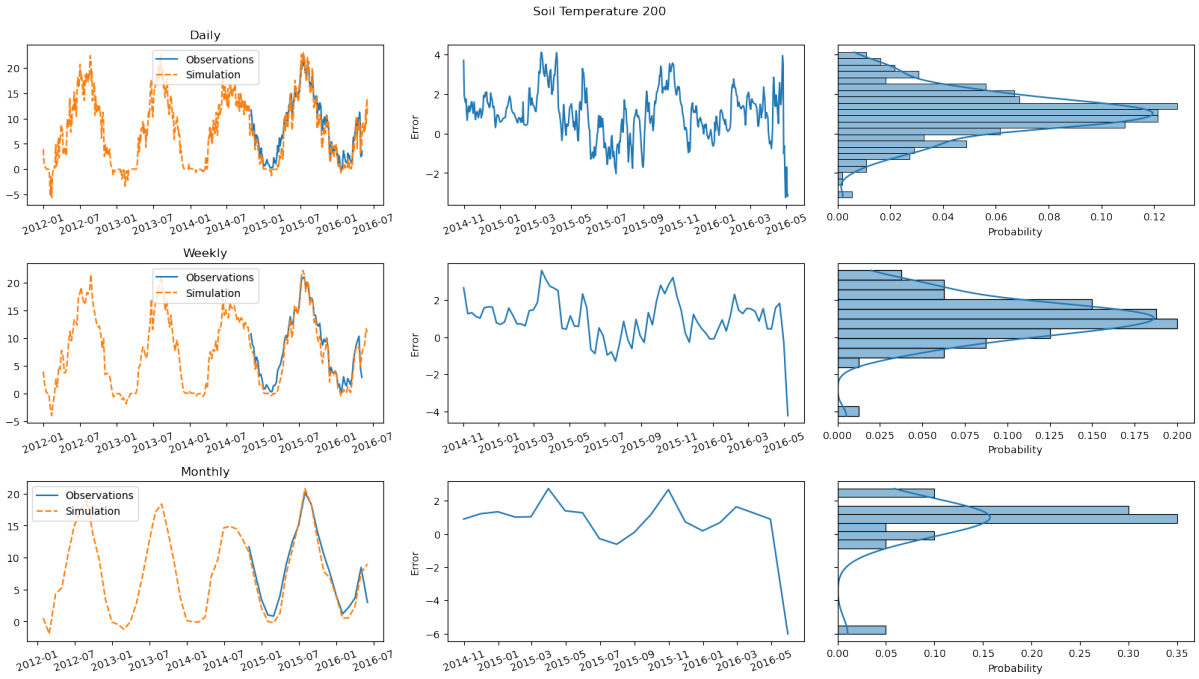
```
plot = plots.convergence([(x.generation, l) for x, l, _ in log if np.isfinite(l)])
plt.close(plot)
sb.glue('convergence_plot', plot, 'display')
```




```
for name, plot in plots.comparisons(predictions, observations).items():
    plt.close(plot)
    sb.glue(name + '_plot', plot, 'display')
```


Soil Moisture 50
Daily
Weekly
Monthly
Observations
Simulation
Error
Probability
Soil Moisture 200
Daily
Weekly
Monthly
Observations
Simulation
Error
Probability


Soil Temperature 20
Daily
Weekly
Monthly
Observations
Simulation
Error
Probability
Soil Temperature 50
Daily
Weekly
Monthly
Observations
Simulation
Error
Probability




```
report = delta_mim(parameters, [(x.args[0], l) for x, l, t in log if np.isfinite(l)])
report['best'] = parameters.from_instrumentation(recommendation)
sb.glue('report', report, 'pandas')

report.sort_values('delta', key=np.abs, ascending=False)
```

	delta	delta_conf	S1	S1_conf	\
FieldCapacity	0.196860	0.016790	0.083046	0.231401	
ThetaRes	0.180005	0.014483	0.104353	0.282825	
ThetaSat	0.168758	0.015870	0.097108	0.266497	
DecayCoeffCanopy	0.167036	0.012630	0.087294	0.237451	
NormalHydrConductivity	0.163122	0.018452	0.043401	0.147470	
CanopyFraction	0.162160	0.017316	0.054883	0.169293	
MinStomatalRes	0.152129	0.013413	0.043169	0.110797	
AlphaVanGenuchten	0.137440	0.014014	0.056531	0.136567	
NVanGenuchten	0.136678	0.017649	0.039043	0.125220	
LSAI	0.127544	0.017510	0.039181	0.119751	
SoilEmissiv	0.125561	0.017766	0.041374	0.119356	
VegSnowBurying	0.119362	0.021109	0.026987	0.086476	
ThermalCapacitySoilSolids	0.117335	0.018992	0.025219	0.095252	
SoilAlbNIRWet	0.116171	0.016200	0.025704	0.095070	
SoilAlbNIRDry	0.112665	0.014766	0.040809	0.094023	
SoilAlbVisDry	0.111565	0.015490	0.023665	0.097089	
ThermalConductivitySoilSolids	0.109694	0.018087	0.030302	0.079683	
SoilAlbVisWet	0.106393	0.018053	0.026151	0.059908	
VMualem	0.104845	0.019058	0.021472	0.065113	
VegReflectVis	0.104551	0.013805	0.015882	0.067012	
SpecificStorativity	0.102199	0.018316	0.021701	0.078875	
RootDepth	0.100619	0.018432	0.023663	0.080024	
CanDensSurface	0.099391	0.017148	0.027862	0.076559	
WiltingPoint	0.097252	0.013059	0.032982	0.074388	
SoilRoughness	0.096809	0.015322	0.030851	0.062250	
VegTransVis	0.088482	0.016014	0.014867	0.052614	
VegTransNIR	0.077449	0.018123	0.012616	0.032653	

(continues on next page)

(continued from previous page)

VegReflNIR	0.068249	0.018172	0.009401	0.027690
VegHeight	0.037858	0.014848	0.003969	0.005968

	best
FieldCapacity	3.870706e-02
ThetaRes	2.985520e-03
ThetaSat	5.557729e-01
DecayCoeffCanopy	1.165855e+00
NormalHydrConductivity	1.975318e-04
CanopyFraction	8.927573e-01
MinStomatalRes	2.081448e+02
AlphaVanGenuchten	2.450772e-03
NVanGenuchten	1.827232e+00
LSAI	2.178355e+00
SoilEmissiv	4.216430e-01
VegSnowBurying	3.022595e+00
ThermalCapacitySoilSolids	3.709680e+06
SoilAlbNIRWet	6.082924e-01
SoilAlbNIRDry	3.240326e-01
SoilAlbVisDry	3.549442e-01
ThermalConductivitySoilSolids	2.883452e+00
SoilAlbVisWet	2.886453e-01
VMualem	2.914712e-01
VegReflectVis	5.010492e-01
SpecificStorativity	1.643646e-07
RootDepth	5.512460e+02
CanDensSurface	4.069320e+00
WiltingPoint	7.603242e-01
SoilRoughness	1.077662e+00
VegTransVis	2.124748e-01
VegTransNIR	4.469694e-01
VegReflNIR	1.406107e-01
VegHeight	9.226614e+02

CHAPTER

NINE

SCALING ANALYSIS AND MODELING

> There are four kinds of lies: Lies, Damn Lies, Statistics, and Visualizations
>
> —Nathaniel S. Borenstein

In this chapter the question of how calibration time T scale with the number of available processing units (PUs) n will be examined. It is possible to perform two types of scaling analyses, one where the *size of the problem* stays fixed and one where it varies with the number of PUs. As customary, I will refer to them as **weak scaling** and **strong scaling** respectively. What do I mean by *size of the problem* in the context of derivative-free optimization? One obvious answer is the number of function evaluations, or budget, in the parlance of the Nevergrad library; but this number is a random variable when we resample after failing computations. Nonetheless, the budget is the infimum for the actual number of function evaluations and `optimizer.tell` calls, which is the amount of information collected by the optimizer. Therefore, the budget is a good definition for the size of the problem after all, as long as one keeps in mind that he is dealing with a stochastic process. Indeed, each particular realization will not depend in a deterministic way on parameters such as the budget. However, changing the budget **will** change the distribution, and the mean value of calibration time in a deterministic way.

For evolutionary algorithms the budget is given by the number of individuals p times the number of generations g. For these algorithms, as more thoroughly explained in previous chapters, changing the population size can have large effects on the result and execution time of optimization. Indeed, the population size determines, on the one hand, the probability of escaping from a local minimum and, on the other, the degree to which optimization can be executed in parallel. The latter statement, for which we have developed an intuition based on the arguments in the previous chapters, will be given a quantitative form later on in this one. For the previous reasons, in the following I will consider p as the size of the problem, and the number of generations g fixed.

```
import scrapbook as sb
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.formula.api as smf
from datetime import timedelta
from mhpc_project.utils import get_scaling_data
from scipy.optimize import curve_fit

plt.rcParams.update({'figure.figsize':(16,9), 'figure.dpi':100})
```

9.1 Scaling and Efficiency

9.1.1 Strong Scaling

In this analysis, the calibration notebook has been executed several times with increasing n while keeping p fixed. This is done using Papermill, which allows parametrizing and executing Jupyter notebooks in an automated way. Later, using the Scrapbook library, it is possible to recover the objects defined during the execution (which are serialized and saved within the notebook) and the metadata saved by Papermill, such as the notebook execution time.

