IPVS

Masterarbeit

Load-balancing for multi-physics simulations

Juri Leonhard Schröder

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. habil. Miriam Mehl
Supervisor:	M.Sc. Amin Totounferoush
Commenced:	December 3, 2018
Completed:	June 3, 2019

Abstract

In many real-world scenarios, multiple different kinds of physics appear together in the same system. In order to predict the behavior of such a system, they need to be combined into one *multi-physics simulation*. Simulations using *partitioned coupling* approaches have proved to be especially efficient concerning resource usage and development costs. They divide the simulation domain into distinct subdomains based on the occurring physics, and then solve them separately using single-physics solvers. This makes them suited for execution on modern supercomputers since they are able to profit off the massively available parallelism. Albeit only when the available cores are distributed in accordance with the load of the single-physics solvers. Otherwise, we face resource wastage and unnecessary increases in run-time. The most commonly used approach to this problem is to estimate the load of the single-physics solvers by comparing their degrees of freedom and then scaling the number of cores accordingly.

This thesis proposes a new approach based on empirical performance analysis. By employing machine learning techniques, predictive models for the run-time of the single-physics solvers are created. Based on these models ideal core assignments are derived through solving of an integer optimization problem. To generate the models two approaches are considered: The first one creates several different regression models and then picks the best fitting one, whereas the second one uses neural networks to approximate the solver run-time. Both of them allow us to incorporate new parameters into the models in addition to the number of cores and degrees of freedom. This enables generalization to previously unseen parameter combinations, for example, new discretization levels.

For a simple test case, the regression approach successfully predicts the solver run-time with high accuracy, leading to performance improvements of over 40% compared to the old load-balancing approach. When considering multiple parameters, the neural network approach generally outperforms the regression approach.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

Single core capabilities of CPUs have stagnated in recent years instead, the number of cores per system is increasing. Especially supercomputers may have up to several millions of cores. For example, the currently [1] fastest computer "IBM Summit" has a peak performance of 200,794.9 TFlop/s and a total of 2,397,824 cores [SDSM18]. Furthermore, Intel and the US Department of Energy recently announced the building of the first exascale supercomputer capable of more than 1,000,000 TFlop/s [Cha19]. Of course, this ever-increasing parallelism requires applications that are able to exploit the power of these machines and creates a demand for software capable of leveraging their full potential.

Large-scale simulations proved to be one of such applications. They are utilized when practical experiments are not feasible or too expensive and theoretical reflections do not suffice. They find use in vehicle and aircraft design, medicine, environment, and many more fields. As diverse as their applications are the goals of their usage: they can be used to find flaws in constructions, predict outcomes, or gaining a better understanding of the simulated system in general. Multi-physics simulations are a special kind of simulation, in which several kinds of different physics occur in the same system. There are two possible strategies for realizing this type of simulation. We can either formulate one equation system capturing the entirety of the physical framework and solve it at once using a single solver, or we divide the system based on the occurring physics into different domains. Then, we can simulate each of the domains separately by a single-physics solver and exchange the necessary information. The former one is called the *monolithic* coupling approach, whereas the latter is referred to as *partitioned* coupling.

Both approaches have advantages and disadvantages. For some instances, monolithic approaches are more efficient and robust since they can be developed specifically for the given problem. Partitioned coupling, on the other hand, simplifies the development process because it allows reuse of matured solvers. Additionally, it provides more flexibility, e.g., by allowing different mesh resolutions in different sub-domains. Lastly, it enables a new layer of parallelism by allowing the domains to be solved at the same time (depending on the coupling scheme) which increases efficiency and resource usage. Therefore, this thesis focuses on partitioned coupling.

[1] as of November 2018

1.1 Partitioned Multi-Physics Simulations

A partitioned multi-physics simulation is composed of N different domains. Together, they describe one physical system which is simulated by one single-physics solver per domain. For the remainder of this thesis the words “domain”, “solver” and “participant” are used interchangeably to refer to a partition of the simulation. Common application scenarios are, for example, Fluid-Structure-Interaction, Fluid-Acoustic-Interaction, and Conjugate-Heat-Transfer.

The procedure for performing a partitioned multi-physics simulation can be summarized as follows: We begin by defining the participants and simulation parameters, such as timestep length, the maximum number of timesteps $t_{\max}$, data mappings and coupling scheme. The data mapping determines how boundary values of one domain are mapped to its neighbors and coupling schemes describe the procedure for finding an agreement on these values. Both of these are outlined in more detail in the two following sections. After that, we need to generate the meshes for each solver and prepare the simulation. The actual simulation consists of an initialization step followed by $t_{\max}$ timesteps where each timestep can be divided into:

1. Solve equations in each domain, using a single-physics solver and input values for the boundary conditions.
2. Each solver computes a mapping of its data points to neighboring domains.
3. Execute coupling using the defined coupling scheme, i.e. exchange boundary information and possibly rerun the solvers multiple times.

The individual solvers may perform smaller timesteps without exchanging boundary information during the first step. This is called *subcycling* and must not be confused with the timesteps of a simulation. After the simulation has finished, the results can be visualized and evaluated.

1.1.1 Coupling Schemes

The different domains in a multi-physics simulation are separated by a boundary. Since the values on this boundary are shared, the participants need to agree on them, otherwise this may cause numerical instabilities or improper modeling of the real world. Consider the flow over a heated plate in Figure 1.1. It consists of two domains: the fluid domain (i.e. the channel) and the solid domain

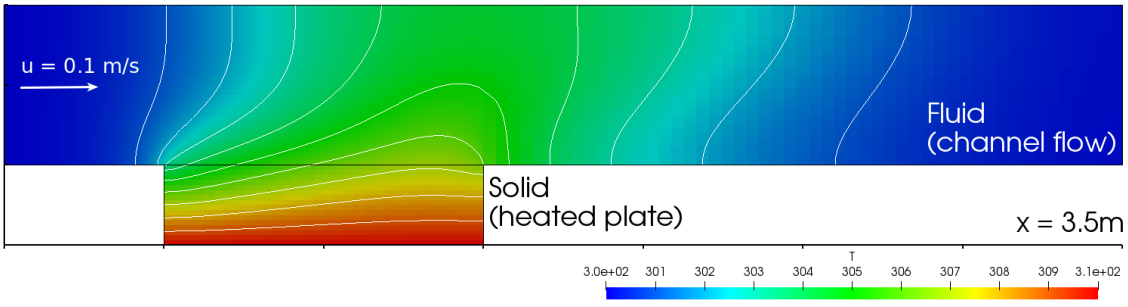


Figure 1.1: Flow of a liquid through a channel over a (solid) heated plate.[Cho19]

(i.e. the heated plate). A fluid enters the channel from the left side with a temperature of 300K,

Algorithm 1.1 Pseudo-code for a simulation using an explicit-serial coupling scheme.

initialize $x_1^{(0)}$
for $t = 1, \ldots, t_{\max}$ **do**
  $x_2^{(t)} \leftarrow S_1(x_1^{(t-1)})$
  $x_1^{(t)} \leftarrow S_2(x_2^{(t)})$
end for

underneath it is a solid plate with a temperature of 310K [Cho19]. Both domains interact with each other: the fluid is heated by the plate and in return changes the plate's heat-flux by drawing thermal energy. To model this, the boundary values of the solid plate serve as input to the fluid flow and vice versa. There are several schemes to implement this so-called *coupling*, depending on properties of the problem and choice of simulation parameters. This section gives a short introduction to the different types of schemes, but only as far as relevant for the sake of this thesis. For further reading and more detailed explanations see [Gat14].

Assume we have two solvers coupled in a partitioned simulation, then they can be modeled as two operators $S_1 : X_1 \mapsto X_2$ and $S_2 : X_2 \mapsto X_1$, each of them taking the output of the other one as input. In the heated plate flow example, S_1 may refer to the fluid solver which takes a heat flux $x_1 \in X_1$ as input and calculates the corresponding temperature on the boundary $x_2 \in X_2$. In return, the solid solver S_2 would take x_2 as an input to calculate the heat flux [Cho17].

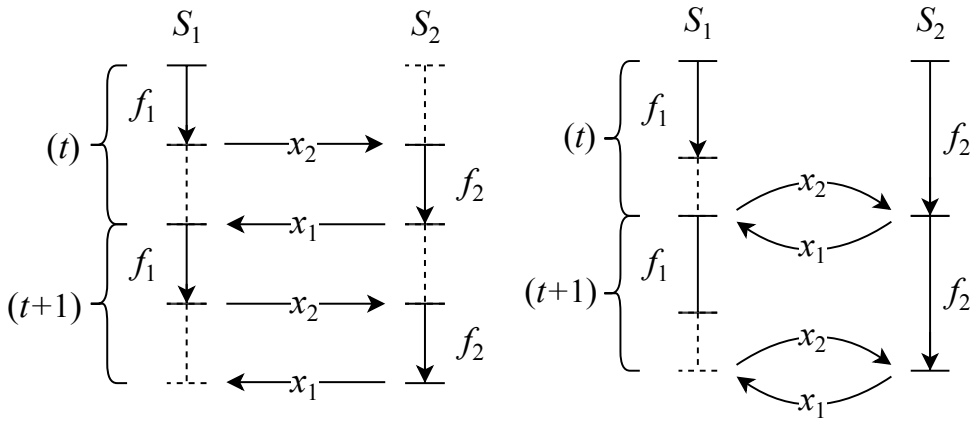
In general, coupling schemes can be distinguished into explicit and implicit schemes. The former calls both solvers a fixed number of times during each timestep, without regard to the convergence of the coupling values. This may lead to numerical instabilities due to an inaccurate representation of the underlying physics. Implicit schemes iterate the solver-coupling until the values on the boundary converge, which is computationally more intensive but prevents the occurrence of instabilities. Additionally, the solvers can either be run in parallel or serial, leading to four distinctions: explicit-serial, explicit-parallel, implicit-serial and implicit-parallel.

In an **explicit-serial** scheme, the first solver S_1 uses the boundary values of the previous timestep $x_1^{(t-1)}$ to solve the domain equations and output new coupling values $x_2^{(t)}$. After the first solver has finished, the second solver S_2 does the same but uses the new coupling values $x_2^{(t)}$ instead. This process is repeated until the end of the simulation $t_{\max}$ is reached. Algorithm 1.1 expresses this scheme as pseudo-code.

Later in this thesis, we look at minimizing the total solve time per timestep (denoted by F). Therefore, we want to express F depending on the run-times of the single-physics solvers f_1 and f_2. To derive such an expression consider Figure 1.2a. It shows the execution of two timesteps of an explicit-serial scheme. The solvers S_1 and S_2 do not run in parallel, hence F is given by the sum over the individual solve times

$$F^{ES} := \sum_i f_i. \tag{1.1}$$

Explicit-parallel schemes use the values of the previous timestep $(x_1^{(t-1)}, x_2^{(t-1)})$ to perform both solving operations in parallel, i.e. lines 3 and 4 in Algorithm 1.2 are executed at the same time. Considering Figure 1.2b we find that the time needed for each timestep is now given by the maximum



(a) Explicit-serial scheme.

(b) Explicit-parallel scheme.

Figure 1.2: Execution diagrams for explicit-serial and explicit-parallel coupling schemes. Dashed lines indicate that the solver is not performing any calculations and the associated cores are idling. *Conv* is the convergence check of the found solution.

Algorithm 1.2 Pseudo-code for a simulation using an explicit-parallel coupling scheme.

1: initialize $x_1^{(0)}, x_2^{(0)}$
2: **for** $t = 1, \ldots, t_{\max}$ **do**
3: $\quad x_2^{(t)} \leftarrow S_1(x_1^{(t-1)})$
4: $\quad x_1^{(t)} \leftarrow S_2(x_2^{(t-1)})$
5: **end for**

of the individual solve times

$$F^{EP} := \max_i f_i. \tag{1.2}$$

Implicit schemes eliminate numerical instabilities, by enforcing the convergence of the coupling values, i.e. at the end of each timestep t it must hold that

$$x_1^{(t)} = S_2(x_2^{(t)}) = S_2(S_1(x_1^{(t)})). \tag{1.3}$$

Implicit-serial schemes achieve this by performing the corresponding fixed-point iterations as described by Algorithm 1.3. Additionally, the solution is usually accelerated and stabilized in a post-processing step, for example, under-relaxation [Cho17]. The execution diagram depicted in Figure 1.3a shows that run-time per timestep is now given by the sum over the run-times for each iteration. Under the assumption that the solve time for the single-physics solvers does not change, we can simply multiply the run-time per iteration (as given by Equation (1.1)) by the number of iterations $j_{\max}$

$$F^{IS} := j_{\max} \cdot \sum_i f_i. \tag{1.4}$$

Implicit-parallel coupling schemes basically perform the same fixed-point iterations. The difference is the parallel execution of the solvers (lines 5 and 6 in Algorithm 1.4).

$$F^{IP} := j_{\max} \cdot \max_i f_i. \tag{1.5}$$

Algorithm 1.3 Pseudo-code for a simulation using an implicit-serial coupling scheme.

initialize $x_1^{(0),0}$
for $t = 1, \ldots, t_{\max}$ **do**
  $j \leftarrow 0$
  while not converged **do**
    $x_2^{(t),j+1} \leftarrow S_1(x_1^{(t),j})$
    $x_1^{(t),j+1} \leftarrow S_2(x_2^{(t),j+1})$
    $j \leftarrow j + 1$
  end while
  $x_1^{(t+1),0} \leftarrow x_1^{(t),j}$ // assign starting value for next iteration
end for

Algorithm 1.4 Pseudo-code for a simulation using an implicit-parallel coupling scheme.

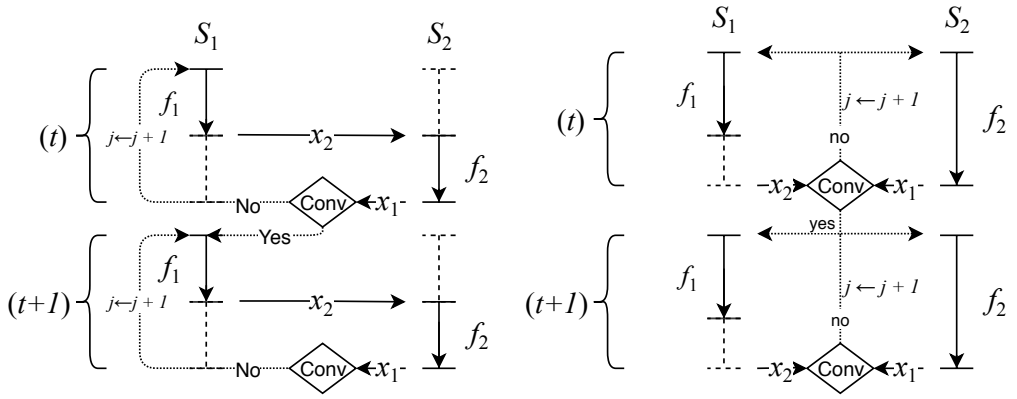
1: initialize $x_1^{(0),0}, x_2^{(0),0}$
2: **for** $t = 1, \ldots, t_{\max}$ **do**
3: $j \leftarrow 0$
4: **while not** converged **do**
5: $x_2^{(t),j+1} \leftarrow S_1(x_1^{(t),j})$
6: $x_1^{(t),j+1} \leftarrow S_2(x_2^{(t),j})$
7: $j \leftarrow j + 1$
8: **end while**
9: $x_1^{(t+1),0} \leftarrow x_1^{(t),j}$ // assign starting values for next iteration
10: $x_2^{(t+1),0} \leftarrow x_2^{(t),j}$
11: **end for**

1.1.2 Data Mapping

In the previous section, we assumed that both solvers provide values for each point on the boundary, but this might not be the case. When we have non-matching meshes, i.e. one solver uses a higher resolution on the boundary than the other, we require rules defining how the missing values are calculated. The different rule sets can be divided into *conservative* and *consistent* mappings. Conservative mappings preserve the integral values of the data points. This is necessary, for example, for forces because physics dictates that they need to be in equilibrium. In comparison, consistent mappings produce the same values for corresponding nodes on both sides of the boundary and are used, for example, for temperatures [Cho17].

There are three commonly used types of mappings:

- **Nearest Neighbor mapping** simply uses the value of the closest point on the source mesh. We therefore only need the position of the vertices.
- **Nearest Projection mapping** linearly interpolates between the values on the source mesh. The receiving mesh then projects its mesh point on the interpolation and uses the values on the projected points.



(a) Implicit-serial scheme. **(b)** Implicit-parallel scheme.

Figure 1.3: Execution diagrams for implicit-serial and implicit-parallel coupling schemes. Dashed lines indicate that the solver is not performing any calculations and the associated cores are idling.

- **Radial Basis Function mapping** creates interpolations using radial basis functions centered around the data points of the source mesh. The interpolation is then evaluated on the vertices of the receiving mesh and the corresponding values are assigned.

1.2 Load-Imbalance

Once more, consider the heated plate flow example in Figure 1.1. Assume that we employ an explicit-parallel coupling scheme, i.e. each timestep involves the following three steps:

1. Solve the fluid domain.
2. Solve the solid domain.
3. Exchange the boundary information.

Where steps 1 and 2 are performed at the same time. Exchanging the boundary information is only possible after both solvers have finished and is necessary before the next timestep can begin. This gives rise to one very important question: What share of the total available cores should be assigned to each solver to achieve maximum performance? Maximum performance, in this case, means minimizing the overall run-time per timestep, thus minimizing the time spent waiting for the other solver to finish.

The most common approach to deal with this problem of load-imbalance is by scaling the number of ranks with the size of the respective domain. If the fluid domain (in the heated plate flow example) had three times the number of elements of the solid domain, then we would assign 75% of the cores to the fluid domain and 25% to the solid domain. While this approach is very simple to realize, it usually does not create optimal results since it disregards many factors which may affect the run-time of the simulation. These factors can be categorized into three classes:

- **System and Architecture properties:**
  Number of cores, clock rate of processors, number of nodes and cores per node [2], communication bandwidth, pipelines, branch prediction mechanisms, caches, etc.
- **Properties of libraries and operating system:**
  The solvers make use of math and communication libraries. The choice of library and its version affect the run-time.
- **Code and problem properties:** Solver and solver settings (e.g. number of iterations), accuracy when solving, length of timesteps, problem size, number of communications, data size for each communication, etc.

This thesis explores two ways of achieving better core assignments than the old approach. Both of them are built around the same principle: by creating prediction models for each solver, the optimal core assignment can be derived, by solving an optimization problem.

1.3 Optimization Problem

Let N denote the number of participants (solvers), $P \geq N$ the total number of available cores and p_i the number of cores assigned to solver i. Furthermore, assume that we know a function $F : \mathbb{N}_+^N \to \mathbb{R}$, which takes the $\vec{p} = (p_1, \ldots, p_N)^T$ as an input and computes the run-time for one timestep. Of course, our goal is to minimize this run-time, as expressed by the following optimization problem:

$$\begin{aligned} \underset{\vec{p}}{\text{minimize}} \quad & F(\vec{p}) \\ \text{subject to} \quad & \sum_{i=1}^{N} p_i \leq P, \;\; p_i \in \mathbb{N}_+. \end{aligned} \tag{1.6}$$

The constraint $\sum_{i=1}^{N} p_i \leq P$ captures that the number of assigned cores must be less or equal to the number of available cores.

To complete the definition we need to define the function F. Assume that we have N functions $f_1, \ldots, f_N \; \forall f_i : \mathbb{N}_+ \to \mathbb{R}$ that predict the run-times of the solvers for a given number of cores p_i, then we can use them to compose the function F depending on the coupling scheme.

For serial schemes, the solvers are not executed in parallel, therefore one might assume that assigning all available cores to all solvers is the best strategy [CHV13]. While this is theoretically possible, the technical realization is difficult [Uek16]: Exchanging the boundary information is often realized by direct point-to-point communication between ranks (see Section 4.1.1), this entails that ranks of both domains need to be active at the same time. Using the considerations of the previous chapter, in particular Equations (1.1) and (1.2), we can derive F for the explicit-serial and explicit-parallel scheme as given in Table 1.1. Their implicit counterparts additionally have a factor of $j_{\max}$ (see Equations (1.4) and (1.5)), but since this factor is unaffected by the number of cores (i.e. the number of iterations until convergence does not change but the speed of each iteration), we can omit it and end up with the same formulas as for the explicit case.

[2] In a supercomputer the cores are usually split up into several nodes, with higher communication bandwidth inside than across nodes.

	serial	**parallel**
explicit	$F(\vec{p}) := \sum_i^N f_i(p_i)$	$F(p) := \max_i f_i(p_i)$
implicit	$F(\vec{p}) := \sum_i^N f_i(p_i)$	$F(p) := \max_i f_i(p_i)$

Table 1.1: Minimization function depending on the coupling scheme.

$$
\begin{array}{rlllllllllll}
(\bullet & \bullet & \bullet & \bullet) \Rightarrow & (\bullet, & \bullet, & \bullet & \bullet), & (\bullet, & \bullet & \bullet, & \bullet), & (\bullet & \bullet, & \bullet, & \bullet) \\
4 \Rightarrow & & & & (1+ & 1+ & & 2), & (1+ & & 2+ & 1), & (2+ & & 1+ & 1)
\end{array}
$$

Figure 1.4: Compositions of 4 into 3 parts.

1.3.1 Solving the Optimization Problem

If we do not make any assumptions about F, the problem in Equation (1.6) is a non-linear, possibly non-convex integer optimization problem. These kinds of problems are generally very hard to solve. Fortunately, there is a simple remedy in our case, which works independently of F: We can simply brute-force all the possible choices for the p_i. This is feasible due to 2 observations:

- The value N is typically small (usually $N \in \{2, 3\}$).
- Evaluations of F are cheap since it is only a composition of the models f_i.

To find the exact number of possible p_i assignments, let us make a simplifying assumption first. Assume that the value of $F(p_1, \ldots, p_N)$ never increases when we increase only one p_i and leave the other ones fixed. This is equivalent to assuming that the total run-time never increases when only increasing the number of cores for one solver.

Theorem 1
Under the assumption that $F(p_1, \ldots, p_N)$ is monotonically decreasing in all p_i, there are $\binom{P-1}{N-1}$ possible p_i assignments.

Proof 1
Due to the assumption, the constraint in Equation (1.6) *can be simplified to $\sum_{i=1}^N p_i = P$ since cases where we do not assign all cores are now irrelevant. Therefore, the number of solutions is equivalent to the number of compositions of P into N parts, where a composition is a concrete assignment $(p_1, \ldots, p_N)$. The number of assignments can be counted as illustrated by the example for $P = 4$ and $N = 3$ in Figure 1.4: Imagine P balls, and between each of them one can either insert a comma or not, with $N - 1$ commas available in total. This will create a unique decomposition of P into N parts with each p_i given by the number of balls between commas. There are $P - 1$ possible places for inserting a comma, hence the number of compositions is $\binom{P-1}{N-1}$ [Tri13].* □

For example, if $N = 2$ and $P = 1000$ we only have to check $\binom{999}{1} = 999$ different assignments, i.e. $(999, 1), (998, 2), \ldots, (1, 999)$.

While the assumption made in Theorem 1 is generally reasonable, there may be instances where it does not hold, for example, when the time spent for communication between the ranks of a solver outweighs the time spent solving.

Theorem 2
There are $\binom{P}{N}$ *possible* p_i *assignments such that* $\sum_{i=1}^{N} p_i \leq P$.

Proof 2
Using Theorem 1 the number of possible assignments can be expressed as $\sum_{i=N}^{P} \binom{i-1}{N-1}$. *By index shifting we get* $\sum_{i=0}^{P-N} \binom{N+i-1}{N-1} = \binom{P}{N}$. □

From Theorems 1 and 2 we can conclude that the brute-force approach of solving the optimization problem is feasible for common values of P and N, especially when we apply the assumption of Theorem 1. Although there are extreme cases where it may not be viable. But if we have an idea about the range of possible p_i values, we can use this knowledge and incorporate it into our search to further reduce the size of the search space.

1.4 Performance Analysis

The only piece missing to be able to use the outlined optimization method for deriving optimal core assignments are the solve-time models $f_1, \ldots, f_N$. Investigating and modeling the performance of programs is called *performance analysis* or *performance modeling*. Previous work by Kerbyson et al. [KAH+01] created performance models for a specific application, by examining its code in order to find dependencies between parameters (problem size, communication bandwidth, …) and the run-time (or other performance metrics). While this is always possible, it requires a lot of work, expert knowledge about the application and is not feasible for very complex applications.

Instead of using analytical models, Calotoiu et al. [CHPW13] automatically create performance models with a regression-based approach. They generate a set of candidates of optimal regressions for varying basis functions and then pick the best model among the candidates. But in order to define suitable basis functions, the nature of the dependence between the parameter and the run-time must be known. Additionally, the time needed to find a model increases exponentially in the number of input dimensions. As a result, the generation process becomes very time consuming for an increasing number of dimensions.

Neural networks do not require predetermined basis functions and have proved to be useful for dealing with higher dimensional problems. By cascading several layers of linear models and applying a non-linear transformation, neural networks can express highly non-linear functions. Oyamada et al. [OZW08] applied them to performance prediction on embedded systems, achieving better results than linear models. Since the release of that paper in 2008, the neural network field experienced a revolution driven by hardware and algorithmic innovations, with plenty of success in many fields. This motivates their usage for our cause.

1.5 Summary

Our goal is to find an optimal distribution of cores to the different solvers involved in a given simulation. Figure 1.5 outlines the steps necessary to achieve this goal.

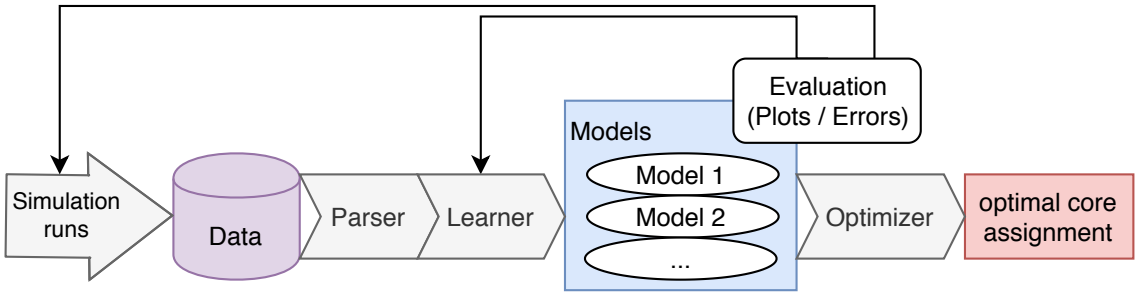


Figure 1.5: Workflow for finding optimal core assignments.

As a preliminary step, we need to collect data, which serves as input for the model generation. In practice, this means that we execute a few timesteps of the simulation and measure the time each solver takes to finish its calculations. It is important to exclude the time needed for communication from the measurements and only consider the time spent solving the equations and calculating mappings. Thus, it is mandatory that the solvers or coupling library provide this information via some kind of log file. Additionally, a set of parameters must be attached to each of the measurements, containing information about the settings of that run. In the simplest case, this is only the number of cores assigned to each solver, but the problem size or any other of the factors mentioned in the previous chapter may also be added.

The resulting data set then has to be parsed and preprocessed. This includes reading the necessary values from the log files, as well as arranging them in a format that is workable by the learners. The learner can be either one of the approaches covered in Chapter 2 or 3. They take the input data and apply machine learning techniques in order to generate one model for each solver involved in the simulation. These models are predictive, in other words, capable of forecasting the run-time of their respective solver for any given combination of input parameters. But they may have flaws, including over-fitting, under-fitting or poor accuracy outside the measured range. There are two possible reasons this may happen: a lack of input data or wrong hyper-parameters for the learner. The user must, therefore, evaluate the created models against his expectations, by looking at plots and different error metrics. In case of problems, he needs to adjust the hyper-parameters or add new simulation runs to the data set to alleviate the models' deficits. As soon as the user is satisfied, the models serve as input to an optimizer which calculates the optimal core assignment.

The goal of this thesis is to provide a framework, which can be used to derive optimal core assignments for arbitrary, coupled multi-physics simulations. We already reduced this problem to finding performance models for the involved single-physics solvers. In order to create these models, we consider two machine learning approaches: Chapter 2 examines the regression-based approach presented by Calotoiu et al. [CHPW13]. The second modeling approach employs neural networks, hence their fundamentals are introduced in Chapter 3. Combining them with the other parts of the pipeline yields an implementation, which is capable of computing optimal core assignments based on input data. Chapter 4 explains the implementation alongside the process and software necessary for generating the needed data. In Chapter 5 the implementation is evaluated for a few example simulations. The final chapter summarizes and concludes the thesis.

2 Regression

Assume that we are given a set of m data points, consisting of pairs (p_k, y_k). Each pair maps a number of ranks p_k to the run-time y_k of a certain algorithm, e.g., one timestep of a simulation. Our goal is to find a function $f : \mathbb{N}_+ \to \mathbb{R}$ which best-possibly predicts the run-time depending on p. This is a core problem in machine learning, and multiple approaches have been proposed to solve it.

Linear regression is one of the best-known methods and has been successfully employed for various problems. It works by building an approximation of the function $f(p_k) \approx y_k$ as a sum of n coefficients $c = (c_1, \ldots, c_n)^T$, multiplied with some basis functions $\phi_l(p)$ (also called features).

$$f(p) := \sum_{l=1}^{n} \phi_l(p) c_l = \phi^T(p) \cdot c, \tag{2.1}$$

where $\phi(p) = (\phi_1(p), \ldots, \phi_n(p))^T$. In the following, the "$\cdot$" for the scalar product is omitted.

Let Φ be the $m \times n$ matrix containing one line per data point, where the entry $\Phi_{k,l}$ is the evaluation of $\phi_l(p_k)$. To determine the coefficients we define a loss function and choose the coefficients such that the loss is minimized. A common choice is the squared error which is given by

$$L^{SE}(c) := \sum_{k=1}^{m} (y_k - \phi^T(p_k)c)^2 = ||y - \Phi c||^2, \tag{2.2}$$

with $y = (y_1, \ldots y_m)^T$. By setting the gradient to zero, we can minimize the loss and find optimal parameters

$$\hat{c} = (\Phi^T \Phi)^{-1} \Phi^T y. \tag{2.3}$$

After finding these coefficients, $f(p)$ can be evaluated for arbitrary p.

In some instances, it is possible that the cross-product matrix $\Phi^T \Phi$ is (almost) singular and hence problematic to invert. Another problem is possible over-fitting, i.e. the model has small errors on the given input data, but does not generalize well to new data [TN17]. We can address both of these problems by adding a small regularization parameter α to the diagonal entries

$$\hat{c} = (\Phi^T \Phi + \alpha I)^{-1} \Phi^T y. \tag{2.4}$$

This is commonly called *Ridge Regression*.

2.1 Performance Model Normal Form

The basis functions ϕ define the space of functions which can be expressed by the regression. Thus, their choice is crucial for the correct representation of the underlying patterns. In our case, we aim to accurately reflect the behavior of computer algorithms. We, therefore, use powers and logarithms of p:

$$\phi(p) = p^i \cdot \log_2^j(p) \tag{2.5}$$

for some i and j. Additionally, we need to choose a value for n in Equation (2.1). This yields the *Performance Model Normal Form* (PMNF) [CHPW13; Shu18]

$$f(p) := \sum_{k=1}^{n} c_k \cdot p^{i_k} \cdot \log_2^{j_k}(p). \tag{2.6}$$

Since this is biased towards 0, we add another term consisting of a single constant c_{n+1} which is then learned during the training process alongside the other constants. This is the same as increasing n by one and setting $i_{n+1}, j_{n+1} = 0$, but for the remainder of this thesis, this additional term is not explicitly stated.

Supposing i_k and j_k are given for $k = 1, \ldots, n$, we can find the coefficients c by arranging the $m \times n$ feature matrix Φ and solving Equation (2.4). To find suitable i_k and j_k, we confine the search to a finite set of values and compare all models in this search space against each other. We denote the range of possible values for i_k and j_k by I and J respectively. Calotoiu et al. [CBE+16] found a choice of

$$n = 2, \;\; I = \{\frac{0}{4}, \frac{1}{4}, \ldots, \frac{12}{4}\}, \;\; J = \{0, 1, 2\} \tag{2.7}$$

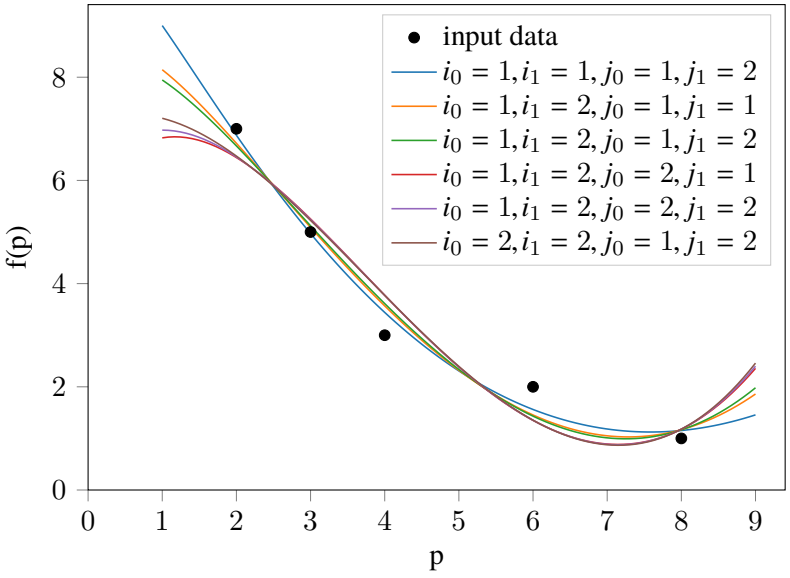
to be suitable for many applications.

A concrete assignment for n and all i_k and j_k is called a *hypothesis* or *model candidate*. Figure 2.1 shows an example data set alongside all the candidates in the search space $n = 2$, $I = \{1, 2\}$, $J = \{1, 2\}$. In order to pick the hypothesis which best fits the data, we need to compare the models using some loss function. A naive approach would be to calculate the squared error from Equation (2.2) on the same data points used for the regression, but this is very prone to over-fitting.

To combat this problem, cross-validation is applied [HBM03]: The input data set is split into training and test data, where the training data is used to create the regression model, and the test data is used for the loss calculation. More concretely, in k-fold cross-validation the set is split into k parts, of which $k - 1$ are used as training data and the remaining one serves as test data. We can repeat this process k times by always choosing a different test set and sum up all the losses. Choosing k equal to the number of data points m is called leave-one-out cross-validation. While this is the most computationally intensive kind of k-fold cross-validation (because a total of m models have to be trained for each hypothesis), it promises the best results for small numbers of data points. This is important because, according to past studies [CHPW13; SCH+15], as few as five data points may suffice for finding accurate performance models. After finding the optimal set of i_k and j_k, we can train on the whole data set to achieve even better results.

p	y
2	7
3	5
4	3
6	2
8	1

(a) Example data values



(b) Candidate regression functions

Figure 2.1: An example of a set of data points and the set of hypotheses for $n = 2, I = \{1, 2\}$ and $J = \{1, 2\}$.

2.1.1 Loss Functions

To compare the hypotheses against each other, we should also consider different kinds of loss functions, instead of focusing solely on the squared error. Reisert et al. [RCSW17] suggest using the **symmetric mean absolute percentage error** (SMAPE).

Examining the squared error in Equation (2.2) we observe two things:

- The error depends on the range of the input data. If we assume that the prediction is wrong by some constant factor, the error increases for large y values and decreases as they get smaller.
- Errors for different data sets are not comparable which makes interpretation hard for users.

Flores [Flo86] initially proposed SMAPE for its application in time series forecasting and defined it as:

$$L^{\text{SMAPE}} := \frac{100\%}{m} \sum_{k=1}^{m} \frac{2 \cdot |y_k - f(p_k)|}{|y_k| + |f(p_k)|}. \tag{2.8}$$

The value $L^{\text{SMAPE}} \in [0\%, 200\%]$ expresses the error of the predictor $f(p)$ as a percentage, i.e. an error of 0% would be perfect. This makes it easily interpretable and comparable across different data sets and models [RCSW17].

The **mean absolute percentage error (MAPE)** is similar to SMAPE but without the symmetry, i.e. the error is expressed as a percentage only of the actual value y_k (instead of both the forecast and the actual value)

$$L^{\text{MAPE}} := \frac{100\%}{m} \sum_{k=1}^{m} \left| \frac{y_k - f(p_k)}{y_k} \right|. \tag{2.9}$$

This means the error can be arbitrarily bad and is not upper-bounded by 200%.

For completeness sake, we also define the **mean squared error**, i.e. the squared error divided by the number of test data points m

$$L^{\text{MSE}} := \frac{1}{m}\sum_{k=1}^{m}(y_k - f(p_k))^2. \tag{2.10}$$

2.2 Extended Performance Model Normal Form

A major disadvantage of the PMNF is that it is only capable of modeling dependencies between a single input parameter (i.e. the number of ranks assigned to the solver) and the output. But the run-time of simulation code may depend on a lot more variables (see Section 1.2), for example, the size of the input problem. Including them in our model allows us to answer questions such as:

- What is the optimal core assignment for a given problem size? Even if the simulation has never been run for this specific problem size.
- How will the optimal run-time change for a constant core count and doubled problem size?