The Scrapbook library offers the convenience function `read_notebooks` that takes the path to a directory and returns a scrapbook. This object has an iterator interface that yields the notebooks within that directory, and that can be used to collect the data from an ensemble of notebooks.

```
strong_scaling_book = sb.read_notebooks('../runs/strong_scaling')
```

We can retrieve the data relevant to scaling using the `get_scaling_data` function from `mhpc_project.utils`. This function takes a scrapbook and for each of its notebooks checks that every code cell has been successfully executed and, if so, it stores its data in a Pandas dataframe.

```
strong_scaling_data =  get_scaling_data(strong_scaling_book)
strong_scaling_data.head()
```

```
                        name  num_cpus  popsize  num_generations  ratio  \
0  testbed-NGO-4096-1024-40d      1024      512                8    0.5
1  testbed-NGO-4096-1024-7oK      1024      512                8    0.5
2  testbed-NGO-4096-1024-BkB      1024      512                8    0.5
3  testbed-NGO-4096-1024-CoT      1024      512                8    0.5
4  testbed-NGO-4096-1024-KpN      1024      512                8    0.5

      duration  num_samples  samples_duration  num_good_samples  \
0  1610.952014         4932     545846.852303              4096
1  1885.252965         4953     551817.418133              4097
2  1630.464289         5002     556402.759809              4096
3  1636.694933         4871     537770.736557              4096
4  2286.894785         5020     556299.929107              4097

   good_samples_duration  efficiency    speedup
0          421527.591743    0.255531  15.183880
1          425539.367783    0.220430  12.974652
2          422624.253288    0.253130  15.002170
3          423323.413666    0.252583  14.945059
4          420288.756369    0.179474  10.695945
```

As noted above, the calibration time is a random variable and, as shown by the entries in the previous dataframe, it can vary widely from one realization to another, although with the same parameters. However, we will be mainly concerned with its mean value T, and, if not explicitly stated otherwise, I will refer to mean values when talking about execution times (the same goes for its derived quantities).

The speedup S is usually defined as

$$S(n,p) = \frac{T(1,p)}{T(n,p)}.$$

However, since we don't have a serial execution as a reference, the following definition will be used

$$S(n,p) = \frac{T(n_0,p)}{T(n,p)},$$

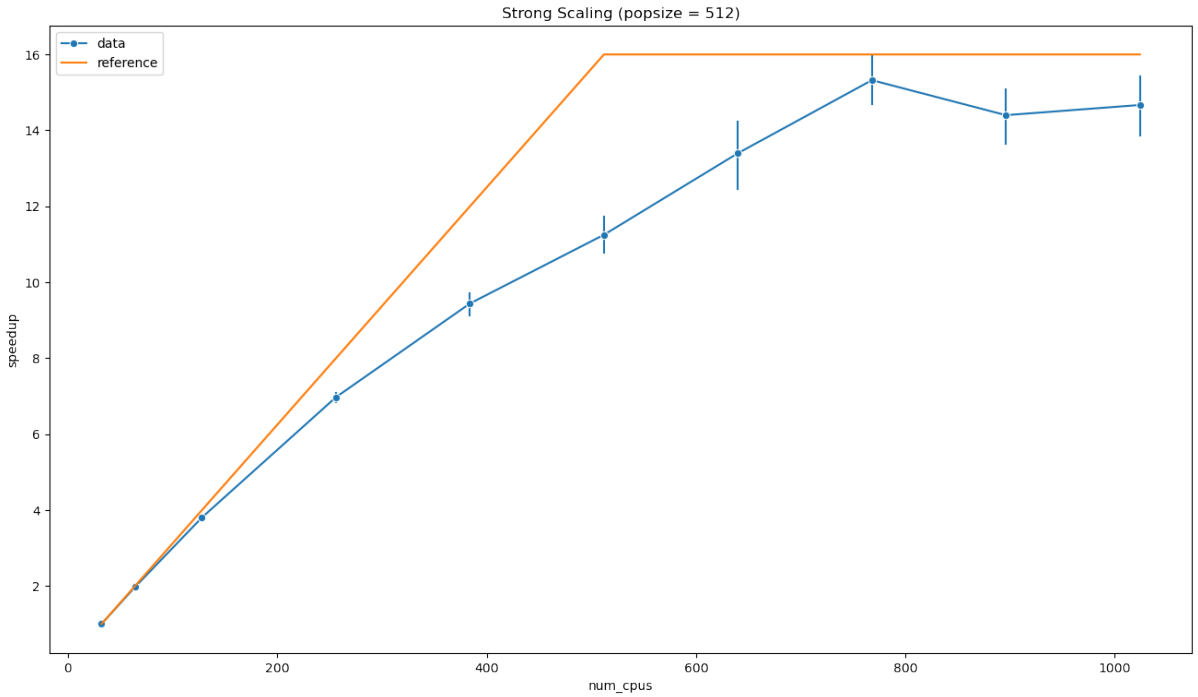
where in our case $n_0 = 32$. In case of perfect (linear) strong scaling, T is inversely proportional to n, and we have

$$S(n,p) = \frac{n}{n_0}.$$

It is possible to improve this simple model of the computation by including the effect of the population size. If the objective never fails, each PU exceeding the population size will idle. Hence, we can have perfect scaling only up to $n = p$

$$S(n,p) = \frac{\min(n,p)}{n_0}. \tag{9.1}$$

```
ax = sns.lineplot(data=strong_scaling_data, x='num_cpus', y='speedup',
             err_style='bars', marker='o', label='data')
sns.lineplot(x=strong_scaling_data['num_cpus'], y=strong_scaling_data[['num_cpus',
↪'popsize']].min(axis=1) / strong_scaling_data['num_cpus'].min(),
             label='reference', ax=ax)
ax.set_title('Strong Scaling (popsize = 512)')
plt.show()
```



As can be evinced from the plot above, the simple model described by the equation {eq}`eq:max_speedup} seems to be very effective. It is interesting to note that the execution appears to reach values near the theoretical maximum speedup, but with n sensibly larger

than p. One possible interpretation is that, as we know, the effective population size is larger than p due to objective function failures. However, in that case the speedup should be also larger than p/n_0. In principle, knowing the statistics of the objective execution times and failures, it should be possible to model more accurately the calibration speedup. However, I will use a simpler approach based on data.

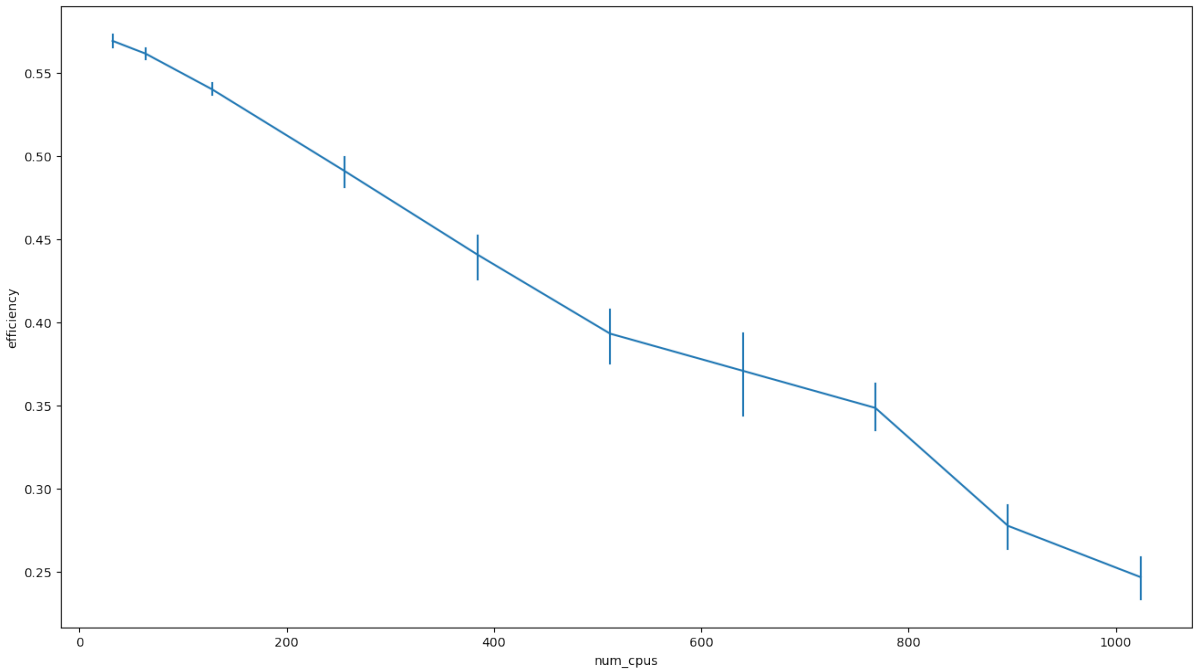
9.1.2 Efficiency

Another interesting quantity is the efficiency, defined as the fraction of time spent by a PU computing the values of the objective actually told to the optimizer

$$\eta(p,n) = \frac{\sum_{x_i \text{ told}} t_i}{nT}.$$

By definition, the remaining part of time is spent idle, computing NaNs or unused values.

```
sns.lineplot(data=strong_scaling_data, x='num_cpus', y='efficiency', err_style='bars')
plt.show()
```