Assuming we have d parameters $p_1, \ldots, p_d$ [1], we can extend the PMNF basis functions (Equation (2.5)) to include them.

$$\phi(p_1, \ldots, p_d) = \prod_{l=1}^{d} p_l^{i_l} \cdot \log_2^{j_l}(p_l). \tag{2.11}$$

The result is the Extended Performance Model Normal Form (EPMNF) as proposed by Calotoiu et al. [CBE+16]:

$$f(p_1, \ldots, p_d) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{d} p_l^{i_{k,l}} \cdot \log_2^{j_{k,l}}(p_l). \tag{2.12}$$

Instead of searching $2n$ parameters i_k and j_k, we now have to find one for each dimension, hence $2nd$ in total. Analogous to the one-dimensional case, we define two sets of possible values for them $i_{k,l} \in I$ and $j_{k,l} \in J$.

Unfortunately, the resulting explosion of the search space quickly becomes a problem. In the single parameter case we have to check $|I| \cdot |J|$ possible combinations for each of the n terms. We do not need to consider models with repeated terms and their order does not matter, resulting in a total of $\binom{|I|\cdot|J|}{n}$ hypotheses. For example, the search space defined in Equation (2.7) contains $\binom{13\cdot 3}{2} = 741$ different models.

In the d-dimensional case, the number of parameters per term increases by a factor of d. Therefore, there are a $(|I| \cdot |J|)^d$ possible combinations per term and $\binom{(|I|\cdot|J|)^d}{n}$ total models [CBE+16]. Expanding the example search space to $d = 2$ dimensions leads to $\binom{(13\cdot 3)^2}{2} = 1,155,960$ candidates, or $\binom{(13\cdot 3)^3}{2} = 1,759,342,221$ for $d = 3$. The number of hypotheses grows exponentially in d and

[1] In Section 1.3 we used the same notation, but referred to different solvers, instead of different parameters of the same solver. For the remainder of this thesis, it is always explicitly stated which one is meant.

n. Additionally, the evaluation of each hypothesis is not trivial since we need to perform all of the cross-validation steps. In conclusion, generating and comparing all of them is generally not feasible, except for very small search spaces.

2.2.1 Heuristics

Calotoiu et al. [CBE+16] propose two heuristics that drastically accelerate the generation process for multi-dimensional models. The first of them prunes the search space to only include hypotheses which are combinations of the best single parameter models. The second one is designed to speed up the search for the best single parameter models.

Hierarchical Search

Hierarchical search aims to reduce the number of evaluated candidates, by selecting only those model-hypotheses which are likely to be the best one. In particular, Calotoiu et al. [CBE+16] operate under the assumption that the best d-dimensional model can be expressed as a combination of the best single parameter models. One important thing to note is, there is no guarantee that the best model of the full search space, is, in fact, part of the restricted one. For example, if the best single parameter models for $n = 1$ and $d = 2$ are $c_1 \cdot p_1^2$ and $c_2 \cdot p_2 \cdot \log_2(p_2)$, we only check the candidates $c_3 \cdot p_1^2 \cdot p_2 \cdot \log_2(p_2)$ and $c_4 \cdot p_1^2 + c_5 \cdot p_2 \cdot \log_2(p_2)$. As before, we then determine the cross-validation based loss for both of these and choose the one with the smaller loss.

If $n > 1$, we need to explore all possible options that can be obtained by combining all subsets of terms. There are 2^n such subsets for each of the d single parameter models, resulting in a total of 2^{nd} combinations. To better understand the impact this has on the number of evaluated hypotheses, let us look at our running example from Equation (2.7) and $d = 3$: In order to find the three best single parameter models, we need to check $3 \cdot 741 = 2223$ hypotheses and then compare the $2^{2 \cdot 3} = 64$ possible combinations, which makes for an overall of 2287. Reducing the full search space of $1,759,342,221$ models to less than 0.02% of its original size.

Modified Golden Section Search

Hierarchical search is based on the best single parameter models, therefore we can accelerate the multi-parameter model construction by speeding up the search for single parameter models. The idea is to reduce the number of evaluated single parameter models by choosing the evaluated hypotheses more intelligently, instead of trying out all of them. The approach is based on finding an ordering of the hypotheses such that the error function is convex and hence has a unique minimum.

Calotoiu et al. [CBE+16] suggest that ranking the models by the magnitude of the first derivative at the data point with the largest parameter value satisfies this condition. We can find the derivative by applying the product rule to each term of Equation (2.6)

$$\frac{df}{dp} = \sum_{k=0}^{n} c_k \cdot \frac{p^{i_k - 1} \cdot \ln^{j_k - 1}(p)}{\ln^{j_k}(2)} \cdot (i_k \ln(p) + j_k), \tag{2.13}$$

where $\ln$ is the natural logarithm.

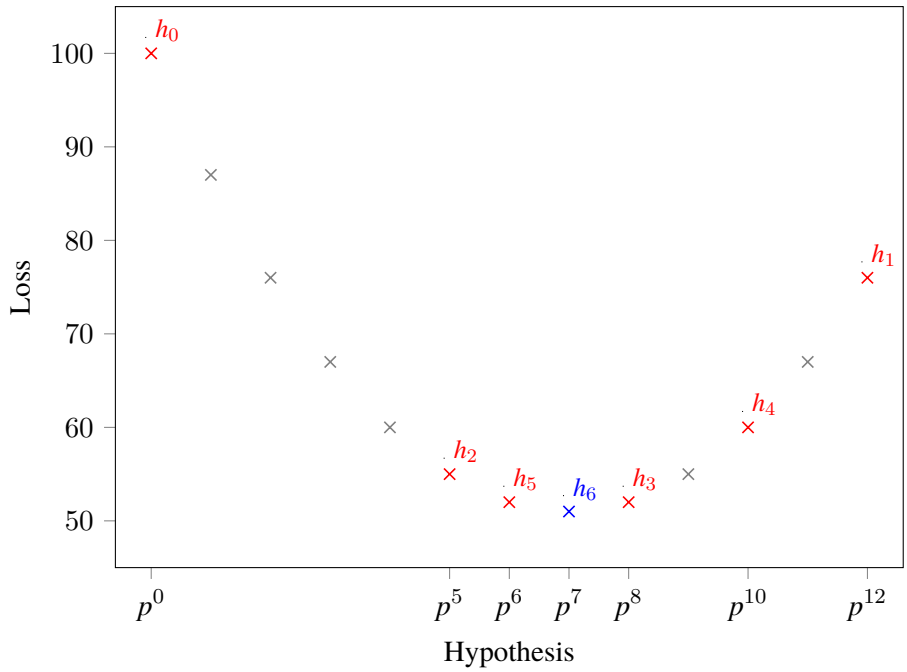


Figure 2.2: Possible error for the example hypotheses and the points chosen by the golden section search.

After ordering the models, golden section search is performed to find the best one. Let us consider an example with $n = 1, I = \{0, 1, \ldots, 12\}$ and $J = \{0\}$. Obviously, the search space then only contains the models $1, p, \ldots, p^{12}$. Assume all $p \geq 1$ (for example p being the processor count), then they are already ordered by the magnitude of their first derivatives (for the largest p).

Golden section search recursively divides the search space and identifies in which interval the minimum is until only one model remains [Kie53]. It begins by evaluating the boundaries of the ordered search space (i.e. h_0 and h_1 in Figure 2.2). Where evaluating means calculating the cross-validation based loss $L(h)$ as described in Section 2.1. Additionally, a third point h_2 is chosen such that $\frac{h_1 - h_2}{h_2 - h_0}$ approximately equals the golden ratio $\varphi = 1.618$ (i.e. by solving the equation for h_2 and rounding to the nearest integer). The fourth point h_3 is chosen in the same fashion, but for the sub-interval $[h_2, h_1]$ instead of $[h_0, h_1]$. Since the function is unimodal, we can now pinpoint in which of the intervals our search needs to be continued. In our case, it holds that $L(h_0) > L(h_1) > L(h_2) > L(h_3)$, thus we can restrict the new search interval to $[h_2, h_1]$. The next point h_4 is chosen so that it divides $[h_3, h_1]$ into two subintervals, with lengths according to the golden ratio. Because $L(h_4) > L(h_3)$ we can exclude the interval $[h_4, h_1]$ and continue with the interval $[h_2, h_4]$. In general, we always exclude the interval which is not adjacent to the current minimum. This process is repeated until the optimal solution h_6 is found. In this example, the application of modified golden section search reduced the number of model evaluations from 13 to 7, when compared to traversing the whole search space. For larger search spaces the advantages become even more apparent: Calotoiu et al. [CBE+16] mention instances where $24,804$ possible candidates can be reduced to as few as 25.

3 Neural Networks

The deployment of deep learning methods has experienced a surge in the past 10 years for various applications. Notably due to their success in pattern recognition and image classification, they have been tried out as a remedy to many challenges. Although neural networks have existed since 1943 [MP43], they have barely been used in practice before the 1980s for two reasons: First, the computers were not sophisticated enough to handle large neural networks. And second, there was no efficient way to train them. These things changed due to technical progress and the introduction of the backpropagation algorithm in 1974 [Wer74]. Over the following years, neural networks were applied to different problems but fell out of favor for more specialized methods. Until new algorithms and improved hardware made them the best choice for certain problems (e.g. image classification [KSH12] and speech recognition [GMH13]) in the early 2010s.

They beat other methods due to the fact, that they are able to learn complex (non-linear) functions, without requiring additional knowledge by domain experts. Additionally, they are able to incorporate large numbers of input parameters into the model, without being limited by dimensionality explosion and the associated increase in training time. Although they have one major drawback: they need a lot of data to find good models.

Our goal is to use them to predict the run-time of the solvers that are involved in the simulation. As discussed in Section 1.2, there are many variables that might affect the solve-time and creating enough data is feasible if we only run a few timesteps for each data point. This makes neural networks a potentially good choice to approach this problem. Therefore, this chapter introduces their key concepts based on Bishop [Bis06].

3.1 Structure and Training

Neural networks exist in many different variants, but we focus on densely connected feedforward networks, as they seem the most appropriate for our use-case. They consist of one input layer, an arbitrary number of hidden layers and an output layer. Figure 3.1 shows a (not densely connected) feedforward neural network with three hidden layers and one output neuron. Each hidden layer is comprised of one or more neurons, whose output is determined by some activation function $\sigma : \mathbb{R} \to \mathbb{R}$. The resulting values are then mapped to the next layer via multiplication with some weight-matrix W. Similarly to regression each layer has a bias term. This is can be interpreted as a neuron without any inputs and constant output of 1. The biases b are the values in the corresponding

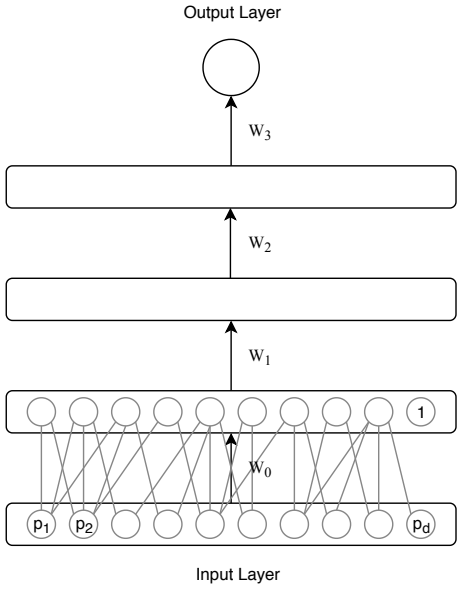


Figure 3.1: A schematic depiction of a neural network with 3 hidden layers.

column of the weight matrix. Neural networks can be used to represent a function with arbitrary output-dimension $f : \mathbb{R}^d \to \mathbb{R}^n$. But since we are interested in using them for predicting run-times, considering NNs with one-dimensional output (i.e. $n = 1$) is enough [1].

Regression is only capable of representing functions that are linear in the input features. To be able to model non-linear functions, the PMNF and EPMNF make use of a "trick": By applying a non-linear transformation to the inputs $p = (p_1, \dots p_d)$ before the regression, they are able to include functions which are combinations of powers and logarithms of p in the function space. The problem is we need to know beforehand what kind of dependencies we are looking for. For the case where we only considered the number of ranks as input to the problem, this worked well, but for other parameters, we might not know a suitable relationship. In comparison, neural networks can represent non-linear functions because the σ are non-linear.

We still need to address how the entries for the weight-matrices W are determined, i.e. how to train the network. Algorithm 3.1 gives an overview of this process. At the beginning, the weights have to be initialized. For example, using Glorot initialization [GB10], whereby the weights are picked from a uniform distribution $U(-a, a)$ with $a = \sqrt{\frac{6}{n_l + n_{l+1}}}$, with n_l and n_{l+1} denoting the number of neurons on layers l and $l + 1$. The weight initialization strategy should be chosen in accordance with the employed activation function. Subsequently, a loop is initiated: For each data point (p, y) we first calculate the output of the network by forward propagation. The inputs of layer l are denoted by z_l and the corresponding activations are x_l, with $x_0 = p$ the activations of the input layer. We use the values of the output layer to calculate the loss $L(f, y)$ and the corresponding gradient $\frac{\partial L}{\partial f}$ for the data point. Possible loss functions are, for example, the mean squared error and mean absolute percentage error. The gradient is then propagated back through the network and used to calculate

[1] Although having $n > 1$ allows for the inclusion of other performance metrics in the model, which may be interesting in case we want to minimize additional metrics, e.g., memory usage.

Algorithm 3.1 Training of a neural network ($\odot$ is the element-wise product).

1. Weight initialization
repeat
  2. Forward propagation:
  $\forall l \in \{1, \ldots, l_{\max}\} : z_l \leftarrow W_{l-1} x_{l-1}\ ,\ x_l \leftarrow \sigma(z_l)$

  3. Calculate loss gradient:
  $f \leftarrow W_{l_{\max}} x_{l_{\max}}$
  $\delta_{l_{\max}+1} \leftarrow \frac{\partial L(f,y)}{\partial f}$

  4. Backpropagation:
  $\forall l \in \{l_{\max}, \ldots, 1\} : \delta_l \leftarrow [\delta_{l+1} W_l] \odot [x_l \odot (1 - x_l)]^T$ // assumes sigmoid activation
  $\frac{dL}{dW_l} \leftarrow \delta_{l+1}^T x_l^T$

  5. Weight update:
  $W_l^{\text{new}} \leftarrow W_l^{\text{old}} - \eta \frac{dL}{dW_l}$ // Gradient descent
until stopping criterion

the loss-gradient δ_l for each layer l with respect to its inputs z_l. The calculation of the loss gradient δ_l depends on the activation function of the neurons. Using the loss-gradient δ_l with respect to the inputs, we then compute the loss with respect to the weights of the layer $\frac{dL}{dW_l}$. The last step is to actually update the weights of each layer. How exactly the weights are updated, depends on the choice of the optimizer (see Section 3.3). With some modifications the loop may also process several data points (also called a *batch*) at a time. It is executed until some stopping criterion (e.g. the loss becomes small or the maximum number of iterations has been reached) is fulfilled. This might include feeding the same data to the network multiple times because the network might not converge after the first pass over the data set. Each pass over the data set is called an *epoch*.

3.2 Activation Functions

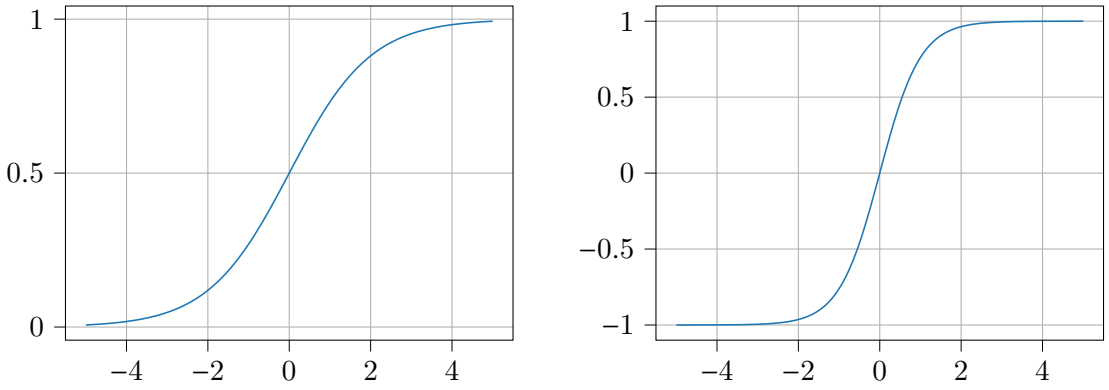
To define a neural network we need to pick a non-linear activation function σ which transforms the input to a neuron to its output. There have been plenty of suggestions for possible functions, all with different properties and use cases. Since it is difficult to predict which choice is the best-suited for our application, this section intends to give an overview of the most common ones.

Sigmoid

The sigmoid activation function is commonly used to introduce NNs and is one of the best-known ones. It is given by

$$\sigma(x) = \frac{e^x}{e^x + 1}. \tag{3.1}$$

This function approaches 1 as $x \to \infty$ and 0 for $x \to -\infty$ as Figure 3.2a shows. From the figure, it is also apparent that for large $|x|$ the gradient becomes very small. This is problematic because the update steps become very small as well, and the network is barely learning.



(a) Sigmoid activation function.

(b) Hyperbolic tangent activation function.

Figure 3.2: Sigmoid and $\tanh$ activation functions.

Hyperbolic tangent

The hyperbolic tangent function is similar to the sigmoid activation function, but its output covers the interval $(-1, 1)$ instead of $(0, 1)$. It is shown in Figure 3.2b and defined as:

$$\sigma(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)}. \tag{3.2}$$

Just like the sigmoid function, the hyperbolic tangent suffers from the problem of vanishing gradients.

Rectifier

Glorot et al. [GBB11] showed that rectified linear units (ReLUs) perform better for training NNs than sigmoid and hyperbolic tangent functions. As of 2017, they are therefore the most common activation function among the deep learning community [RZL17]. In the simplest case, it is given by

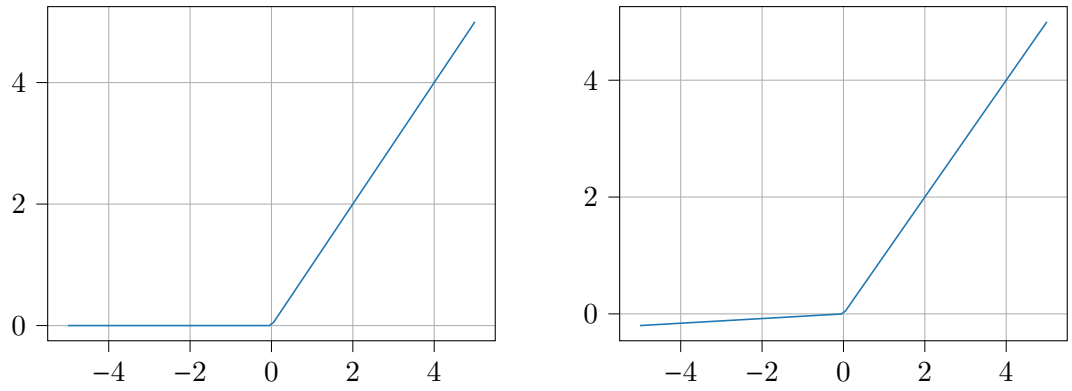
$$\sigma(x) = \max(0, x). \tag{3.3}$$

Although the gradient is still 0 for all negative inputs, it does not vanish for large x (see Figure 3.3a). If $x = 0$, the gradient is undefined, but in practice, this is very rarely the case. Nonetheless, a value for the unlikely case it happens should be selected, e.g. 0.

The problem for negative inputs can be fixed with leaky ReLUs:

$$\sigma(x) = \begin{cases} x & \text{for } x \geq 0, \\ \varepsilon x & \text{for } x < 0, \end{cases} \tag{3.4}$$

where ε is some small value e.g. $\varepsilon = 0.01$. Figure 3.3b shows this function for $\varepsilon = 0.04$. Alternatively, finding ε can also be integrated into the learning process and alongside the weights.



(a) ReLU activation function. **(b)** Leaky ReLU activation function for $\varepsilon = 0.04$.

Figure 3.3: Rectifier activation functions.

3.3 Optimizers

The most important factor for creating well-performing neural networks are the parameters W. During training, we essentially solve an optimization problem to find them. The set of optimal parameters is characterized by minimizing the loss for the given training data. Unsurprisingly, the choice of the optimizer is heavily contributing to the result, and the invention of new optimization strategies plays a big role in the recent successes of neural networks. Based on previous work by Ruder [Rud16], this section introduces the most common types of optimizers and assesses their strengths and weaknesses.

All of the presented methods work in the same fashion: They begin with some initial parameter estimate $W^{(0)}$ and update this estimate according to some rule $W^{(t+1)} \leftarrow W^{(t)} + \eta^{(t)}\vec{s}$. Their main difference is the choice of the search direction $\vec{s}$ and learning-rate (or step-size) $\eta^{(t)}$ for step (t). The parameters are updated until either a maximum number of iterations is reached, or some convergence criterion is fulfilled. We consider only first-order optimization schemes, meaning that we do not utilize higher-order derivative information of our objective function and instead only use the Jacobian.

3.3.1 Gradient Descent

Gradient descent (or steepest descent) is probably the simplest and most well-known optimization method. The search direction $\vec{s}$ is just the negative gradient of the loss function w.r.t. the parameters, resulting in the following update rule

$$W^{(t+1)} \leftarrow W^{(t)} - \eta \frac{dL}{dW^{(t)}}. \tag{3.5}$$

Note that the learning-rate is independent of the step (t) since gradient descent usually uses a constant step-size. There are however implementations that use a decaying learning-rate.

For neural networks, one has to differentiate between three variants of gradient descent. They differ in the number of data points they use per update step: **Batch gradient descent** accumulates the losses for the entire data set and then performs one (large) step of gradient descent. Depending on

the input size, this might become a problem when the data set can no longer fit into memory. For the problems considered in this thesis, the data sets typically have between 500 and 1500 samples, hence this is not an issue. Batch gradient descent converges to a local minimum, which might be the global minimum of the objective function. In comparison, **stochastic gradient descent** only uses the gradient of one data point at a time. This is a lot faster to calculate but may lead to high fluctuations in the objective function. While these fluctuations are usually undesirable, this also allows the optimizer to jump out of the current valley to other, possibly better local minima [Rud16]. **Mini-batch gradient descent**[2] is a middle ground between the other two. By allowing the user to define the batch size, he can ensure the batches still fit into memory while maintaining the convergence stability of batch gradient descent. This is the most general formulation, i.e. the other two types of gradient descent can be considered special cases with a mini-batch size equal to the number of samples and mini-batch size 1. Additionally, a lot of parallel and highly efficient implementations (e.g. Hogwild [RRWN11] and Downpour SGD [DCM+12]) of mini-batch gradient descent exist, making it the usually favored method. In the following we therefore always consider mini-batch gradients, unless specified otherwise.

No matter which version of gradient descent is chosen, they all have two issues in common:

1. Finding a good learning-rate η is not trivial. Bad choices can either lead to slow convergence or even cause the optimizer to diverge. Additionally, a suitable learning-rate may not even exist, unless it can be chosen separately for all parameters.
2. If the objective function is non-convex and has multiple local minima or saddle points, gradient descent likely converges to one of those instead of the global optimum. According to Dauphin et al. [DPG+14], especially saddle points are a big problem and notoriously hard to escape.

Resolving these issues is one of the key challenges when developing new optimization algorithms for neural networks.

3.3.2 Adagrad

The idea of Adagrad [DHS11] is to address the first problem by employing adaptive and parameter-dependent learning-rates. If a parameter experienced large updates in the past optimization steps, it should receive smaller updates than a parameter which was not updated as much. To implement this, we sum up the squares of the gradients for each individual parameter. Let $t+1$ be the current optimization step, then the sum of all squared gradients $g_i^{(t)}$ for parameter i is given by

$$g_i^{(t+1)} := g_i^{(t)} + \left(\frac{dL}{dW_i^{(t)}} \right)^2, \quad \text{with} \quad g_i^{(0)} = 0. \tag{3.6}$$

The new learning-rate is anti-proportional to the square root of this value, meaning for individual parameters the new update rule is given by

$$W_i^{(t+1)} \leftarrow W_i^{(t)} - \frac{\eta}{\sqrt{g_i^{(t)} + \epsilon}} \frac{dL}{dW_i^{(t)}}, \tag{3.7}$$

[2] In literature mini-batch descent is also commonly referred to as stochastic gradient descent.

where ϵ is a small smoothing term (e.g. $\epsilon = 10^{-8}$) [Rud16]. In order to formulate an update rule for all parameters, we arrange the individual learning-rates in a diagonal matrix $G^{(t)} := diag\left(\frac{\eta}{\sqrt{g_0^{(t)}+\epsilon}}, \ldots, \frac{\eta}{\sqrt{g_n^{(t)}+\epsilon}}\right)$, thus the new update rule is

$$W^{(t+1)} \leftarrow W^{(t)} - G^{(t)} \cdot \frac{dL}{dW^{(t)}}. \tag{3.8}$$

A common choice for the global learning-rate is $\eta = 0.01$.

3.3.3 Adadelta

The major problem of Adagrad is the diminishing learning-rate if the accumulated gradients are large. This entails that Adagrad basically stops learning as soon as the values in the matrix $G^{(t)}$ become too large. Adadelta's [Zei12] goal is to prevent this from happening while retaining the parameter adaptive learning-rate. Instead of using the uniformly weighted sum over all the squared past gradients, Adadelta applies an exponential weight decay for older gradients:

$$\tilde{g}_i^{(t+1)} := \gamma \tilde{g}_i^{(t)} + (1-\gamma)\left(\frac{dL}{dW_i^{(t)}}\right)^2, \text{ with } \tilde{g}_i^{(0)} = 0. \tag{3.9}$$

The suggested value for the weight decay factor γ is 0.9.

In addition, Zeiler [Zei12] mentions the need for a global learning-rate η as another drawback of Adagrad. In order to eliminate it from Equation (3.7), it is replaced by the sum of squared parameter updates, with the same exponential weight decay applied

$$h_i^{(t+1)} := \gamma h_i^{(t)} + (1-\gamma)(W_i^{(t+1)} - W_i^{(t)})^2, \text{ with } h_i^{(0)} = 0. \tag{3.10}$$

Analogously to Adagrad, we define two matrices

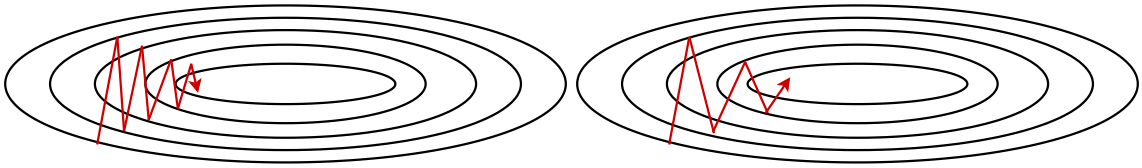
$$\tilde{G}^{(t)} := diag\left(\frac{1}{\sqrt{\tilde{g}_0^{(t)}+\epsilon}}, \ldots, \frac{1}{\sqrt{\tilde{g}_n^{(t)}+\epsilon}}\right) \text{ and } H^{(t)} := diag\left(\sqrt{h_0^{(t)}+\epsilon}, \ldots, \sqrt{h_n^{(t)}+\epsilon}\right), \tag{3.11}$$

and combine them in the following update rule

$$W^{(t+1)} \leftarrow W^{(t)} - H^{(t)}\tilde{G}^{(t)} \cdot \frac{dL}{dW^{(t)}}. \tag{3.12}$$

3.3.4 ADAM

ADAM [KB15] (Adaptive Moment estimation) uses a concept called *momentum* to further improve the convergence of the optimization. **Momentum** is conceived by observing a common problem in Gradient descent based optimizers. Take a look at Figure 3.4a. When navigating ravines with gradients much steeper in one dimension than the others, Gradient descent oscillates between slopes



(a) Without moment. **(b)** With moment.

Figure 3.4: Schematic depiction of gradient descent optimization with and without utilizing momentum.

while only slowly converging towards the minimum. Momentum aims to solve this by adding a fraction $\rho_1 \in [0, 1)$ of the updates performed in the previous steps to the current one

$$\begin{aligned} m^{(t)} &:= \rho_1 m^{(t-1)} + (1 - \rho_1)\frac{dL}{dW} \\ W^{(t+1)} &\leftarrow W^{(t)} - m^{(t)}. \end{aligned} \tag{3.13}$$

The result is an acceleration of the optimization towards the bottom of the ravine and thus the minimum.

In addition to the first moment in Equation (3.13), ADAM uses the second moment

$$v^{(t)} := \rho_2 v^{(t-1)} + (1 - \rho_2)\frac{dL}{dW} \odot \frac{dL}{dW}. \tag{3.14}$$

The parameters $\rho_1, \rho_2 \in [0, 1)$ control the exponential decay of the moments and are usually close to 1. Kingma and Ba [KB15] suggest values of $\rho_1 = 0.9$ and $\rho_2 = 0.999$. Moreover, the moments are initially zero-vectors $m^{(0)} = \vec{0},\ v^{(0)} = \vec{0}$ which leads to a bias towards $\vec{0}$, especially during the first few steps. In order to counteract this, we use the bias corrected moments

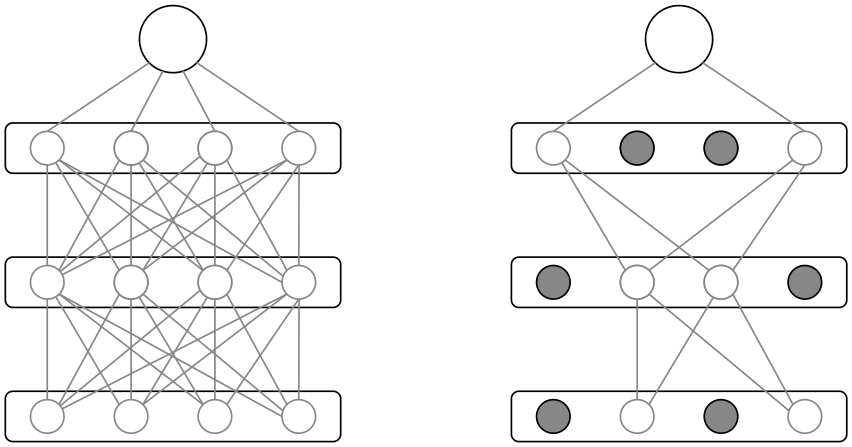
$$\begin{aligned} \hat{m}^{(t)} &:= \frac{1}{1 - \rho_1^t} m^{(t)}, \\ \hat{v}^{(t)} &:= \frac{1}{1 - \rho_2^t} v^{(t)}. \end{aligned} \tag{3.15}$$

To be clear: ρ_1^t is the t-th power of ρ_1 and not some step specific constant (the same holds for ρ_2^t). The ADAM update rule is then

$$W^{(t)} \leftarrow W^{(t-1)} - \eta \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}. \tag{3.16}$$

The authors suggest a global learning-rate of $\eta = 0.01$ and a smoothing value of $\epsilon = 10^{-8}$.

Empirical evidence [KB15] shows that ADAM outperforms other optimizers in terms of convergence speed and quality of the found solution in many cases. In general, adaptive learning-rate methods seem to be well suited for sparse input data sets, such as the ones we consider in Chapter 5 [KB15; Rud16].



(a) Fully connected neural network.

(b) Neural network with dropout of 0.5 for each layer.

Figure 3.5: A 2 hidden-layer neural network with and without Dropout applied.

3.4 Regularizations

Just like regression, neural networks may suffer from over-fitting, meaning that, after training, the performance is good on the training data set, but poor on test data due to a lack of generalization. This section presents different techniques and extensions to prevent this.

3.4.1 Dropout

Dropout [SHK+14] counteracts over-fitting by randomly deactivating some neurons during each training step (batch). Figure 3.5a shows a fully connected neural network with 4 layers and 4 neurons for the input as well as each hidden layer. Applying a Dropout of $p = 0.5$ to a 4 neuron layer means choosing 2 neurons uniformly at random and disabling them, as shown by the black neurons in Figure 3.5b. [3] This includes removing all their inputs and outputs, which is equivalent to setting the corresponding entries of the matrix W_l to zero. Reducing the number of neurons leads to larger updates to the weights of the neurons. During testing, i.e. after the training has finished and the neural network is used for prediction, all neurons are enabled, hence we have to compensate for that by scaling all weights with $(1 - p)$. This way we have the same expected output value during test and training time [SHK+14]. Training multiple models and subsequent combination of them to get a better model improves the results for many machine learning techniques. But for neural networks, it was too impractical and compute intensive to train multiple networks. Dropout can be understood as a way of realizing this while still making training efficient and feasible.

[3] Note that in the original paper [SHK+14] the probability p is defined as the probability to retain a neuron, whereas we define it as the share of disabled neurons in a layer, which is consistent with the definition used in Keras [Cho+15].

3.4.2 Shuffling

The training data for our neural network is the output of some parser (see Section 4.2). It is therefore likely that it has some kind of inherent order, for example, the samples may be sorted alpha-numerically by the name of the log files. The order of the samples affects the gradients used during optimization (and thus the result) because it changes which samples are put together into batches. Hence it is a good idea, to shuffle the data set before using it as training data. An even better idea is to shuffle the data after every epoch for three reasons:

- Different batch gradients during each epoch allow the optimizer to escape local minima and jump to new potentially better ones
- We use batches under the assumption that their gradients serve as a good approximation of the global gradient (over all samples). It is possible that some of the batches have "bad" (unrepresentative) gradients, which may ruin the optimization results. This becomes much less probable when shuffling after each epoch.
- As described in Section 3.3 some of the optimizers use adaptive learning-rates based on the previous steps. If we use the same mini-batches, the points will be biased by their ordering. For instance, in Adadelta the most recent update has the largest effect on the $\tilde{g}$ and h values and therefore on the current step.

The only exception is if we have some kind of expert knowledge about the data which makes training in a certain order better.

3.4.3 Batch Normalization

As mentioned in Section 3.1, a good weight initialization is choosing them $\sim U(-a, a)$ with $a = \sqrt{\frac{6}{n_l + n_{l+1}}}$. The reasoning is that, for this choice, the output variance is equal to 1 (assuming the input variance is equal to 1) which facilitates learning [GB10]. Unfortunately, this property is lost more and more after each update step, a phenomenon to which Ioffe and Szegedy [IS15] refer to as *internal covariance shift*. Batch normalization [IS15] prevents this by applying a transformation to all outputs, such that the mean is 0 and the variance is 1. Evaluation of this method has shown that using it

- allows for higher learning-rates without causing the optimization to become unstable,
- makes the network less sensitive to weight initialization,
- serves as additional regularization [IS15].

Batch normalization must not be confused with normalization of the input data. Input normalization refers to scaling and shifting of the input parameters such that their values are in the interval $[0, 1]$.

4 Software

The previous chapters all focused on theoretical aspects of simulations, load-balancing, and machine learning, while this one is about the software for a practical realization. We will first take a look at tools for running the coupled simulations to create the necessary run-time measurements. After that, an implementation is presented which solves the load-balancing problem by using this data.

4.1 Tools

Realizing a coupled multi-physics simulation requires sophisticated tools. The two most important aspects are the solvers and coupling library. For our purposes, we use the coupling library *preCICE* [1], as well as the single-physics solver *Ateles*. Additionally, we may need tools to post-process and visualize the results. But since we are not interested in the actual results and only care about the run-time of the solvers, they are not presented here.

4.1.1 preCICE

One of the main advantages of partitioned coupling is the reuse of already existing solvers. This reduces development costs and allows for easy migration to different solvers if needed. Of course, this flexibility comes at a price; we need a way of combining multiple single-physics solvers to one multi-physics solver. Fortunately, there is *preCICE* [BLG+16] which is a library developed exactly for this cause.

Figure 4.1 shows an overview of how *preCICE* works. It provides three important functionalities for realizing a coupled multi-physics simulation:

1. **Inter-solver communication**: In order to exchange boundary information, the participants of a coupled simulation need to be able to communicate with each other. *preCICE* creates the necessary communication channels using either MPI ports or TCP/IP sockets. Channels are created between individual ranks of the different solvers on a point-to-point basis without a central server. However, one process per solver is tasked with steering the simulation [Cho17].
2. **Calculation of data mappings:** *preCICE* supports all three of the mappings described in Section 1.1.2 in their consistent and conservative form.
3. **Implementation of coupling schemes:** *preCICE* supports all four of the coupling schemes mentioned in Section 1.1.1 (explicit-serial, explicit-parallel, implicit-serial and implicit-parallel).