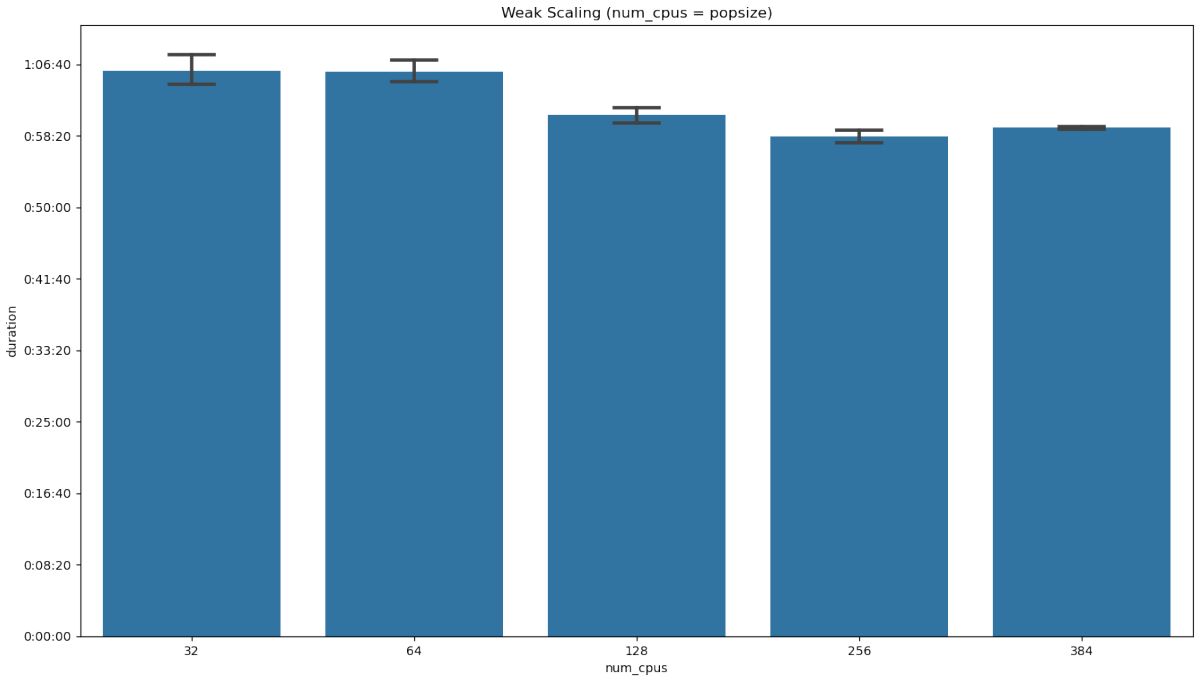
It is worth noting that while we can get a speedup by using more PUs, the efficiency will drop.

9.1.3 Weak Scaling

In weak scaling analysis, the calibration notebook has been executed several times with increasing p and n, while keeping the ration p/n fixed. Note that from strong scaling analysis we know that larger p/n values correspond to smaller errors.

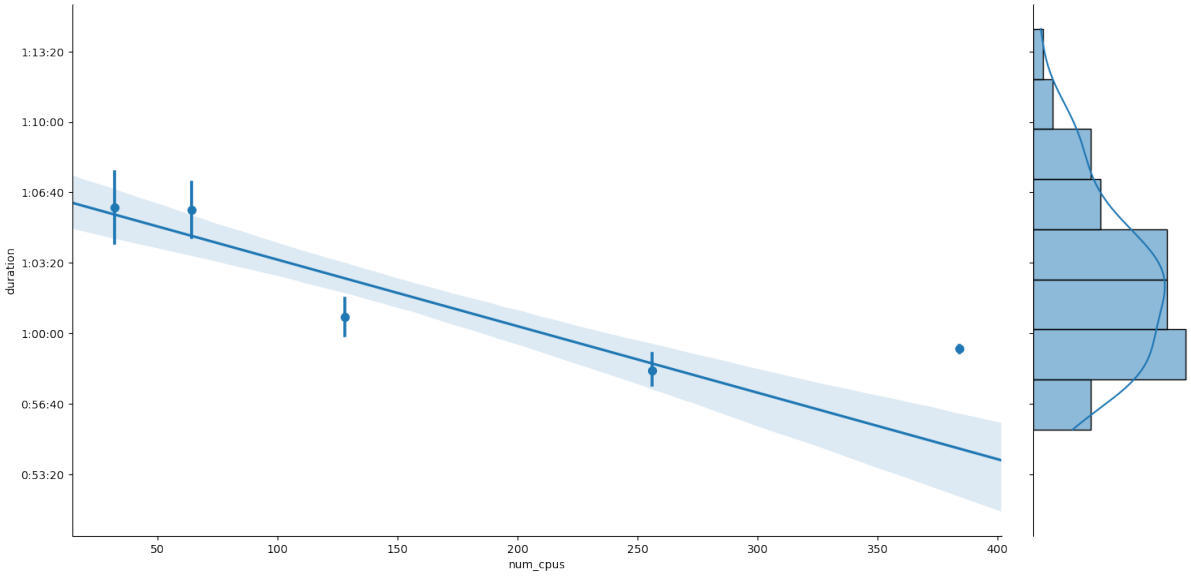
In case of weak scaling, the speedup is not a meaningful metric, since the size of the problem changes. Instead, we are interested in the execution time T, and perfect scaling happens when T is constant.

```
weak_scaling_book = sb.read_notebooks('../runs/weak_scaling')
weak_scaling_data = get_scaling_data(weak_scaling_book)
ax = sns.barplot(data=weak_scaling_data, x='num_cpus', y='duration',
             label='data', capsize=0.2, color="#1f77b4")
ax.yaxis.set_major_formatter(lambda value, position: timedelta(seconds=value))
ax.set_title('Weak Scaling (num_cpus = popsize)')
plt.show()
```



Visual inspection suggests that the hypothesis of perfect weak scaling is compatible with the data within the errors. The same thing can be shown by means of linear regression in the plot and summary below.

```
grid = sns.JointGrid(data=weak_scaling_data, x='num_cpus', y='duration')
grid.fig.set_figwidth(16)
grid.fig.set_figheight(9)
grid.plot_joint(sns.regplot, x_estimator=np.mean, truncate=False)
grid.ax_marg_x.set_axis_off()
sns.histplot(data=weak_scaling_data, y='duration', kde=True, ax=grid.ax_marg_y)
grid.ax_joint.yaxis.set_major_formatter(lambda value, position: timedelta(seconds=value))
plt.show()

smf.ols(formula='duration ~ 1 + num_cpus', data=weak_scaling_data).fit().summary()
```




```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                 duration   R-squared:                       0.531
Model:                             OLS   Adj. R-squared:                  0.524
Method:                  Least Squares   F-statistic:                     72.55
Date:                 Wed, 17 Feb 2021   Prob (F-statistic):           3.95e-12
Time:                         10:57:35   Log-Likelihood:                -432.59
No. Observations:                   66   AIC:                             869.2
Df Residuals:                       64   BIC:                             873.5
Df Model:                            1
Covariance Type:             nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept   3997.5967     35.440    112.798      0.000    3926.797    4068.397
num_cpus      -1.8878      0.222     -8.518      0.000      -2.331      -1.445
==============================================================================
Omnibus:                        7.769   Durbin-Watson:                   1.713
Prob(Omnibus):                  0.021   Jarque-Bera (JB):                7.190
Skew:                           0.779   Prob(JB):                       0.0275
Kurtosis:                       3.434   Cond. No.                         267.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
↪
"""
```

9.2 Linear Scaling Model

Previous results make sense since, without further assumptions, neither perfect weak scaling imply perfect strong scaling (as would contradict our observations), nor vice-versa. Also, the way in which we measure the problem size affects weak scaling. Howeer, if the execution time is proportional to the problem size, then one form of scaling imply the other.

This fact is more evident in equations than words: the two contitions are

$$T(p,n) \propto \frac{1}{n} \qquad \text{for strong scaling}$$
$$T(p,n) = T(\frac{p}{n}) \qquad \text{for weak scaling}$$

hence, if $T(p,n) \propto p$, then one imply the other.

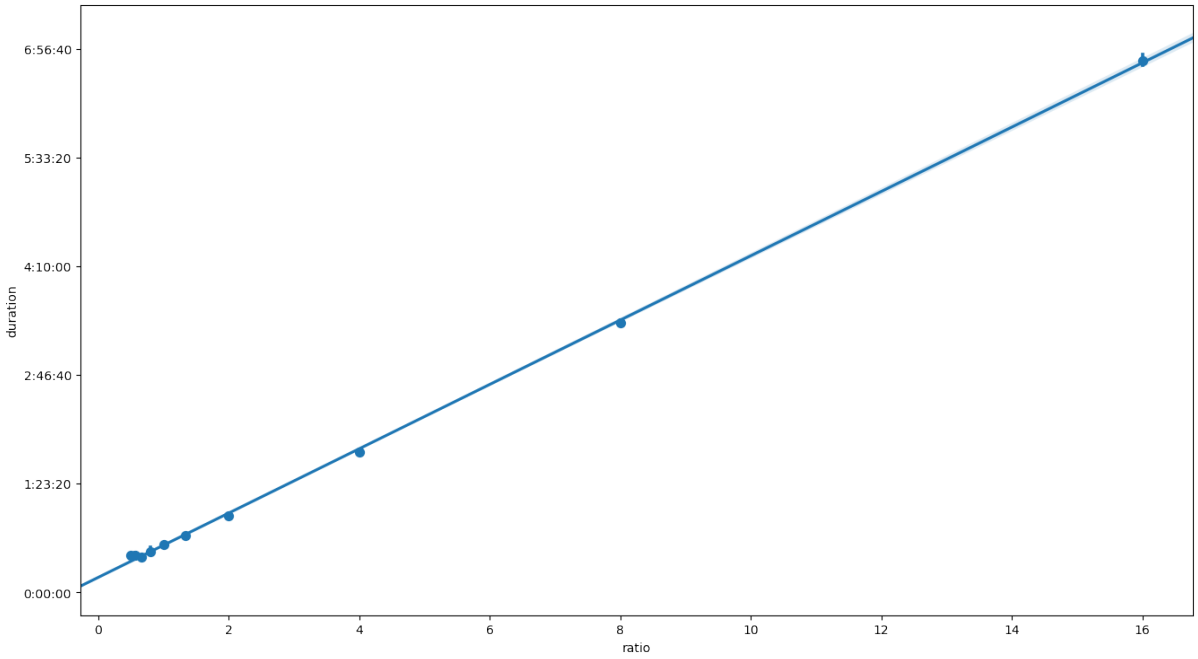
More often, a sensible requirement on the size p is just that $T(p,n) \sim \mathcal{O}(p)$. In this sense, choosing the population size p as the problem size seems very reasonable. Indeed, it is plausible, as a first approximation, that the fraction of successful objective function calls stays constant as we change p: hence, the required amount of computation goes asymptotically as p.

What we can deduce from the previous analyses? From perfect weak scaling we know that T is a function of p/n alone. From strong scaling we know that the speedup for large n seems to saturate at a non-zero value, which means that T cannot have neither poles nor zeros for finite (of course, non-negative) p/n. Therefore, the simplest form for T is a linear model

$$T(p,n) = T_0 + T_1 \frac{p}{n}\,. \tag{9.2}$$

Let's see if this model fits the data and what can be deduced from it.

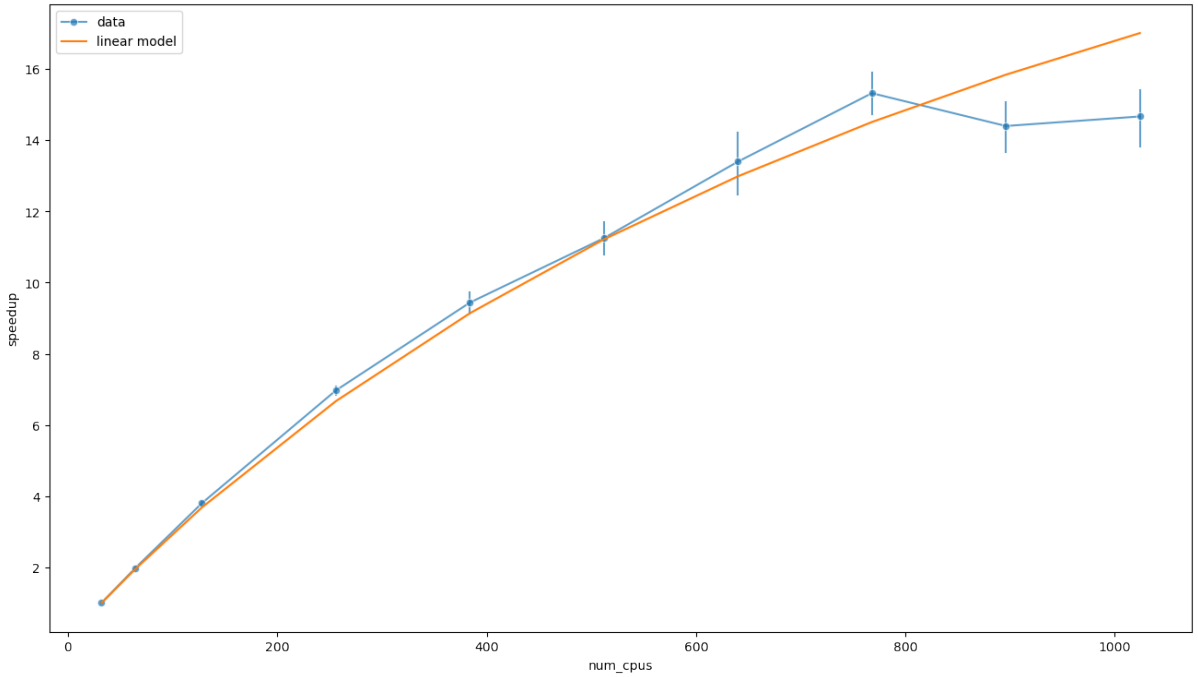
```
data = strong_scaling_data
ax = sns.regplot(data=strong_scaling_data, x='ratio', y='duration', x_estimator=np.mean,␣
↪truncate=False)
ax.yaxis.set_major_formatter(lambda value, position: timedelta(seconds=value))
plt.show()

fit = smf.ols(formula='duration ~ 1 + ratio',data=data).fit()
fit.summary()
```




```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:               duration   R-squared:                       0.998
Model:                            OLS   Adj. R-squared:                  0.998
Method:                 Least Squares   F-statistic:                 9.484e+04
Date:                Wed, 17 Feb 2021   Prob (F-statistic):          4.06e-219
Time:                        10:57:36   Log-Likelihood:                -1116.9
No. Observations:                 158   AIC:                             2238.
Df Residuals:                     156   BIC:                             2244.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    692.1514     28.355     24.410      0.000     636.142     748.161
ratio       1478.7292      4.802    307.957      0.000    1469.244    1488.214
==============================================================================
Omnibus:                       56.222   Durbin-Watson:                   1.968
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              132.789
Skew:                           1.533   Prob(JB):                     1.46e-29
Kurtosis:                       6.281   Cond. No.                         7.43
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
↪
"""
```

Linear regression of the optimization time T gives a strong indication that equation (9.2) might be the right candidate. The same functional relation can be represented also in the scaling plot.

```
ys=(fit.params[0] + fit.params[1] * 512 / 32) / fit.predict(data['ratio'])
sns.lineplot(data=data, x='num_cpus', y='speedup', alpha=0.7, marker='o', err_style='bars',
 ↪ label='data')
sns.lineplot(x=data['num_cpus'], y=ys, label='linear model')
plt.show()
```



We can also explicitly calculate the speedup

$$S(p,n) = \frac{T_0 + T_1\frac{p}{n_0}}{T_0 + T_1\frac{p}{n}} = \frac{1}{\frac{T_0}{T_0+T_1\frac{p}{n_0}} + \frac{T_1\frac{p}{n_0}}{T_0+T_1\frac{p}{n_0}}\frac{n_0}{n}},$$

and compare this expression with the Amdahl's law (corrected to use $T(p,n_0)$ as a reference)

$$S(n) = \frac{1}{(1-f) + f\frac{n_0}{n}},$$

where f is the fraction of the program that can benefit from the speedup.

Therefore, we have that

$$f = \frac{T_1\frac{p}{n_0}}{T_0 + T_1\frac{p}{n_0}} \approx 97\%$$

or, equivalently, that the fraction of time that the program spend in serial code (that is not split among PUs, i.e. the internals of the optimizer) is approximately the 3%. Note that this has nothing to do with the data dependency that exists between a generation of individuals and the next, which actually limits the possibility to scale the execution on larger numbers of PUs.

We can also calculate the maximum speedup

$$S_{\text{max}} = \frac{1}{1-f} = 1 + \frac{T_1}{T_0}\frac{p}{n_0} \approx 35,$$

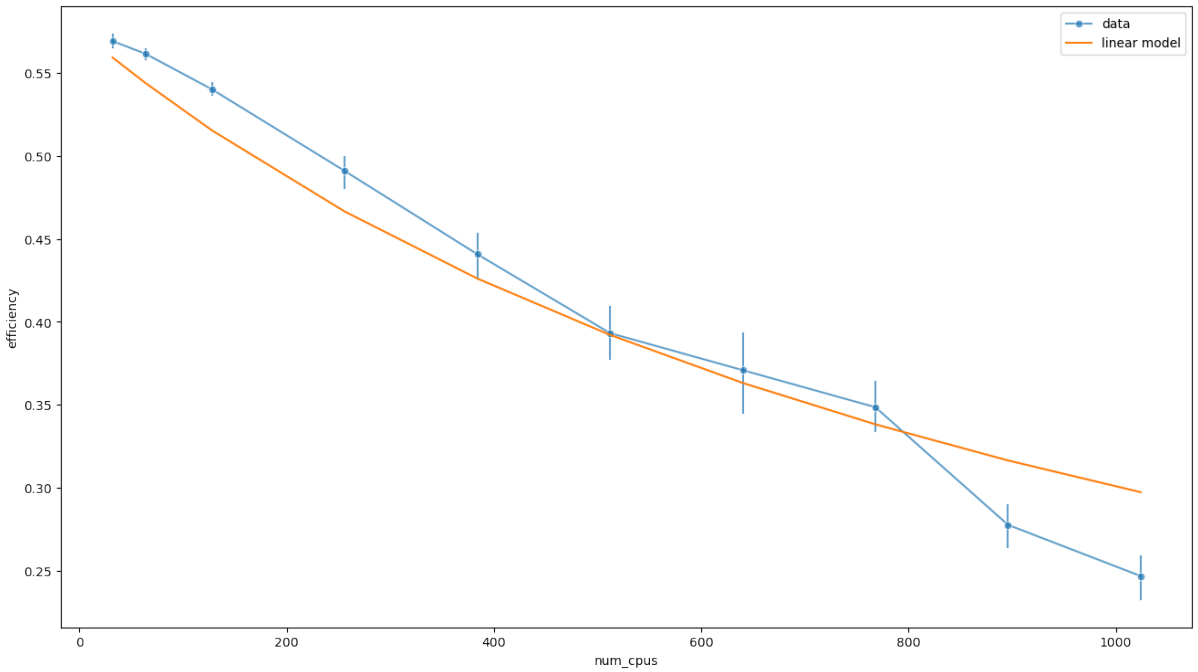
which is sensibly higher than predicted using the simple model based on the assumption of perfect strong scaling up to p.