[1] *preCICE*: http://www.precice.org/

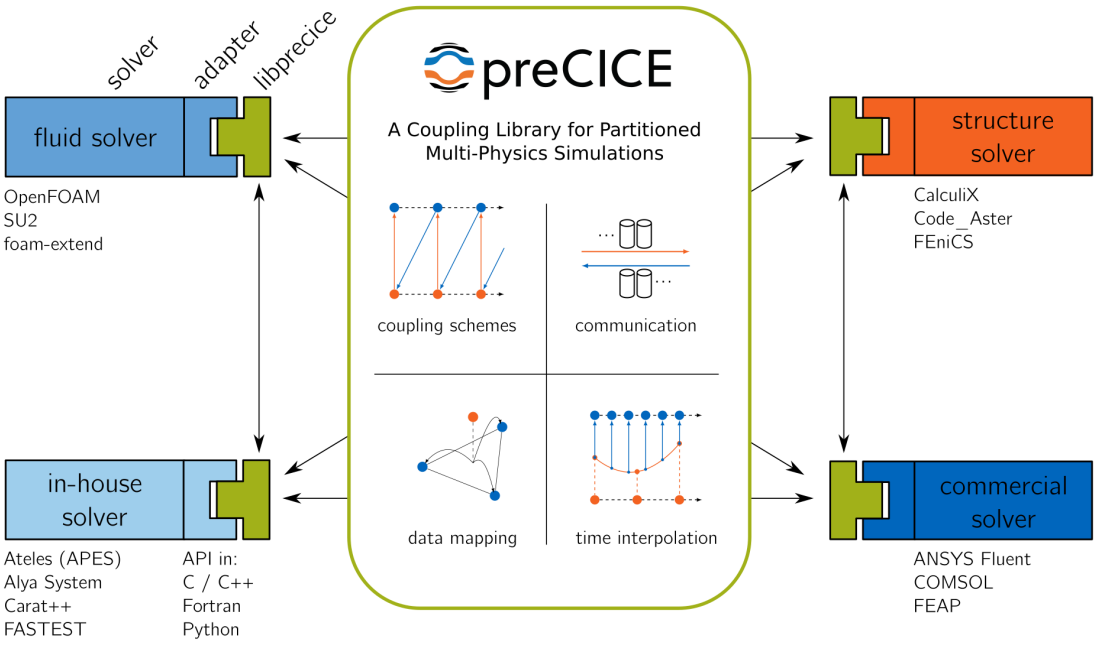


Figure 4.1: Overview of *preCICE*'s functionalities and solver interaction. (Source: http://precice.org.)

The individual solvers are treated as black-boxes by *preCICE*, ignoring their internal state and only manipulating the inputs processing the output. *preCICE* is a library, meaning that it does not call the solvers but the solvers call *preCICE*. It is, therefore, necessary to create a so-called *adapter* for each solver, which serves as an interface between the solver and *preCICE*. Via the adapter *preCICE* has access to the solver's data and control elements [Cho17].

To run a simulation using *preCICE*, the user must provide an XML-configuration-file. This file is used to define the involved participants, as well as configuring certain aspects of the simulation. This includes selecting a coupling scheme, a data mapping method, and the communication method.

4.1.2 Ateles

Ateles is part of the APES (Adaptable Poly-Engineering Simulator) framework, which is developed at the chair of Simulation Techniques and Scientific Computing of the University of Siegen. The framework is developed as a solution for performing mesh-based simulations on supercomputers and therefore needs to be highly scalable. It is built around the TreElM library [KHZR12] for octree meshes. *Ateles* is the frameworks acoustics far-field solver and uses the modal Discontinuous Galerkin Method for discretization. It supports solving of inviscid and linearized Euler, as well as compressible Navier-Stokes equations. Due to its high scalability and compatibility with *preCICE*, it is the ideal solver for the validation of the proposed load-balancing method.

4.1.3 Seeder

Just like *Ateles*, *Seeder* is part of the APES suite. Its function is to generate the meshes, which then serve as input to the solvers (e.g. *Ateles*). For our purpose, this is necessary when we want to vary the input size of the problem. This allows us to quickly generate samples by running simulations for different numbers of elements.

4.2 Implementation

The previous section explained the tools for running simulations. They allow us to generate the necessary data for the pipeline in Figure 4.2. For the remaining steps (orange box) a Python

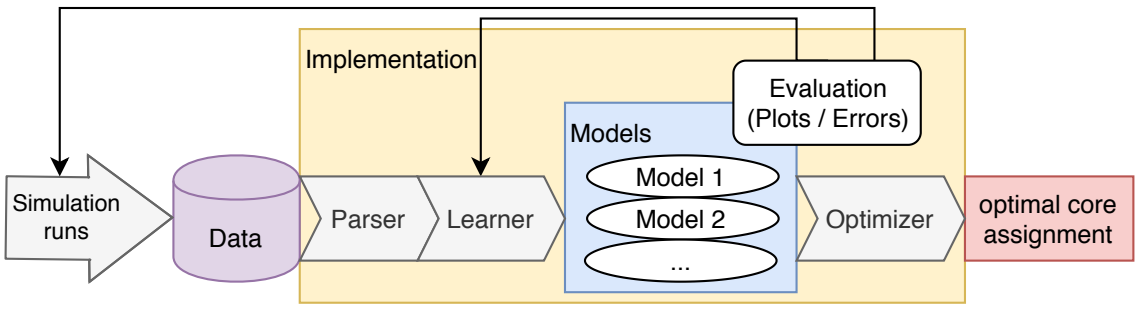


Figure 4.2: Pipeline for finding optimal core assignments.

implementation was developed. This section presents this implementation, alongside the choices made during the development process.

4.2.1 Overview

Python 3 [2] was used as programming language for several reasons:

- **Rapid-prototyping:** Pythons simple syntax, and the fact that it does not need to be compiled, ensure that changes and new ideas can be realized quickly.
- **Library support:** As explained in the previous chapters, a wide range of different fields are combined in our program, for example, combinatorics, machine learning and visualization. Python libraries already provide a lot of the needed functionalities in an easily accessible way.
- **Portability:** Sometimes it might be desired to run the program on the same computer as the simulation since this eliminates the need to transfer the log files. Python distributions are readily available for most operating systems, therefore this is no problem in most cases.

[2] Python: http://www.python.org/

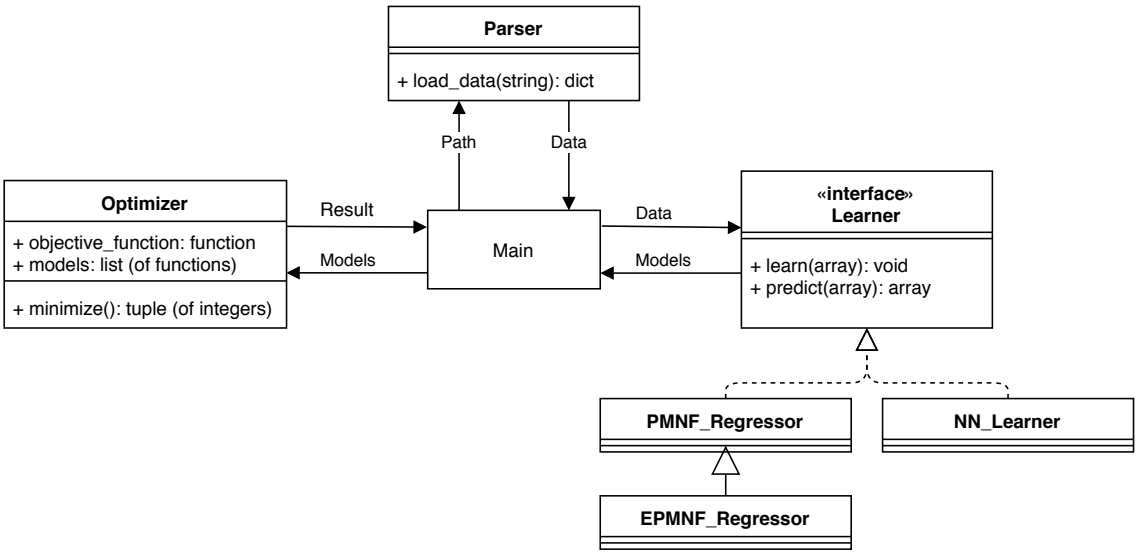


Figure 4.3: Simplified UML-Diagram of the application.

Figure 4.3 shows a simplified UML-Diagram of the applications class structure. It is apparent that the design of the application maps the steps of the pipeline to classes, which are connected by the *Main* class. Thus the structure of the program is easily understandable and maintenance is unproblematic. Also, the tool is extensible, for example, adding new kinds of learners, optimization criteria, or parsers for different file types is possible.

4.2.2 Parser

Loading the data into the program is done by a parser. Its job is to read the output files created by *preCICE* or the solvers, extract the needed values, and convert them into a format that is usable by the rest of the program. The parser supports five different file types: *preCICE* event csv files, *preCICE* event json files, *Ateles* timeinfo files, *Seeder* mesh header files, and generic *.ini* files. The first three of those can be used to find information about the run-time and the number of cores per solver, the mesh header files contain information about the problem size and the *.ini* files can be used to add custom variables. The output of the parser is an array where each row represents one data point. A data point consists of the parameters and the corresponding run-time.

preCICE Event Files

Performing a simulation with *preCICE* produces log files with information about the simulation-run. One such log file is created for each domain of the simulation. Most importantly, they contain details about certain events alongside a timestamp and the MPI-rank. For all relevant steps during the execution such an event is triggered, including the handoff to the solver code and the reentrance to the *preCICE* code, as well as the beginning and end of the mapping calculation. This allows us to calculate the solve time for each time step for each rank by summing up the time needed for solving

Listing 4.1 Defining hardware properties using a params.ini file.

```
[DEFAULT]
clock_rate=2600
cores_per_node=28
interconnect_bandwidth=5100
```

and mapping. The median value of these run-times, together with the core count p (and possibly additional parameters) serves as one data point for this solver. To find p, we simply take the highest observed rank number and add 1.

Ateles Timeinfo Files

There may be circumstances under which the *preCICE* event files are not available, for example, when running non-coupled simulations in order to generate data for testing a learner. If the solver is *Ateles*, we can analyze its output files instead. It generates a timeinfo file containing a table with minimum, maximum and average run-time for various steps during the simulation. This includes the time spent solving (referred to as *simLoop*).

Seeder Mesh Header Files

Part of the generated mesh by *Seeder* is a header.lua file. It contains various details about the generated mesh, including the number of elements, i.e. the mesh size. All we have to do is read the appropriate line beginning with *nElems*.

Generic ini Files

An *ini* file is a simple text file consisting of one or more sections. In Listing 4.1 the only section is the default section. Each section defines multiple parameters and assigns values to them. This can be used to add variables, which should be included in the learning process, but are not part of any other file, e.g. hardware properties. To parse these files the Python library *configparser* is used.

4.2.3 Optimizer

Implementing the optimizer, according to the description in Section 1.3, is straight forward. In order to entirely define an optimization problem the user needs to provide:

- The one-dimensional functions $f_1, \ldots, f_N$ to compose the target function.
- The coupling scheme which defines the target function F.
- An upper bound for the total number of cores P.

Listing 4.2 Creating compositions of P into N parts using Python.

```
def compositions(self, P, N, parent=tuple()):
    """
    Generator for all the compositions of P into N parts,
    without zero values in any of the positions.
    E.g. for P=5 N=2: [1,4], [2,3], [3,2], [4,1]
    """
    if N > 1:
        for i in range(1, P):
            if i < N - 1:
                continue
            if P - i not in self.p_range:
                continue
            for x in self.compositions(i, N - 1, parent + (P - i,)):
                yield x
    else:
        if P not in self.p_range:
            return
        yield parent + (P,)
```

Listing 4.3 Creating compositions of all numbers $\leq P$ into N parts using Python.

```
def less_or_equal_compositions(P, N):
    """
    Generator for all compositions of numbers less or equal to P into N parts.
    without zero values in any of the positions.
    E.g. for P=5 N=2: [1,1], [1,2], [1,3], [1,4], [2,1], [2,2], ... [4,1]
    """
    for i in range(N, P + 1):
        for comp in self.compositions(i, N):
            yield comp
```

Additionally, there are two optional parameters: The first allows the user to define a range to restrict the possible number of cores each solver may be assigned. The second one is a flag which, if passed, causes the optimizer to not assume that $F(p_1, \ldots, p_N)$ is monotonically decreasing, this means that it has to check all assignments such that $\sum_i^N p_i \leq P$, instead of $\sum_i^N p_i = P$.

Without the flag, all necessary compositions can be created using the recursive implementation in Listing 4.2. If the flag is passed, we can reuse this implementation and repeatedly call it for all integers in the interval $[N, P]$ to generate all compositions, as shown in Listing 4.3. All left to do is to iterate over all these compositions $(p_1, \ldots, p_N)$, evaluate the function F, and pick the composition with the lowest value $F(p_1, \ldots, p_N)$.

4.2.4 Learner

Learners (PMNF-Regressor, EPMNF-Regressor, and NN-Learner) have to adhere to the *LearnerInterface* which specifies two functions: *predict* and *learn*. The former is only a template function, whereas the latter accepts the data points as parameters, performs some checks, and then calls the actual implementation (*learnImpl*) of the subclass. For the regression learner, this is either the PMNF-Regressor in the one-dimensional case or the EPMNF-Regressor for $d > 1$.

Listing 4.4 Generating all possible PMNF-hypotheses in Python.

```
from itertools import combinations, product
IcrossJ = product(I, J)
candidates = combinations(IcrossJ, r=n)
```

PMNF-Regressor

The PMNF-Regressor implements the, in Section 2.1 presented, method for finding prediction models. It supports user-defined search-spaces and adheres to the introduced notions for n, I and J. As a reminder, the PMNF is given by:

$$f(p) := \sum_{k=1}^{n} c_k \cdot p^{i_k} \cdot \log_2^{j_k}(p). \tag{4.1}$$

Observe that each term of the PMNF can be uniquely identified by an assignment for i_k and j_k. Therefore the cartesian product $I \times J$ creates all possible terms. To generate all hypotheses, we need to find all length n combinations of these terms, without repetitions and ignoring their order. An implementation using the *itertools* Python package is shown in Listing 4.4. We can now iterate this list to find the best model among the candidates. For each hypothesis three steps are necessary:

1. Create the corresponding feature matrix.
2. Perform k-Fold cross-validation by repeatedly training a regression and evaluating it on the remaining data set.
3. Calculate the total loss by summing up over the individual losses.

To train the regression, the scikit library [PVG+11] is used, which supports standard Linear Regression as well as Ridge Regression. The user can choose between the mean squared error and the symmetric mean approximation error as the loss function for the second step.

EPMNF-Regressor

As described in Section 2.2, the multi-dimensional regression is covered by the Extended Performance Model Normal Form. Finding a model can be done with two different approaches; the unoptimized approach, covering the whole search space, and the optimized approach, which only considers models that are combinations of the best single parameter models (see Section 2.2.1). To find out how much the assumption made by the hierarchical search affects the quality of the resulting model, both of them were implemented.

For the EPMNF a term can no longer be defined by a tuple (i_k, j_k), but by two lists of length d, i.e. $i_k \in I^d, j_k \in J^d$, hence we need to create two $n \times d$-matrices for each model. With this observation, composing all possible terms and hypotheses works analogously to the one-dimensional case. We first create all possible terms $I^d \times J^d$ and then generate all models by forming all length n subsets. The corresponding Python code is shown in Listing 4.5. For large search spaces, the list of *candidates* might become really big. Fortunately, Python generates each entry on-the-fly while iterating through the list, thus memory constraints are not an issue.

Listing 4.5 Generating all possible EPMNF-hypotheses in Python.

```
from itertools import combinations, product
Id = product(I, repeat=d)
Jd = product(I, repeat=d)
terms = product(Id, Jd)
candidates = combinations(terms, r=n)
```

Listing 4.6 Generating all combinations of subsets of terms.

```
for regressor in regressor_list:
  # one term is defined by a pair (i,j) this generates all possible terms
  terms = zip(regressor.opt_i_list, regressor.opt_j_list)
  # all possible subsets of terms
  regressor.powerset = powerset(terms)
# all possible combinations of terms for all variables
all_combinations = regressor_list[0].powerset
for regressor in regressor_list[1:]:
  all_combinations = product(all_combinations, regressor.powerset)
```

The hierarchical search approach uses the best single parameter models to derive the best multi-parameter model, therefore we use the previously introduced PMNF-Regressor to find the best single-parameter models in a first step. This provides us with lists of i_k and j_k assignments for each dimension. As explained in Section 2.2.1, we want to create all combinations of subsets of terms, which is implemented by the code in Listing 4.6. To generate all subsets of terms for one dimension, we first create a list of all terms out of the i_k, j_k-lists (line 3), and then compute the powerset (line 5). All that is left to do, is to take the cartesian product over all these powersets (lines 7-9). In the next step, we simply iterate over all *all_combinations*, assemble the corresponding i- and j-matrices, and create the model. Minimizing the loss yields the optimal one.

Modified golden section search (see Section 2.2.1) was not implemented since hierarchical search is sufficient to make EPMNF regression feasible.

Neural Network Learner

To implement the neural network learner, *Keras* [Cho+15] with the TensorFlow [MAP+15] backend is used. *Keras* is an easy to use Python library for neural networks. It functions as an abstraction layer on top of the powerful *TensorFlow* framework providing its capabilities in a more accessible fashion. One of the core concepts is mapping the components of a neural network, such as layers, activation functions, and optimizers to classes and objects. To clarify this, take a look at the example in Listing 4.7. After importing the required libraries (line 1 and 2), a sequential neural network is created (line 3) and populated. There is no need to explicitly define an input layer, therefore the first layer is a hidden layer with 30 neurons and ReLu activation function (line 5). We then apply a Dropout to that layer (line 5) and add another layer with 20 neurons and sigmoid activation (line 6). The last layer is the output, in our case one neuron suffices (line 7). Before a model can be used, it needs to be compiled and the optimizer and loss function defined. The final step is to feed the data into the network (line 9), i.e. training. This example shows that *Keras* already packages a lot of different choices for activation functions, loss functions, and layers. Additionally, it can be extended by custom tailored modules.

Listing 4.7 Example code for composing a Neural Network with 2 hidden layers and Dropout using *Keras*.

```
import tensorflow as tf
from tensorflow.keras import layers
model = tf.keras.Sequential()
model.add(layers.Dense(30, activation='relu'))
model.add(layers.Dropout(0.3))
model.add(layers.Dense(20, activation='sigmoid'))
model.add(layers.Dense(1))
model.compile(optimizer=tf.train.AdamOptimizer(), loss='mse')
model.fit(X, y, epochs=1000, batch_size=len(X), verbose=1)
```

4.2.5 Evaluation

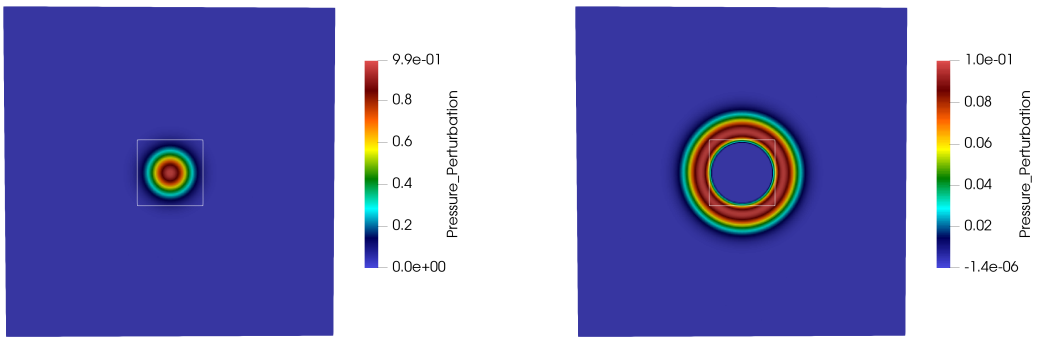
The application supports the evaluation of prediction models through the user in several ways. It is possible to split the input data into training and test data via a user-defined function. This is crucial for the validation of machine learning algorithms because a model might perform very well on the input data points, but generalize poorly to new data. Only the training data is used to create the models and the test data is used for the evaluation. After training, several error metrics and visualizations are available to the user, including 3D and 2D-plots of the model and data points.