In this case, the previous argument would suggest an effective population size of $p_{\text{eff}} \approx n_0 S_{\max} \approx 1120$ individuals. The new estimate of $S_{\max}$ is however too high, and the reason is that the linear model considered fails for small p/n, that is large n. We will address this issue later.

We can use the formula for T also to express the efficiency. Since the number of `optimizer.tell` calls is approximately p, then, for large enough p, we can approximate the sum in the definition of η as g times p times the mean execution time of successful computations $\langle t_s \rangle$

$$\eta(p,n) = \frac{p}{n}\frac{g\langle t_s\rangle}{T} = \frac{g\langle t_s\rangle}{\frac{n}{p}T_0 + T_1}.$$

```
average_sample_duration = (data['good_samples_duration'] / data['num_good_samples']).
↪mean()
predicted_efficiency = data['num_generations'] * average_sample_duration / (fit.params[1]␣
↪+ fit.params[0] * data['num_cpus'] / data['popsize'])
sns.lineplot(data=data, x='num_cpus', y='efficiency', alpha=0.7, marker='o', err_style=
↪'bars', label='data')
sns.lineplot(x=data['num_cpus'], y=predicted_efficiency, label='linear model')
plt.show()
```



The maximum efficiency is reached for $n \ll p$ and is equal to

$$\eta_{\max} = \frac{g\langle t_s\rangle}{T_1}.$$

```
data['efficiency'].max(), (data['num_generations'] * data['good_samples_duration'] / data[
↪'num_good_samples']).mean() / fit.params[1]
```

```
(0.5826281218414213, 0.5756400950552586)
```

The story holds up: on the one hand, gpT_1 is the amount of work that can be split among workers, i.e. the average computation time times the budget. On the other, if we consider a serial execution with a large population (that is $n = 1$ and $p \gg 1$), then the efficiency is the ratio between the average time spent on a successful computation and the average computation time per unit budget $\langle t \rangle$. In both cases $T_1 = g\langle t \rangle$. Therefore, the maximum efficiency is $\eta_{\max} = \frac{\langle t_s \rangle}{\langle t \rangle}$, and is characteristic of the objective function, the optimizer and the timeout.

Also, notice that there is an implicit dependency on g in T_1, and hence in $\langle t \rangle$. Indeed, as the generations pass, the optimizer explores regions the search space associated to smaller and smaller losses, and it is plausible that the objective function fails less and less. Hence, the value of $\langle t \rangle$ calculated within a generation is expected to decrease from one generation to the next. In principle, there could be a also a dependency on p, but the data for large values of p/n seems to exclude that (more on this topic in the next section).

Therefore, we can estimate T_1 from the data in this other way.

```
t1_estimate = (data['num_generations'] * data['samples_duration'] / data['num_good_samples
↪'']).mean()

fit.params[1], t1_estimate
```

```
(1478.7292162544938, 1222.7824143247535)
```

Notice that $\langle t \rangle$ can be larger than the timeout of the objective function. The reason is that there could be multiple failing computations per successful one.

9.3 Models for Large Numbers of CPUs

The previous model fails to describe the situation for large n. The reason is that when n is larger than the effective population size, adding more PUs should decrease sublinearly the execution time. Indeed, if we neglect the effects due to the statistics of successful computations, T should be constant. For this reason, I considered other two models which introduce a scale x_0 for p/n separating the two regimes.

The first one is a piecewise linear function

$$T_{\text{piecewise}}(p, n) = \begin{cases} T_0 + T_1 \left(\frac{p}{n} - x_0 \right) & \text{if } \frac{p}{n} \geq x_0 \\ T_0 & \text{otherwise} \end{cases},$$

the other one is the lowest order rational function with vanishing slope at the origin and same asymptotic behaviour

$$T_{\text{rational}}(p, n) = T_0 + T_1 \frac{x_0 \left(\frac{1}{x_0} \frac{p}{n} \right)^2}{1 + \frac{1}{x_0} \frac{p}{n}}.$$

We can fit the speedup using these models for the execution time. Since the speedup is defined as a ratio, it does not depend on the unit of measure of T. This also means that it is not a function of both T_0 and T_1 but depends only on their ratio. Since the behaviour we

are trying to capture is at $p/n \to 0$, the three models are asymptotically equivalent, and the error on T_1 in the fit of the linear model is small, it is convenient to consider T_1 fixed to the previously calculated value.

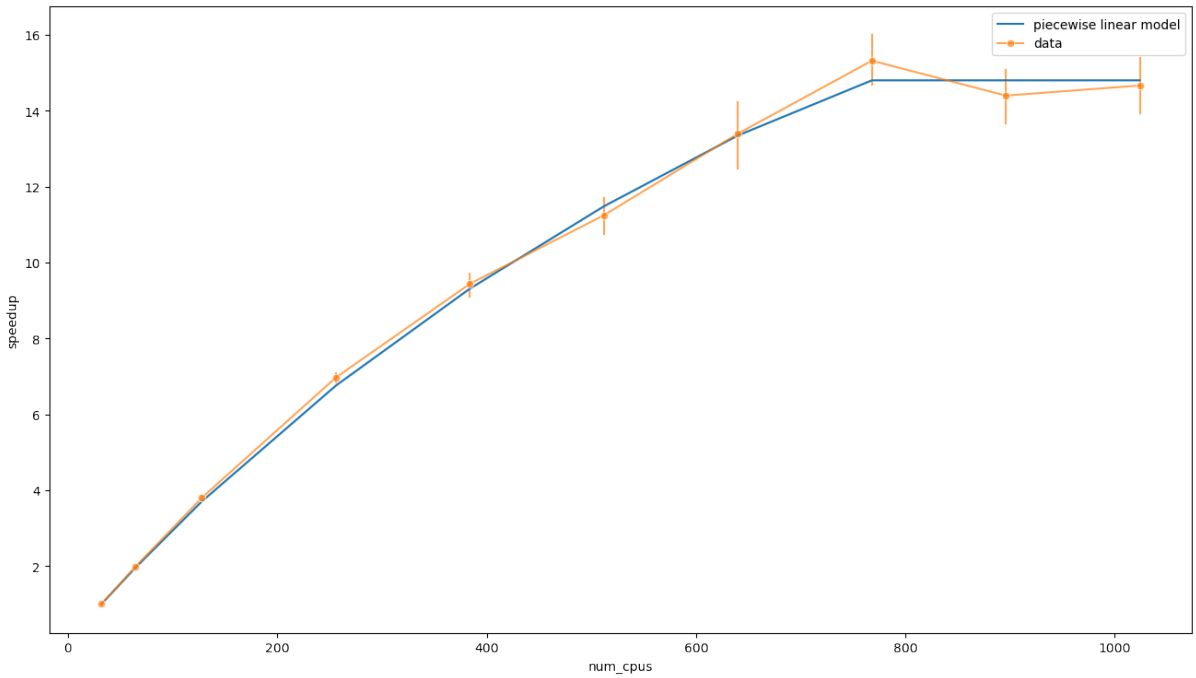
We can now fit the piecewise model,

```
def speedup_from_t(t, n, params, p=512, n0=32):
    return t(np.asarray([p / n0]), *params) / t(p / n, *params)

def t_pwise(xs, t0, t1, x0):
    return np.apply_along_axis(lambda col: np.piecewise(col, [col >= x0, col < x0],␣
 ↪[lambda x: t0 + t1 * (x - x0), t0]), 0, xs)

(t0_pwise, x0_pwise), cov = curve_fit(lambda x, t0, x0: speedup_from_t(t_pwise, x, [t0,␣
 ↪fit.params[1], x0]),
                                      data['num_cpus'], data['speedup'],
                                      bounds=[(0.5 * fit.params[1], 0.4),(2 * fit.
 ↪params[1], 0.8)])
(t0_pwise, x0_pwise), np.sqrt(np.diag(cov))
```

```
((1641.178379546566, 0.6792061302521527), array([17.60840277,  0.02223963]))
```

and plot the results

```
ax = sns.lineplot(x=data['num_cpus'], y=speedup_from_t(t_pwise, data['num_cpus'], [t0_
 ↪pwise, fit.params[1], x0_pwise]), label='piecewise linear model')
sns.lineplot(data=data, x='num_cpus', y='speedup', alpha=0.7, marker='o', err_style='bars',
 ↪ label='data', ax=ax)
plt.show()
```



Visual inspection confirms that there is a good overlap between the data and this model. Indeed, there might even be overfitting. We can also interpret T_0 and x_0 and use their values to calculate the maximum speedup, which is

$$S_{\text{max, piecewise}} = 1 + \frac{T_1}{T_0}\left(\frac{p}{n_0} - x_0\right) \approx 15\,.$$

This value is basically the average speedup at $n = 1024$.

The parameter T_0 is the lower bound for the execution time (i.e. when the work is split among infinite PUs), and the value of $T_0 \approx 27$m is much more reasonable than $T_0 \approx 14$m found in the linear model. Indeed, in the limit $p/n \to 0$, we can estimate this parameter

$$T_0 \approx g\langle t_s \rangle + \Delta T_0$$

where ΔT_0 is the amount of time spent by the system starting the Dask cluster, doing plots, etc. We can calculate this number using the available data

```
t0_pwise - (data['num_generations'] * data['good_samples_duration'] / data['num_good_
↪samples']).mean()
```

```
789.9625529408412
```

that is $\Delta T_0 \approx 13$m.

It is also interesting to explicit the meaning of x_0. In the piecewise model this parameter is the value of p/n at which the time abruptly stops scaling: this means that the number of PUs has exceded the effective population, hence

$$p_{\text{eff}} \approx \frac{p}{x_0} \approx 740\,.$$

This value is also the optimal number of PUs.

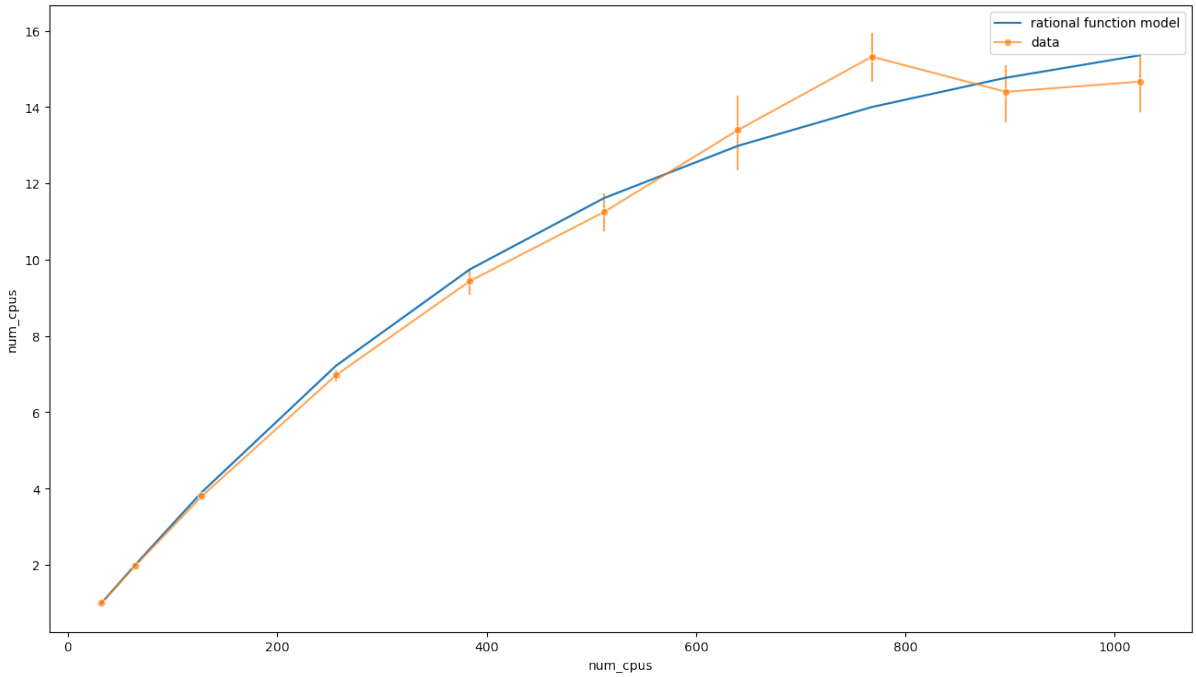
We can fit in the same way the rational model

```
def t_rat(x, t0, t1, x0):
    return t0 + t1 * x0 * (x / x0) ** 2 / (1 + (x / x0))


(t0_rat, x0_rat), cov = curve_fit(lambda x, t0, x0: speedup_from_t(t_rat, x, [t0, fit.
↪params[1], x0]),
                                  data['num_cpus'], data['speedup'], bounds=(0,␣
↪np.inf))
(t0_rat, x0_rat), np.sqrt(np.diag(cov))
```

```
((1286.5499151493627, 0.9916561641074632), array([34.62543519,  0.18726127]))
```

plot the result

```
ax = sns.lineplot(x=data['num_cpus'], y=speedup_from_t(t_rat, data['num_cpus'], [t0_rat,␣
↪fit.params[1], x0_rat]), label='rational function model')
sns.lineplot(data=data, x='num_cpus', y='speedup', alpha=0.7, marker='o', err_style='bars',
↪ label='data', ax=ax)
plt.show()
```



and see that the behaviour at large n is quite different. In this case we have

$$S_{\text{max, rational}} = 1 + \frac{T_1}{T_0} \frac{x_0 \left(\frac{1}{x_0}\frac{p}{n_0}\right)^2}{1 + \frac{1}{x_0}\frac{p}{n_0}} \approx 23\,.$$

Also, in this case, the estimated serial time

```
t0_rat - (data['num_generations'] * data['good_samples_duration'] / data['num_good_samples
↪']).mean()
```

```
435.3340885436379
```

is much less, $\Delta T \approx 7\text{m}$.

All in all, the piecewise linear model seems to fit better the data, although it is a bit less realistic. Putting all together we have

$$T(g, p, n) = \Delta T_0 + g\langle t_s \rangle \left[1 + \eta_{\text{max}} \max\left(0, \frac{p}{n} - x_0\right)\right],$$

where $\langle t_s \rangle$ depend on the objective, η_{max} on the objective and the algorithm, and x_0 (hence p_{eff}) on the objective, the algorithm and on g. For optimization algorithms where there is no notion of convergence, such as random search, x_0 does not depend on g. For the others, the value of p_{eff} is expected to decrease for increasing g, since, as the generations pass, the objective should fail less and less. If one considers the change of x_0 due to g negligible, the previous formula allows estimating the scaling of calibration for a fixed combination of objective and algorithm using only two points, and only one point if ΔT_0 is known.

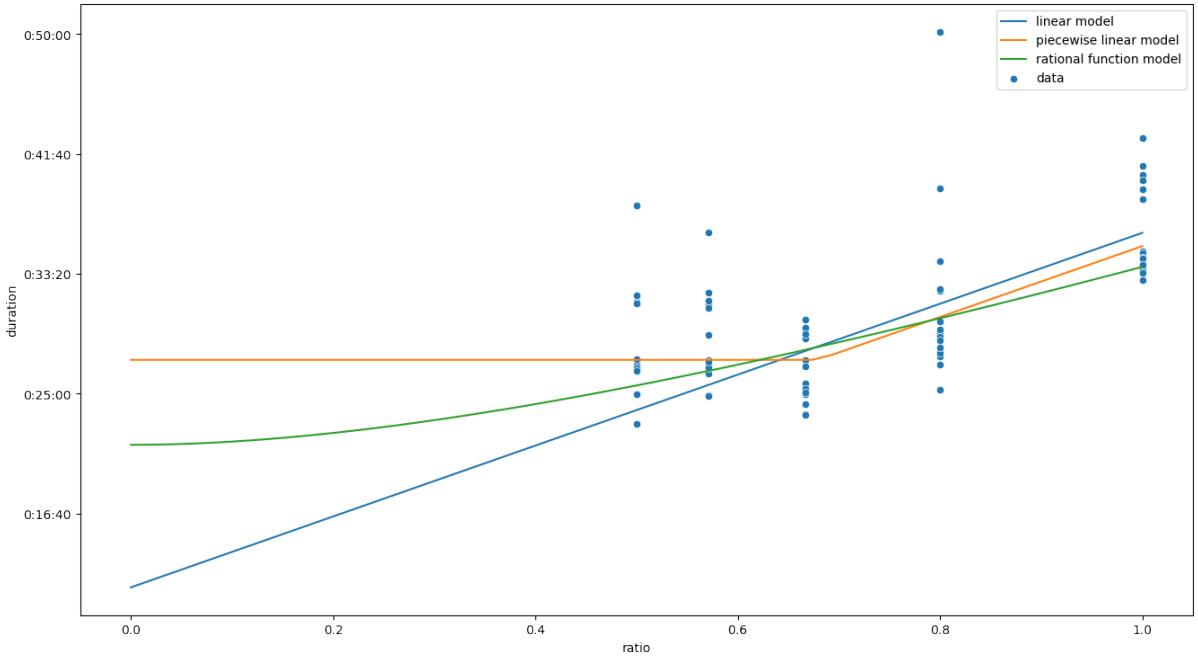
Finally, it is instructive to take a look at the different models for execution times near $p/n \to 0$, or equivalently to the speedups at $n \gg p$.

```
xs = np.linspace(0, 1)
ax = sns.scatterplot(data=data[data['ratio'] <= 1.0], x='ratio', y='duration',␣
↪estimator=np.mean, label='data')
```

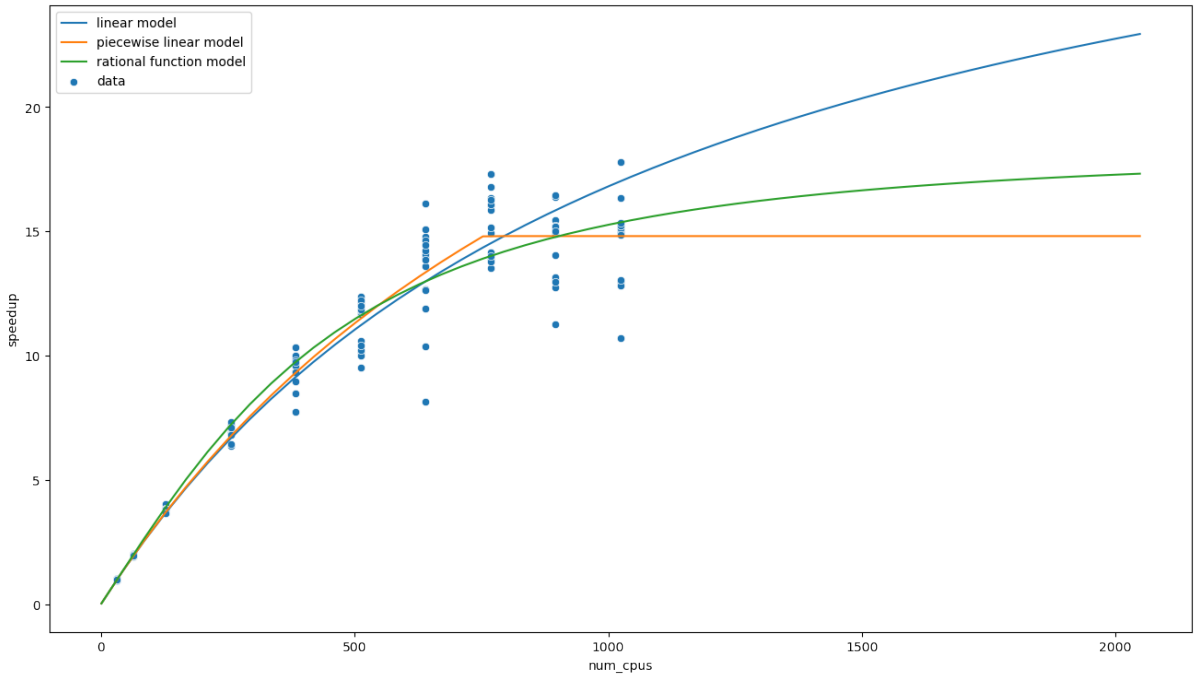
(continues on next page)