5 Validation

This chapter presents several test cases in order to confirm the proposed method works. The main ingredient for successfully applying this method is creating good performance models, hence a second focus is on comparing the two modeling approaches presented in Chapters 2 and 3.

5.1 Load-balancing

The first test case we consider is a gaussian pressure pulse with half-width 0.25 and magnitude 1.0. It begins in the center and spreads over the domain as the simulation progresses, as shown in Figure 5.1. While – strictly speaking – this is not a multi-physics simulation (because there is only one kind of physics involved), it still has all the elements we are looking for. The system is divided into an inner (inside the white box in Figure 5.1a) and an outer domain. Both of them use the discontinuous Galerkin discretization and are solved using *Ateles*. In the inner domain the inviscid Euler equations of scheme order 3 are solved, whereas in the outer domain we consider linearized Euler equations of scheme order 6. The inner domain uses a finer mesh resolution with $262,144$ elements in a $1 \times 1 \times 1$ unit length cube. In comparison, the outer domain spans a cube of $5 \times 5 \times 5$ (including the inner domain), but only has $63,488$ elements. Due to the non-matching meshes, nearest projection mapping is used. Coupling of the equations is performed by *preCICE* using an explicit-parallel scheme, meaning the optimization function is given by $F(\vec{p}) := \max_i f_i(p_i)$ (see Table 1.1). We set the pressure and background pressure for the linearized equations to $100\ kPA$. The density and background density are both set $1.0\ kg/m^3$. In both domains the timestep length is $dt = 10^{-6}\ s$.



(a) Beginning of the simulation $t = 0\ s$.

(b) After $t = 1.2 \cdot 10^{-3}\ s$.

Figure 5.1: Gaussian pressure pulse for different times during the simulation. The white square indicates the domain boundary.[TPS+19]

Number of cores	150	170	200	250	300	330	350
Median run-time [ms]	453	379	305	253	269	196	176

Table 5.1: Inner domain measurements.

Number of cores	74	110	136	142	148	174	210
Median run-time [ms]	418	281	220	229	224	186	140

Table 5.2: Outer domain measurements.

We performed our tests on the supercomputer *SuperMUC* at the Leibniz Supercomputing Centre (Leibniz-Rechenzentrum) [1]. Per node, it uses two *Haswell Xeon E5-2697 v3* processors with 14 cores each. The nominal frequency of each processor is $2.6\ GHz$ and the bisection bandwidth for inter-node communication amounts to $5.1\ TByte/s$.

We run 20 timesteps of the simulation for different core numbers and then parse the produced *preCICE* log files. The median of the solvers' run-times along with the number of cores serves as input for our regression model (see Tables 5.1 and 5.2). [2] By restricting the number of measurements to seven data points, we can evaluate how well the approach works even for few samples. The total number of cores (inner + outer domain) is always a multiple of 28 because there are 28 cores per node on the test machine. The core numbers for the solvers are chosen so that the expected optimization results are within the measurement range.

Feeding them into the regression-learner with the search space

$$n = 2,\ \ I = \{-2, -1.75, -1.5, \ldots, 2.75\},\ \ J = \{-2, -1, \ldots 2\}, \tag{5.1}$$

yields the regressions shown in Figures 5.2 and 5.3. Compared to the examples in Chapter 2, a range of negative values was included in I and J, which allows for a more accurate reflection of the behavior exhibited by the data. Using these regressions, we solve the optimization problem for $P \in \{280, 336, 392, 448, 504, 560\}$. The next step is to re-run the simulation with the optimal core assignments shown in Table 5.3. In order to confirm the quality of our models, we compare the measured run-time for the optimized core assignments (see green points in Figures 5.2 and 5.3) to the model predictions. We find that the deviations are usually small and the models seem to generalize well to new data.

To assess if they can still be enhanced, we study the effect of two possible improvements. First we increase the size of our search space by incrementing n and expanding J:

$$n = 3,\ \ I = \{-2, -1.75, -1.5, \ldots, 2.75\},\ \ J = \{-2, -1, \ldots 5\}.$$

P	280	336	392	448	504	560
Inner	190	228	266	304	342	381
Outer	90	108	126	144	162	179

Table 5.3: Optimization results.

[1] For a full system description see https://www.lrz.de/services/compute/supermuc/systemdescription/Flyer.pdf.
[2] Before taking the median, it should be ensured that the solve-time is approximately the same for each timestep.

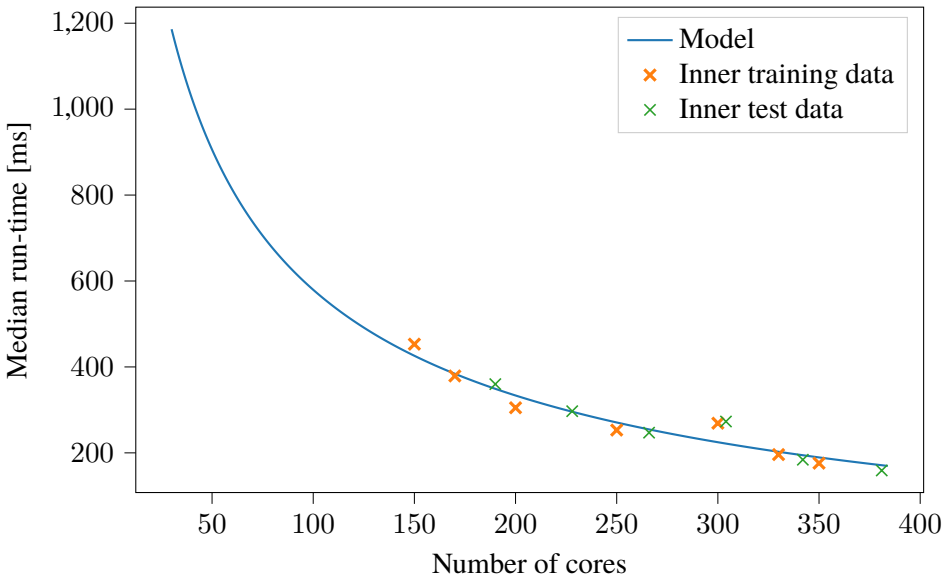


Figure 5.2: Regression and data points for the inner domain.
$i_1 = -1.5,\ i_2 = -1.0,\ j_1 = 2.0,\ j_2 = 2.0$

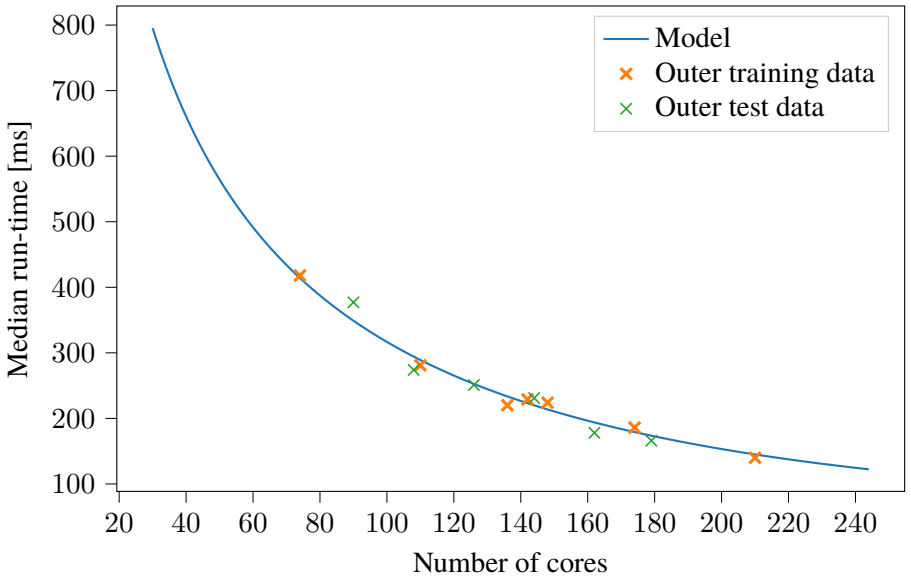


Figure 5.3: Regression and data points for the outer domain.
$i_1 = -0.25,\ i_2 = 0.0,\ j_1 = 2.0,\ j_2 = 1.0$

We choose to expand J because both values (j_1 and j_2) hit the upper bound of the interval in the original model. Second, we can use the new data points as additional inputs and incorporate them into our model. For the inner domain, the three models are shown in Figure 5.4. While there is a notable difference outside the measurement range, all models are very similar in the interval $[150, 380]$. This indicates the model is, in fact, a suitable run-time predictor in this range. However,

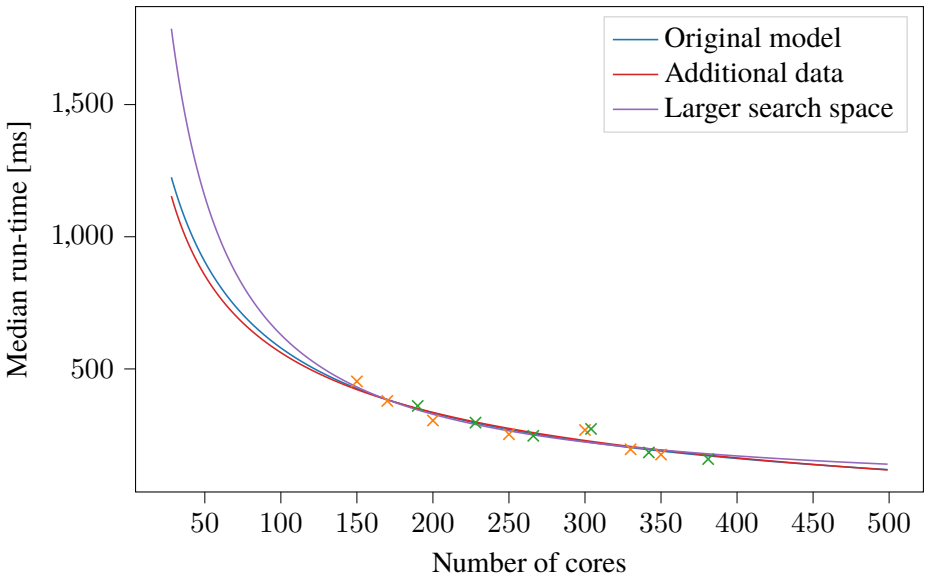


Figure 5.4: Comparison of the original regression model, the model incorporating the new data and the model for the larger search space.

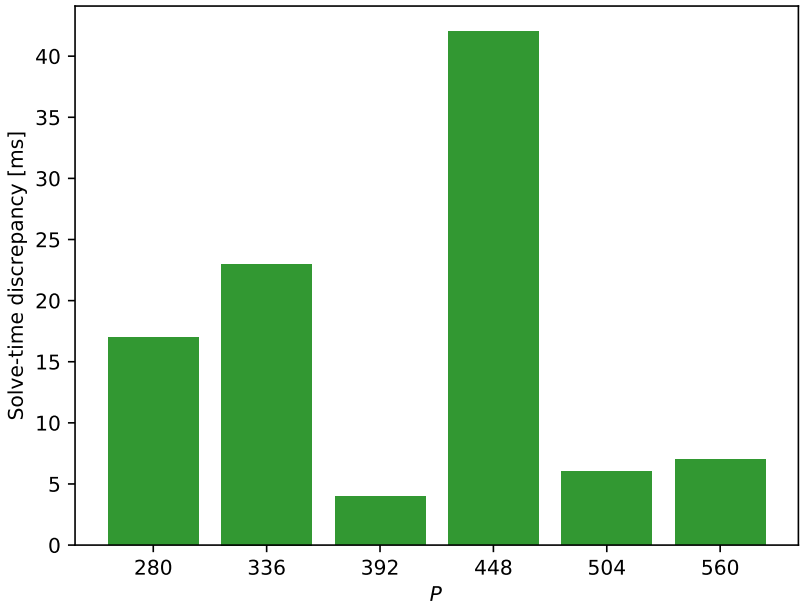


Figure 5.5: Solve-time discrepancies for the optimized core assignments.

small deviations are still possible, which may lead to non-optimal core assignments (±3 cores). But since the actual run-time of the solvers is subject to noise and might fluctuate between timesteps, perfect precision is unachievable anyway.

Whether the load-imbalance was successfully eliminated, becomes apparent by looking at the solve-time discrepancy between the two domains (for optimal core assignments).

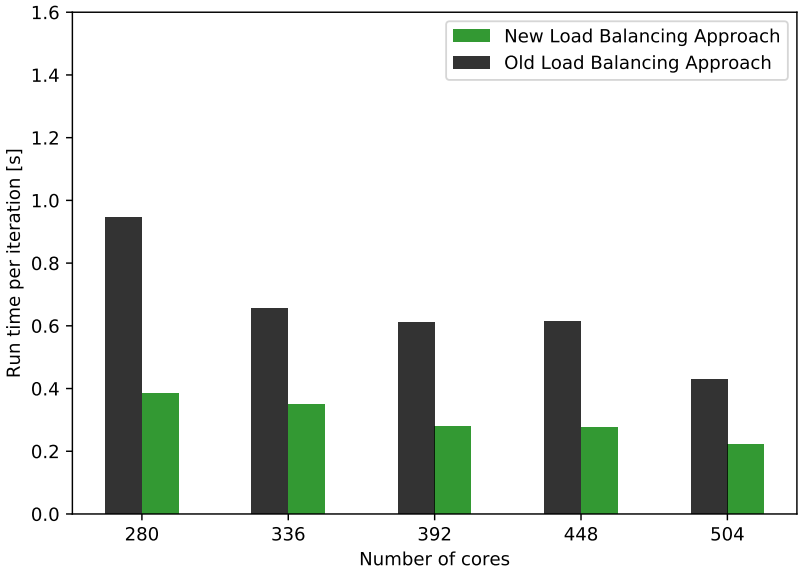


Figure 5.6: Comparison of solve-time per timestep for the old and new load-balancing approach.[TPS+19]

If the run-time difference between the inner and outer domain solvers is large for (supposedly) optimal core assignments, the entire method would be flawed. Fortunately, Figure 5.5 shows that the discrepancy is generally low ($\leq$ 23 ms) for most of the optimized core assignments. One exception is the instance for $P = 448$ with a difference of 42 ms. This sample corresponds to the assignment $p_{\text{inner}} = 304$ and $p_{\text{outer}} = 104$ (see Table 5.3). By examining the regression models in Figures 5.2 and 5.3 the problem becomes apparent: the model for the inner domain underestimates and the outer model overestimates the actual run-time. Especially the prediction for the inner domain is very poor for this particular sample. As there are two very close samples with similar results, random noise is unlikely to be the cause. Additionally, the run-time even increases in comparison to the previous sample with about 40 cores less. This might indicate some unexpected solver behavior, which is hard to distinguish from noise during learning, and incorporating this into a model is difficult.

To unequivocally prove the effectiveness of the method, we can compare the run-times with those of the old load-balancing approach. The old load-balancing approach refers to scaling the number of cores according to the domain's degrees of freedom. Figure 5.6 shows this comparison for the different core numbers. The results are impressive, with about 40% run-time improvement per iteration for all P. We can conclude that the new approach is significantly better than the old one.

In addition, it was tried to use neural networks for creating the run-time models. But the models were either heavily under- or over-fitting, and had unsatisfying results for all (tested) hyper-parameter choices. This can generally be expected for problems with very few data points and small dimensionality, as neural networks are not really suited for that.

5.2 Performance Modeling

The previous section proved that the optimization and general approach are working. During our next test case, we, therefore, focus on the performance modeling aspect. In particular, we compare the neural network and regression approach using the same test case in a monolithic version. Instead of dividing the simulation domain, we consider non-linear Euler equations in the whole cube. Additionally, we reduce the timestep length to $dt = 5 \cdot 10^{-7}$ *s* because we want to test higher mesh resolutions, and may violate the *CFL condition* [CFL28] otherwise. The other parameters and test hardware remain unchanged.

5.2.1 Different Mesh Resolutions

A secondary goal of this thesis is to investigate the quality of performance models when including additional parameters. A variable of particular interest is the number of elements in the grid. We use *Seeder* (Section 4.1.3) to create meshes of different sizes for *Ateles*, and train our models using the core number and discretization level as parameters. The discretization level l is simply the $\log_8$ of the number of elements since the grid structure is octree based. In combination with load-balancing this would, for example, allow us to deduce optimal core assignments for previously untested core assignments.

We generate the data by running the simulation for $l = 3, 4, \ldots, 8$ and $P = 28, 42, 56, \ldots, 560$. Levels $l = 1, 2$ are skipped because the number of elements (8 and 64 respectively) cannot be lower than the number of cores. Higher levels than 8 caused problems for *Ateles*, which is the reason why they are ignored as well. Moreover, *Ateles* crashes for $l = 8$ and low core numbers for an unknown reason, leaving us with 220 total data points.

In a first test, we will split the whole data set into 80% training and 20% test data and then study the quality of the EPMNF-Regression and neural network models. For the EPMNF-Regression, a first task is to compare the results of searching the full search space and the hierarchical search. Unfortunately, the full search space for $d = 2$ and the parameters in Equation (5.1) contains $\binom{(20\cdot5)^2}{2} = 49,995,000$ models. The implementation is capable of evaluating about 225 models per second on the test machine [3], which means the search would take about 62 hours. Instead we shrink the search space to

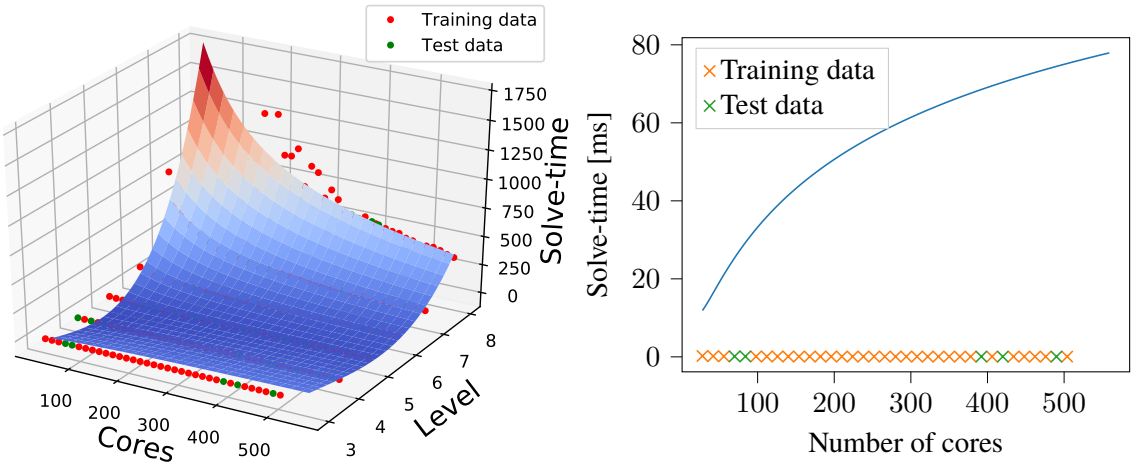
$$n = 2, \;\; I = \{-1, 0.5, \ldots, 1.5\}, \;\; J = \{-1, 0, \ldots 3\}, \tag{5.2}$$

totaling in $\binom{(6\cdot5)^2}{2} = 404,550$ candidates. The generated model, using the mean squared error as loss function, is shown in Figure 5.7a.