(continued from previous page)

```
sns.lineplot(x=xs, y=fit.params[0] + fit.params[1] * xs, label='linear model', ax=ax)
sns.lineplot(x=xs, y=t_pwise(xs, t0_pwise, fit.params[1], x0_pwise), label='piecewise␣
↪linear model', ax=ax)
sns.lineplot(x=xs, y=t_rat(xs, t0_rat, fit.params[1], x0_rat), label='rational function␣
↪model', ax=ax)
ax.yaxis.set_major_formatter(lambda value, position: timedelta(seconds=value))
plt.show()
```




```
xs = np.linspace(1,2048)
ax = sns.scatterplot(data=data, x='num_cpus', y='speedup', estimator=np.mean, label='data
↪')
sns.lineplot(x=xs, y=speedup_from_t(lambda x, t0, t1: t0 + t1 * x, xs, fit.params), label=
↪'linear model', ax=ax)
sns.lineplot(x=xs, y=speedup_from_t(t_pwise, xs, [t0_pwise, fit.params[1], x0_pwise]),␣
↪label='piecewise linear model', ax=ax)
sns.lineplot(x=xs, y=speedup_from_t(t_rat, xs, [t0_rat, fit.params[1], x0_rat]), label=
↪'rational function model', ax=ax)
plt.show()
```


linear model
piecewise linear model
rational function model
data
speedup
num_cpus
0
5
10
15
20
0
500
1000
1500
2000

CHAPTER

TEN

CONCLUSIONS

In this thesis a modern HPC approach has been applied to calibrate the GEOtop hydrological model [RBO06][EGDallAmicoR14], a complex, over parameterized hydrological model, with the aim of predicting the time evolution of variables as soil water content and evapotranspiration for several mountain agricultural sites in South Tyrol.

After developing a Python wrapper for the GEOtop code, I applied the derivative-free optimization algorithms implemented in the Facebook Nevergrad Python library [RT18] on a HPC cluster, thanks to the Dask framework [Tea16].

Particular care has been put in the implementation in order to properly treat model failures, as the model does not produce a valid solution for all combinations of parameters.

The use of HPC solutions allowed calibrating GEOtop using HPC within a reasonable time and with acceptable results, notwithstanding the large parameters space. The code developed, which is published and freely available on GitHub, also shows how libraries and tools used within the machine learning community could be useful and address Earth-system and environmental models calibration. Finally, a simple performance model for the computation has been discussed.

Some examples of further research topics are:

- empirical studies on the choice of optimization algorithms and hyperparameters,
- determination of the optimal population size,
- use of local optimization algorithms for refinement of the solutions, and
- interpretation and validation of the solutions from a physical point of view.

CHAPTER

ELEVEN

FINAL THOUGHTS BEYOND THE SCOPE OF THE THESIS

> It just happened to be an unusual experience. By training I was a scientist: by vocation I was a writer.
>
> —C. P. Snow, *The Two Cultures*

> [... Man] has no time to be anything but a machine. How can he remember well his ignorance–which his growth requires–who has so often to use his knowledge?
>
> —Henry David Thoreau, *Walden; or, life in the woods*

11.1 Apology of the Thoughtful Dabbler

The work presented in this thesis is faceted, and placed at the intersection of hydrology, black-box optimization and HPC. For reasons of space and time, I focused just on the last.

Given the diversity of topics involved, the extent of the material, the time assigned to this project, its focus on HPC, and finally my background, I had to *use some tools* (algorithms, concepts, etc.) without fully mastering them. This unavoidable fact is reflected in the frugality of the bibliography and in their presentation, which occasionally could be sloppy or contain plain errors. The responsibility for those is mine and mine only. However, I hope that the material presented here, if not the subject for more in-depth and broader research by the author, will be at least a prompt for more expert readers.

The growth of complexity in science, to which specialism was the universal response, is not going to decline; but maybe the compartmentalization of specialism will. When we will seek for systematic answers to the problems posed by this Cambrian explosion, one place to look will be computer science, which under many regards is the art of managing complexity by the human mind through abstractions. The simplest and most ubiquitous abstraction is the black box, and hence there is no shame in using black boxes when dealing with problems outside our competence. However, in order to make scientific statements about them, neglecting their inner workings, one needs to use the strictest rigour on the assumptions on their inputs and outputs.

It is unlikely that scientists will be replaced by scientific programmers in the future. However, good scientists surely will also be good programmers, i.e. they will be able to express elegantly both declarative knowledge by means of equations and of procedural knowledge, with the help of a computer.

11.2 Rage Against Machine Learning

The main problem of present scientific research is the adoption of the ideas and methods from capitalist market economy. Research institutes must profit from the work of researchers and profit is measured by publications, funding, and patents. The phenomenon is not new, but the degree to which it permeates academia is unedited. Also, it is tightly linked to the sociopolitical and economic system in which we live, and it is probably irreversible. This problem not only intoxicates the academic environment and the life of people which work in academia, but also affects negatively the quality of research. It creates positive feedback loops and inflates some research topics while stagnating others. We live in a time of a profound, yet silent, crisis in science.

Of course, technology and applied science, which have an immediate return, are more likely to be funded and hence get more attention from researchers. In some cases, the situation is exacerbated by lack of scientific contents and rigour. Private investments in the loop worse the situation.

Prominent examples of speculative bubbles are HPC and topics in machine learning as artificial neural networks. The latter deserves a word of caution. The ubiquitous application of artificial neural networks for modeling and inference, as surrogate human understanding, is the utmost failure of reductionism. We should invest more resources in their understanding, even if it is less rewarding than simply using them. Otherwise, the use of mathematical language, scientific method, and reductionism that fueled the most spectacular achievements in our understanding of nature is in danger.

Some problems in modern science requires HPC, although fewer than promised by exa-scale evangelists. Nonetheless, is it HPC an interesting scientific topic per se? I think that it is. Understanding the performance of large distributed systems, running thousands of coordinated processes simultaneously, is an interesting subject indeed. However, it needs a paradigm shift: it should be investigated for the sake of it, out of pure curiosity. The accent should be on comprehension, not on making things and doing stuff. Paradoxically, it is from gratuitous knowledge that the greatest technological advances come out, in the long run.

A related situation exists in computer science as a whole. Its cultural relevance and brief history rich of beautiful ideas are not recognized. As a consequence, programming is often taught and learned abysmally. The resources for these practices have no shortage of terms: tutorials, cookbooks, howtos. In the words of Kevlin Henney, from a GOTO talk of 2018.

> There's something else in software that we are particularly bad at: we have a very weak sense of history. So it is not simply that we keep rediscovering and reinventing the wheel, and eventually we might actually make it round; it's that we have a very poor cultural sense of history, and so we live in a constant state of astonishment and rediscovery.

At best, computer science is misunderstood, as wonderfully explained by Abelson and Sussman in their classic on the subject [ASPS96].

> Underlying our approach to this subject is our conviction that "computer science" is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called procedural epistemology–the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing

> precisely with notions of “what is”. Computation provides a framework for dealing precisely with notions of “how to”.

If we don’t free ourselves from utilitiarian views on computer science and programming, we will be doomed to poorly reinvent the wheel. Only when we will clearly see their beautiful ineffectiveness, we will be able to make progress.

It is at least naive to suggest stopping using computers for science, and appreciate procedural epistemology for his own sake only. Still, there are practical reasons to approach computing more thoughtfully.

In many research fields, articles, reviews, and conference proceedings account only for a part of the writing duty of researchers, the other being computer programs. Nonetheless, this is something for which they are not always given the proper credit. Furthermore, code is not always published, and its quality is not always at best. However, if programming is theory-building [Nau85], then we need to consider code as literature and write correct, clear, reusable and extensively documented code to make real scientific progress and not waste our collective efforts.

The issue is not just the consideration we have for programming as an intellectual activity. If one wants to apply the scientific method to numerical experiments, he has to attain the same standards used for laboratory experiments: in other words, they must be reproducible. Hence, the reason of the importance of code and documentation in scientific computing is that they are part of the reproducibility effort.

11.3 Math is the Ultimate Javascript Framework

I would go as far as saying that a playful interest in fundamental questions, beyond practical applications, is a good professional investment for a programmer too.

During the last Christmas holidays, I was chatting with my father about medium wave and longwave antennas. Since physical evidence is better than words, he suddenly went away and reappeared from the closet with a radio in his hand. It was a Grundig from the mid 80’s. Once opened the back of the radio, the ferrite rod antenna and the long and medium wave windings were clearly visible.