While the model may not seem bad at first sight, it performs quite bad for a majority of the training and test data points as Table 5.4 proves. A primary issue is the negligence of data points with small levels. Since the run-time is generally low for those l, the MSE barely penalizes deviations by the model. The result for $l = 3$ is depicted in Figure 5.7b [4]. In contrast, the model seems to

[3] Intel i5-4300U, 8GB RAM

[4] Due to the different magnitudes of regression and data points, it seems as if the solver does not scale, when in fact it does (see Figure 5.13).



(a) Full search. **(b)** Slice for $l = 3$.

Figure 5.7: Plots of the EPMNF model using full search and MSE loss.

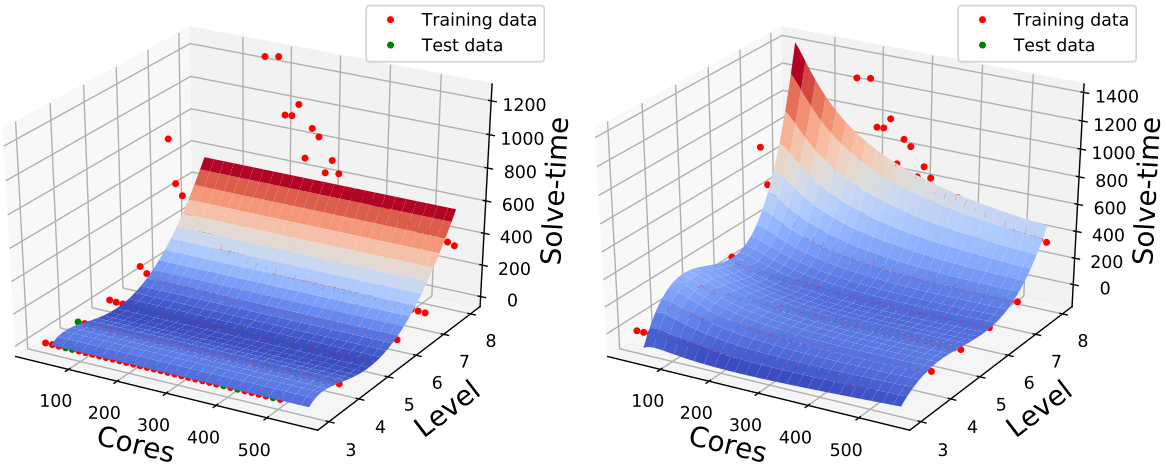
	MSE (training)	SMAPE (training)	MSE (test)	SMAPE (test)
EPMNF (full, MSE-loss)	6997.8	140.2	6696.7	151.7
EPMNF (full, SMAPE-loss)	17343.80	133.7	3866.5	135.9
EPMNF (hierarchical, SMAPE-loss)	8381.5	136.2	6841.4	143.9
Neural Network (MSE-loss)	1721.9	95.63	633.4	81.3
Neural Network (MAPE-loss)	1760.8	9.4	52.1	14.3

Table 5.4: Errors for the different models.

reasonably-well predict run-times for large l. A possible remedy is to use a different loss function such as SMAPE. Albeit the results improve for $l = 3$, the overall model seems to underfit and is almost insensitive to core number changes (Figure 5.8a). This is most likely due to the limited search space.

We, therefore, continue to use the SMAPE loss and run the hierarchical search, using the same search space. Figure 5.8b shows the model and Table 5.4 proves that it is still far from being a good model. Even extending the search space provides no notable improvements.

To find out why it is so difficult to generate a good model using hierarchical search, take a look at the single parameter models created in the first step of the procedure (Figures 5.9 and 5.10). The single variable regressions need to fit all data points at the same time, without any information about the filtered dimension. Ultimately, this leads to choosing a single-parameter model that poorly captures the behavior of the solve-time with regard to the considered parameter (see Figure 5.9). Of



(a) Full search EPMNF model.

(b) Hierarchical search EPMNF model.

Figure 5.8: Comparison of the full EPMNF and hierarchical EPMNF using SMAPE loss.

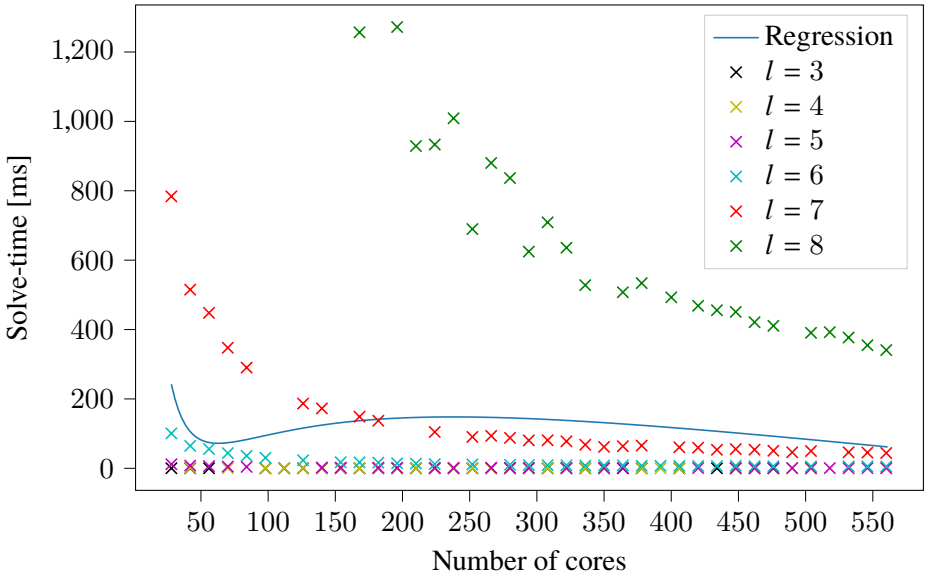


Figure 5.9: Single parameter regression for the core count.

course, we cannot expect to find a good model as a combination of single parameter models if these are unfit. This fundamental flaw in the hierarchical search approach is hard to come by without going back to the full search.

For an initial test of the neural network approach, we consider a network with 2 hidden layers with 30 (ReLU) neurons each and a dropout probability of 0.3 between those. Moreover, we use the MSE as loss function and ADAM as the optimizer. The batch size is set to the number of input data points (176), and the optimization is run for 8000 epochs. Plenty of other hyper-parameter combinations were tested, but these showed the best results. Furthermore, we normalize the input data by shifting and scaling the values for each parameter to the interval $[0, 1]$. The results exhibit the same problem as the first run of the full EPMNF search: the model is under-fitting data points

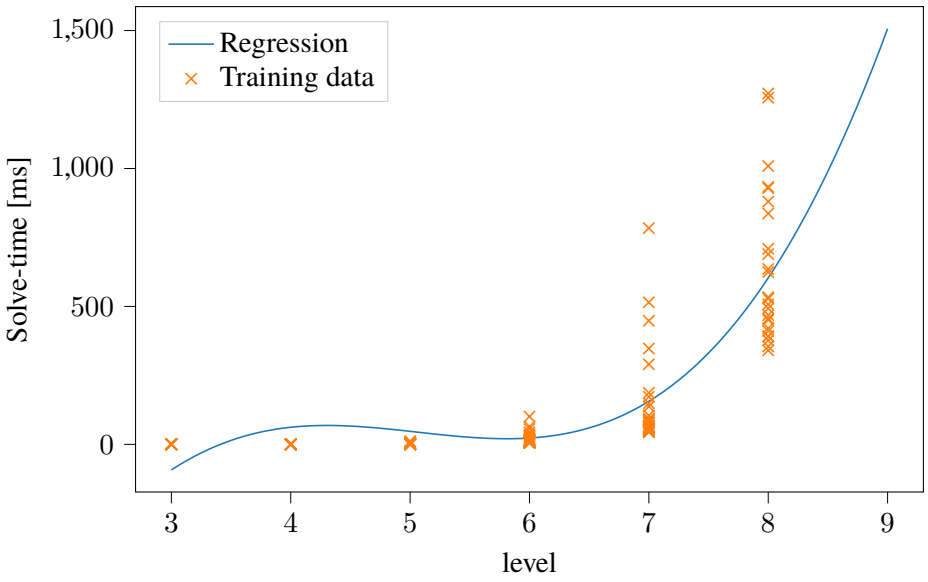


Figure 5.10: Single parameter regression for the level.

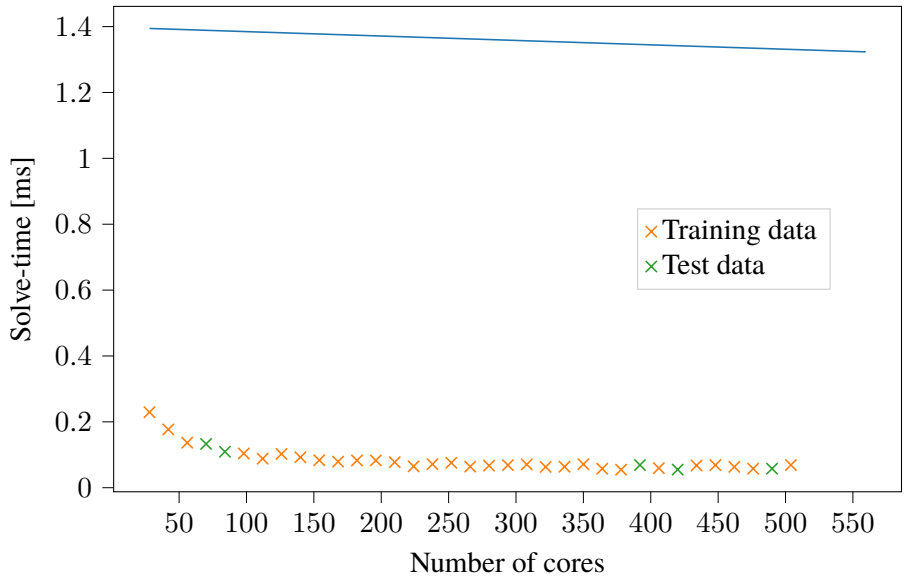


Figure 5.11: Slice for $l = 3$ of the neural network model with MSE loss.

with small l (Figure 5.11), while the overall model seems to be the best one so far considering the errors (Table 5.4). Fortunately, *Keras* implements MAPE as another possible loss function. But using MAPE increases the complexity of the objective function, hence we need to increase the model complexity as well. We raise the number of neurons per layer to 300 and keep the other parameters. The model depicted in Figure 5.12 and errors in Table 5.4 prove, that this is the best modeling approach. Figure 5.13 proves that we successfully fixed the problem of under-fitting for small l.

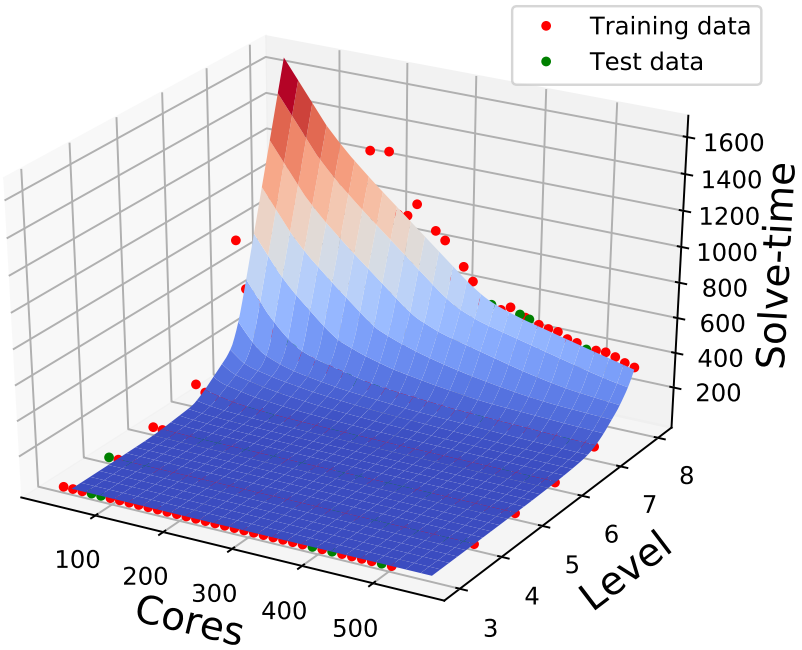


Figure 5.12: Neural network model using MAPE loss.

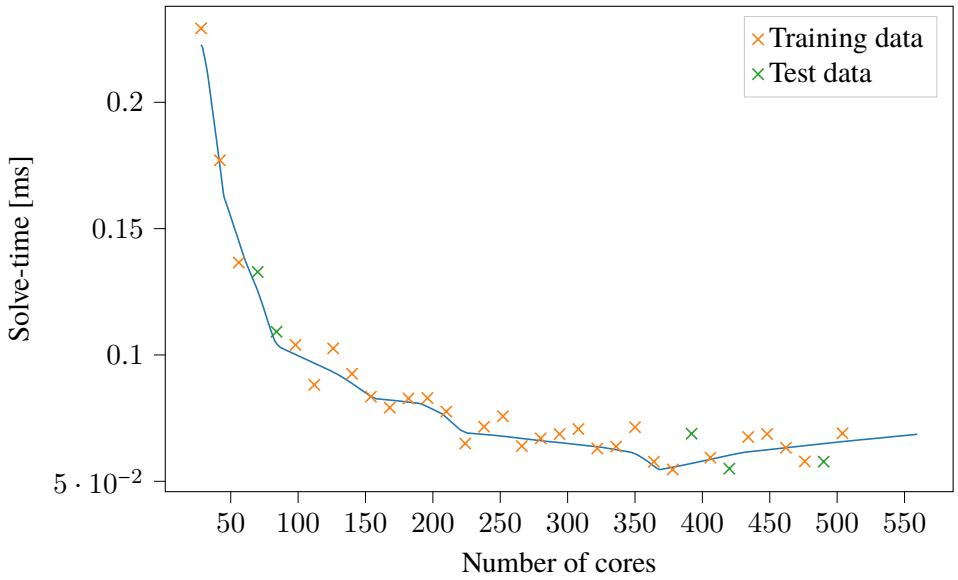


Figure 5.13: Slice for $l = 3$ of the neural network model with MAPE loss.

Fewer Training Samples

In practice we may not always be able to generate a lot of data as quickly as for this test case for two reasons:

- It is possible that even running a few timesteps of the simulation takes a long time, which makes the creation of each data point expensive.

	MSE (test)	SMAPE (test)
EPMNF (hierarchical, SMAPE-loss)	21235.4	150.8
Neural Network (MAPE-loss)	1966.8	11.1

Table 5.5: Test errors for the different models when training on 34 and testing on 186 data points.

- Varying a certain dimension can be difficult, e.g., when accounting for hardware properties, we have to set up the whole simulation on a different machine.

Therefore it is interesting to study, which of the performance analysis approaches better deals with a low amount of input data.

For each level we only use the data points where $p \in \{28, 84, 196, 294, 420, 504\}$ as training data. We end up with 34 total samples (for level 8 the data points $p = 28$ and $p = 84$ do not exist), which is slightly more than the 5^d suggested as lower bound for the EPMNF by Calotoiu et al. [CHPW13]. The remaining data is used for testing. The errors in Table 5.5 show, that the neural network outperforms the EPMNF by a large margin. This is surprising due to the good performance of the regression approach in Section 5.1 and the fact that the EPMNF includes additional "information" by predetermining the basis function.

Predicting the run-time for new levels

Equipped with the insights gained during the previous tests, we now look at the models' generalization properties for previously untested levels. Concretely this means, we use all data points for $l \leq 7$ as training and the rest (i.e. $l = 8$) as test data. Using the same hyper-parameters as in the previous section (SMAPE-loss, 2 hidden layers, 300 neurons each), we rerun the neural network modeling. Consider the slice of the model created for $l = 8$ Figure 5.14. Clearly, the model did not learn the correct dependency between the level and run-time increase, which leads to large errors on the test data (Table 5.6). Hyper-parameter changes (more / less neurons, different activation functions, increase / decrease dropout) did not lead to notable improvements.

One may expect the EPMNF-Regressor to perform better in these cases since it predetermines the dependency as a product of powers and logarithms. Unfortunately, this is not true. Figure 5.15 proves that the single variable regression fails to find the proper dependency, too. As a result, the combined model cannot be good. Because increasing the search space does not yield improved results as well, we need to look for another method.

By applying the $\log_2$ to the run-time of the data points before training and accounting for it later, it is possible to significantly improve the results. Especially, for the neural network approach the predictions become very accurate, as proven by Table 5.6 and Figure 5.16. A likely reason for this is that the neural network has trouble to learn the exponential dependency between the level and the solve-time. The logarithm simplifies this relationship to a linear one, as shown in Figure 5.17. One of the major advantages of neural networks is that no prior knowledge about these dependencies is necessary, but by using the $\log$-improvement we are doing exactly that.

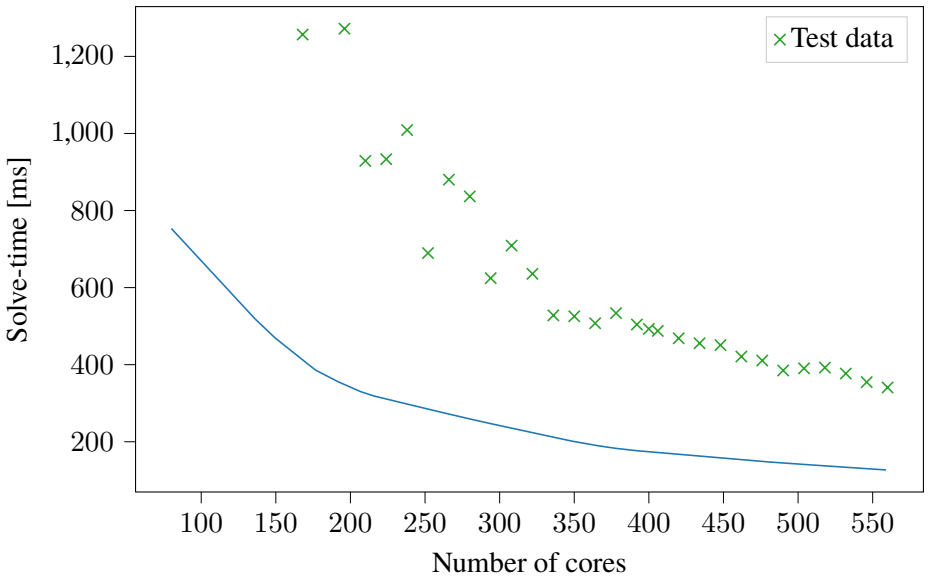


Figure 5.14: Slice for $l = 8$ (test data) of the neural network model with MAPE loss.