I was mesmerized and stared intensely at the inner of the radio. The whole circuitry contained less than ten bipolar junction transistors, for both amplitude and frequency modulation. I pointed with my finger at some components I wasn’t able to recognize, and asked him what they were and why they were there, as I used to do as a kid. The strangest thing to me were the small radio frequency inductors. They had small red marks made with permanent ink. A long time before, he explained to me, he marked the original positions of the tuning screws, before trying to adjust the tune. A whole piece of the circuit board was covered with wax—which I thought was glue—, to avoid changing the position and orientation of some components, changing the inductance of the circuits and losing the right tuning.

That world is gone[1]. We now live in a world of integrated circuits. In our world, physical devices are assembled using tools similar to the ones found in software: abstractions. Black-boxes are combined, or replaced with equivalent ones. Most of the time, it is simply cheaper to use programmable electronics.

[1] I can’t help thinking how awfully sad this is. That world was beautiful and understandable. It was made of things you could tear apart and put back together, things that you could touch and smell, things that would let you the time to think.

This transformation is concerned with physical objects only to a lesser degree. The biggest shift was cultural. A generation of highly trained engineers has become obsolete. The knowledge and mental models they used were outdated. It has not happened during a century, but in few decades. It is highly unlikely that it will not happen again. Instead, if we extrapolate the trend, it is conceivable that the timescale of transformations will become shorter and shorter, due to technological acceleration.

In short, there are excellent chances that the knowledge about a particular technology that one acquires today will be worthless within the time span of his professional career[2]. It is probable that our high-level code will look to future programmers like assembly looks to us today, i.e. something that should be generated and manipulated by machines and not humans. All the more reason for believing that our carefully hand-unrolled loops will be considered as primitive as the monkeys screaming around the monolith in 2001: A Space Odissey by Stanley Kubrick[3]. The GPT-3 model offers a hint of how this might happen.

Are we doomed to learn useless stuff? I think that the answer is no. Nonetheless, I think that learning a particular technology, which does not teach us something more general, is a waste of time. Details of implementation will come and go[4], general questions will stay: the deeper the longer. Bartosz Milewski ends with the following words an article on his blog [Mil20] titled "Math is your insurance policy".

> I'm often asked by programmers: How is learning category theory going to help me in my everyday programming? The implication being that it's not worth learning math if it can't be immediately applied to your current job. This makes sense if you are trying to locally optimize your life. You are close to the local minimum of your utility function and you want to get even closer to it. But the utility function is not constant–it evolves in time. Local minima disappear. Category theory is the insurance policy against the drying out of your current watering hole.

We will need new tools however. From the preface of in Category Theory for Programmers [Mil18]

> There is an unfinished gothic cathedral in Beauvais, France, that stands witness to this deeply human struggle with limitations. It was intended to beat all previous records of height and lightness, but it suffered a series of collapses. Ad hoc measures like iron rods and wooden supports keep it from disintegrating, but obviously a lot of things went wrong. From a modern perspective, it's a miracle that so many gothic structures had been successfully completed without the help of modern material science, computer modelling, finite element analysis, and general math and physics. I hope future generations will be as admiring of the programming skills we've been displaying in building complex operating systems, web servers, and the internet infrastructure. And, frankly, they should, because we've done all this based on very flimsy theoretical foundations. We have to fix those foundations if we want to move forward.

[2] Frontend developers experience this inconvenience on a daily basis. However, here I am not really talking about javascript frameworks.

[3] Actually, this is true as of today.

[4] As well as languages, libraries, compilers, and beloved compiler flags.

BIBLIOGRAPHY

[EGDallAmicoR14] S. Endrizzi, S. Gruber, M. Dall'Amico, and R. Rigon. Geotop 2.0: simulating the combined energy and water balance at and below the land surface accounting for soil freezing, snow cover and terrain effects. *Geoscientific Model Development*, 7(6):2831–2857, 2014. URL: https://gmd.copernicus.org/articles/7/2831/2014/, doi:10.5194/gmd-7-2831-2014.

[RT18] J. Rapin and O. Teytaud. Nevergrad — a gradient-free optimization platform. 2018. Last accessed on 2021-2-17. URL: https://GitHub.com/FacebookResearch/Nevergrad.

[RBO06] Riccardo Rigon, Giacomo Bertoldi, and Thomas M. Over. Geotop: a distributed hydrological model with coupled water and energy budgets. *Journal of Hydrometeorology*, 7(3):371 – 388, 01 Jun. 2006. URL: https://journals.ametsoc.org/view/journals/hydr/7/3/jhm497_1.xml, doi:10.1175/JHM497.1.

[Tea16] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: https://dask.org.

[Cor15] E. Cordano. Geotopoptimpso calibration/optimization wrapper r package for the hydrological model geotop. 2015. Last accessed on 2021-3-21. URL: https://github.com/ecor/geotopOptim.

[CDF15] E. Cordano, Andreis D., and Zottele F. Geotopbricks: an r plug-in for the distributed hydrological model geotop. 2015. Last accessed on 2021-3-21. URL: http://cran.r-project.org/package=geotopbricks.

[B+18] E. Bortoli and others. Reengineering and optimization of geotop software package. 2018.

[EGDallAmicoR14] S. Endrizzi, S. Gruber, M. Dall'Amico, and R. Rigon. Geotop 2.0: simulating the combined energy and water balance at and below the land surface accounting for soil freezing, snow cover and terrain effects. *Geoscientific Model Development*, 7(6):2831–2857, 2014. URL: https://gmd.copernicus.org/articles/7/2831/2014/, doi:10.5194/gmd-7-2831-2014.

[VG80] M. Th. Van Genuchten. A closed-form equation for predicting the hydraulic conductivity of unsaturated soils. *Soil science society of America journal*, 44(5):892–898, 1980.

[BFG01] D. Bagley, B. Fulgham, and I. Gouy. The computer language benchmarks game. Python. 2001. Last accessed on 2021-2-17. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python.html.

[Col20] M. Coletti. Github. Scaling distributed dask to run 27,648 dask workers on the Summit supercomputer. 2020. Last accessed on 2021-2-17. URL: https://github.com/dask/distributed/issues/3691.

[CHP+13] Y. Collette, N. Hansen, G. Pujol, D. Salazar Aponte, and R. Le Riche. Object-oriented programming of optimizers–examples in scilab. *Multidisciplinary Design Optimization in Computational Mechanics*, pages 499–538, 2013.

[HMullerK03] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18, 2003.

[RT18] J. Rapin and O. Teytaud. Nevergrad — a gradient-free optimization platform. 2018. Last accessed on 2021-2-17. URL: https://GitHub.com/FacebookResearch/Nevergrad.

[RBZ+19] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, and others. Ten simple rules for writing and sharing computational analyses in jupyter notebooks. 2019. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1007007, doi:10.1371/journal.pcbi.1007007.

[SKU18] M. Seal, K. Kelley, and M. Ufford. The netflix tech blog, Part 2: Scheduling Notebooks at Netflix. 2018. Last accessed on 2021-2-17. URL: https://netflixtechblog.com/scheduling-notebooks-348e6c14cfd6.

[CHP+13] Y. Collette, N. Hansen, G. Pujol, D. Salazar Aponte, and R. Le Riche. Object-oriented programming of optimizers–examples in scilab. *Multidisciplinary Design Optimization in Computational Mechanics*, pages 499–538, 2013.

[GKYM09] Hoshin V Gupta, Harald Kling, Koray K Yilmaz, and Guillermo F Martinez. Decomposition of the mean squared error and nse performance criteria: implications for improving hydrological modelling. *Journal of hydrology*, 377(1-2):80–91, 2009.

[NS70] J.E. Nash and J.V. Sutcliffe. River flow forecasting through conceptual models part i — a discussion of principles. *Journal of Hydrology*, 10(3):282–290, 1970. URL: https://www.sciencedirect.com/science/article/pii/0022169470902556, doi:https://doi.org/10.1016/0022-1694(70)90255-6.

[EGDallAmicoR14] S. Endrizzi, S. Gruber, M. Dall'Amico, and R. Rigon. Geotop 2.0: simulating the combined energy and water balance at and below the land surface accounting for soil freezing, snow cover and terrain effects. *Geoscientific Model Development*, 7(6):2831–2857, 2014. URL: https://gmd.copernicus.org/articles/7/2831/2014/, doi:10.5194/gmd-7-2831-2014.

[RT18] J. Rapin and O. Teytaud. Nevergrad — a gradient-free optimization platform. 2018. Last accessed on 2021-2-17. URL: https://GitHub.com/FacebookResearch/Nevergrad.

[RBO06] Riccardo Rigon, Giacomo Bertoldi, and Thomas M. Over. Geotop: a distributed hydrological model with coupled water and energy budgets. *Journal of Hydrometeorology*, 7(3):371 – 388, 01 Jun. 2006. URL: https://journals.ametsoc.org/view/journals/hydr/7/3/jhm497_1.xml, doi:10.1175/JHM497.1.

[Tea16] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: https://dask.org.

[ASPS96] H. Abelson, G.J. Sussman, A.J. Perlis, and J. Sussman. *Structure and Interpretation of Computer Programs*. Electrical engineering and computer science series. Addison-Wesley, 1996. ISBN 9780262510875.

[Mil18] B. Milewski. *Category theory for programmers*. Blurb, 2018. URL: https://github.com/hmemcpy/milewski-ctfp-pdf.

[Mil20] B. Milewski. Math is your insurance policy. 2020. Last accessed on 2021-2-17. URL: https://bartoszmilewski.com/2020/02/24/math-is-your-insurance-policy.

[Nau85] Peter Naur. Programming as theory building. *Microprocessing and microprogramming*, 15(5):253–261, 1985.