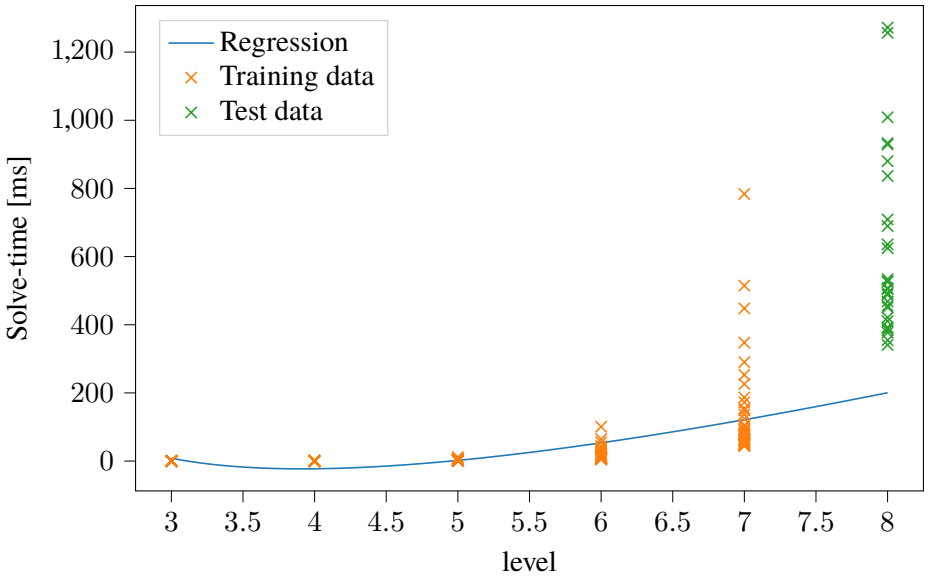


Figure 5.15: Single-parameter regression for the level when using $l = 8$ as training data.

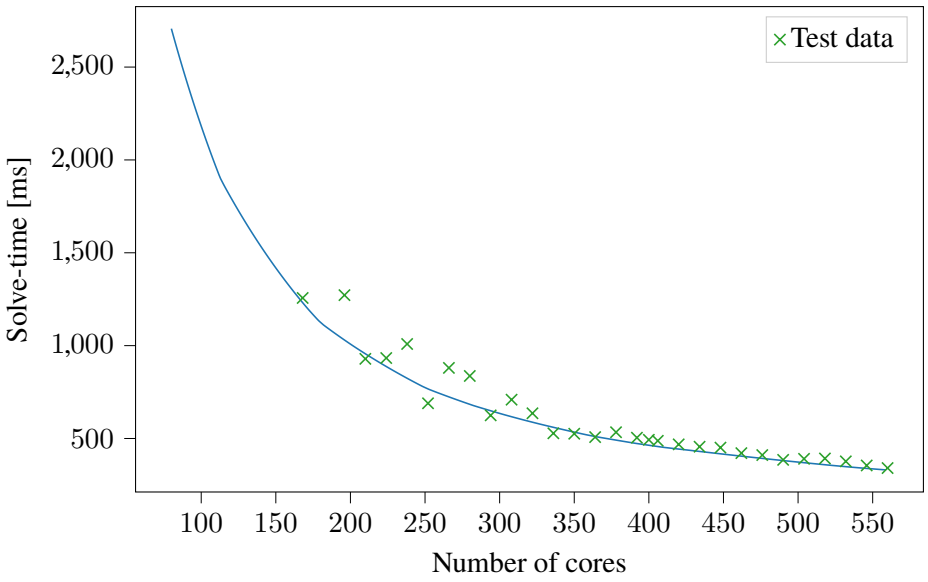


Figure 5.16: Slice for $l = 8$ (test data) of the neural network model using MAPE loss and the log-improvement.

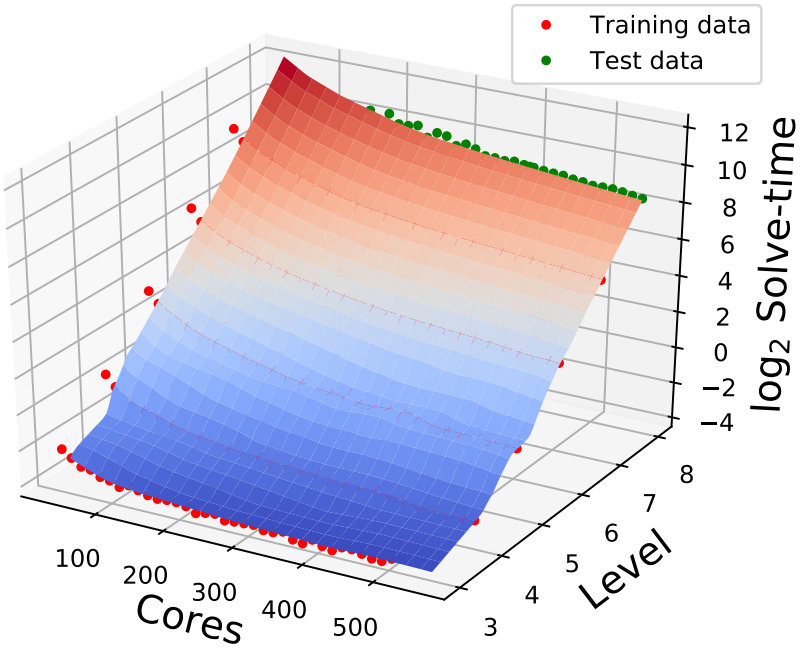


Figure 5.17: Neural network model for the $\log_2$ of the run-times.

	MSE (test)	SMAPE (test)
EPMNF (hierarchical, SMAPE-loss)	342495.8	145.7
EPMNF (hierarchical, SMAPE-loss, log-improvement)	75076.8	38.9
Neural Network (MAPE-loss)	197853.5	96.5
Neural Network (MAPE-loss, log-improvement)	4629.6	6.2

Table 5.6: Test errors for the different models, when using $l = 8$ as test data.

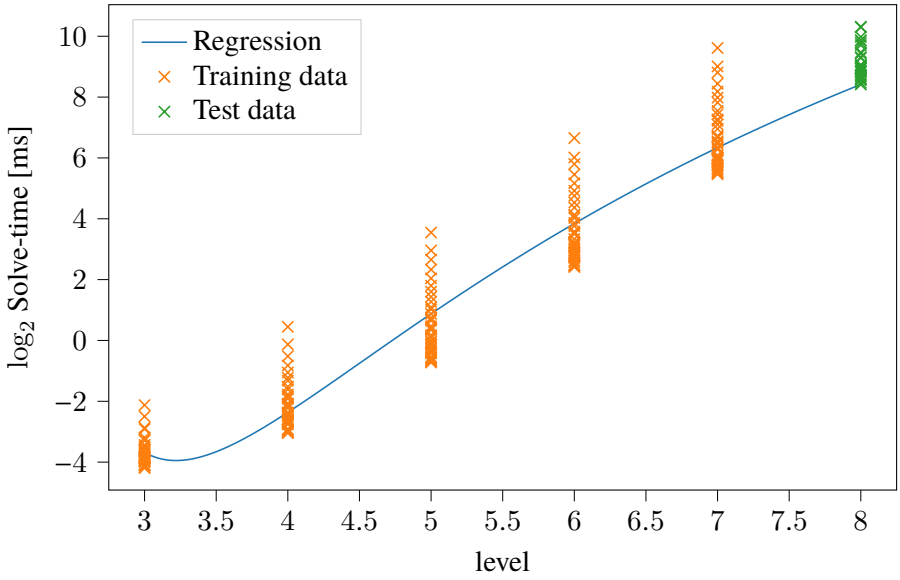


Figure 5.18: Single parameter regression with respect to the level for the hierarchical EPMNF with log-improvement.

Applying the same method, significantly improves the predictions of the hierarchical EPMNF, too. Figure 5.18 shows that this can be attributed to the better single parameter model found for the level variable. Although, the resulting model still underestimates the run-time for all test data points.

5.2.2 Different Hardware Parameters

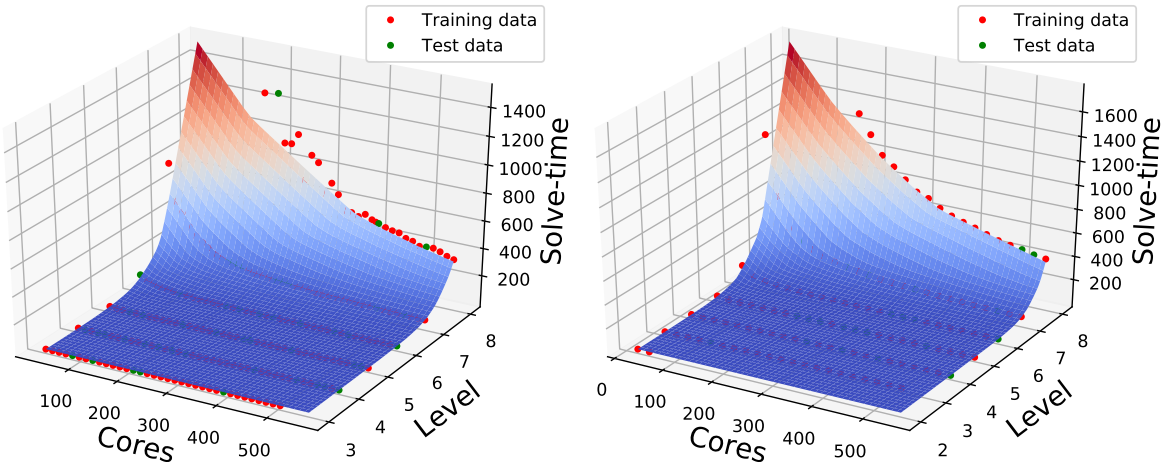
Besides the simulation parameters, hardware properties have a large influence on the run-time. In this test case, we, therefore, consider two different machines: the first is *SuperMUC* as described in Section 5.1. The second is *HazelHen* a supercomputer at the High-Performance Computing Center

Stuttgart (HLRS). Their hardware properties are listed in Table 5.7. Highlighted rows indicate properties serving as additional inputs for the machine learning problem, meaning we end up with four variables (number of cores, level, clock rate and cores per node).

	SuperMUC[5]	**HazelHen**
CPU	Intel Xeon E5-2697 v3	Intel Xeon E5-2680
Clock rate [MHz]	2600	2500
Total number of nodes	3072	7712
Cores per node	28	24
Memory per node [GB]	128	128
Interconnect	Infiniband FDR14	Cray Aries

Table 5.7: Hardware properties of SuperMUC and HazelHen.

For *SuperMUC* we can reuse the samples generated for the previous tests. For *HazelHen* we run the simulation for the same levels and $p = 24, 48, \ldots, 552$. We then train a network using 80% of the 352 total data points (281), with the remaining 71 samples as test data, and the same hyper-parameters as before. Figure 5.19 visualizes these results on a per system basis, i.e. the non-displayed parameters (clock rate and cores per node) were fixed to the respective values of *SuperMUC* and *HazelHen*. The first thing to note is that the *SuperMUC*-model is almost identical to the original model (see



(a) SuperMUC **(b)** HazelHen

Figure 5.19: Plots of the same neural network model for different clock rate and cores per node combinations.

Figure 5.12), showing that the additional parameters are not interfering with already good results. This claim is further supported, by the errors on the test data: MSE: 1799.0; SMAPE: 14.4.

[5] SuperMUC has multiple phases with different hardware properties. The information corresponds to phase 2, which is the one used for our tests.

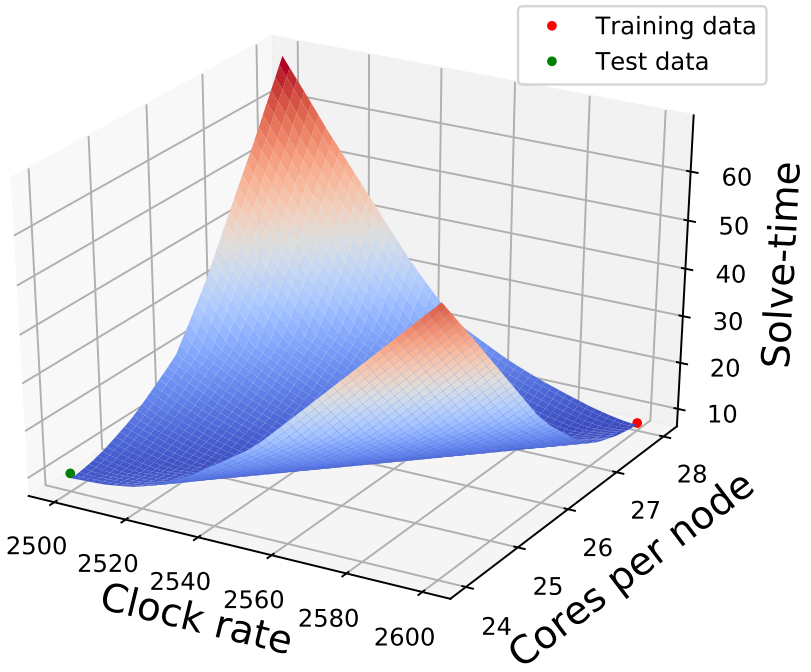


Figure 5.20: Model for $p = 336$ and $l = 6$.

Figure 5.20 shows the solve-time behavior of the model with respect to the newly included parameters. The number of cores is fixed to 336 and the level to 6. We cannot draw any conclusions about the actual dependencies between the solve-time and the parameters for two reasons:

- The differences in hardware properties between the two systems are too large to be reduced to (these) two metrics.
- There are not enough data points for the plotted dimensions. Since we only have two different values for clock rate and cores per node in our data set, the neural network is incapable of learning the dependency between them and the solve-time.

But this proves that we can use the same approach to include hardware parameters in our modeling.

6 Summary & Future Work

This thesis introduced a novel approach for load-balancing of coupled multi-physics simulations. We created a framework to derive optimal core assignments, which consists of two steps. First, we use performance analysis to model the run-time of each solver, depending on the assigned number of cores. Then, the predictions of these models are incorporated into an integer optimization problem. Section 1.3 described a simple method to solve this problem using brute-force. In order to create performance models, two machine learning methods were presented. The first one is based on the work by Calotoiu et al. [CHPW13], and searches a space of candidate regressions and chooses the best-fitting one. The other trains a neural network for the given set of run-time measurements. Both of them were realized in a Python implementation alongside the necessary optimization and evaluation tools.

During the validation, we found that the new approach significantly decreases the solvers' run-times. As expected, the neural network approach is unsuitable for single-parameter models with very low amounts of data, but is generally favorable when considering problems with multiple parameters. Additionally, either SMAPE or MAPE loss should be used when handling data with run-times of different orders of magnitudes.

Despite the primarily positive results, there are some aspects that could be improved. First of all, for very large core numbers and many different solvers involved, the proposed method for solving the optimization problem is infeasible. For example, suppose a simulation consists of $N = 4$ domains and is running on a machine with $P = 10^5$ cores. Even with the simplifying assumption from Section 1.3.1, we need to check approximately 10^{15} different core assignments, hence a more efficient optimization method is needed.

Additionally, the usage of neural networks for performance modeling should be further studied by performing other experiments. This includes enhancing hyper-parameter choices and topology to better generalize to new parameter combinations. Especially, exponential dependencies seem to be a problem as proven by Section 5.2.1. Furthermore, we should examine the predictive qualities of a model for larger numbers of parameters, e.g., hardware properties. In Section 5.2.2 this was attempted, but in the end, the lack of data prevented us from making any meaningful statement. One may also investigate the possibility of using multiple outputs of a neural network to model different performance metrics. This allows answering questions such as, "What is the optimal core assignment while adhering to certain memory constraints?" Apart from improving existing approaches, there may also exist entirely different methods that are better suited for creating run-time models.

Bibliography

[Bis06] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006 (cit. on p. 31).

[BLG+16] H.-J. Bungartz, F. Lindner, B. Gatzhammer, M. Mehl, K. Scheufele, A. Shukaev, B. Uekermann. "preCICE – A fully parallel library for multi-physics surface coupling". In: *Computers and Fluids* 141 (2016). Advances in Fluid-Structure Interaction, pp. 250–258. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2016.04.003. URL: http://www.sciencedirect.com/science/article/pii/S0045793016300974 (cit. on p. 41).

[CBE+16] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, F. Wolf. "Fast multi-parameter performance modeling". In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2016, pp. 172–181 (cit. on pp. 26, 28–30).

[CFL28] R. Courant, K. Friedrichs, H. Lewy. "Über die partiellen Differenzengleichungen der mathematischen Physik". In: *Mathematische annalen* 100.1 (1928), pp. 32–74 (cit. on p. 56).

[Cha19] R. Chan. *Intel and the Department of Energy are building America's first exascale supercomputer, a computer capable of a quintillion calculations per second*. https://www.businessinsider.de/intel-department-of-energy-aurora-first-exascale-supercomputer-2019-3. 2019 (cit. on p. 15).

[Cho+15] F. Chollet et al. *Keras*. https://keras.io. 2015 (cit. on pp. 39, 48).

[Cho17] G. Chourdakis. "A general OpenFOAM adapter for the coupling library preCICE". In: (2017) (cit. on pp. 17–19, 41, 42).

[Cho19] G. Chourdakis. *preCICE Wiki: Tutorial for CHT: Flow over a heated plate*. https://github.com/precice/openfoam-adapter/wiki/Tutorial-for-CHT:-Flow-over-a-heated-plate. 2019 (cit. on pp. 16, 17).

[CHPW13] A. Calotoiu, T. Hoefler, M. Poke, F. Wolf. "Using automated performance modeling to find scalability bugs in complex codes". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM. 2013, p. 45 (cit. on pp. 23, 24, 26, 61, 67).

[CHV13] E. Casoni, G. Houzeaux, M. Vázquez. "Parallel aspects of fluid-structure interaction". In: *Procedia Engineering* 61 (2013), pp. 117–121 (cit. on p. 21).

[DCM+12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. "Large scale distributed deep networks". In: *Advances in neural information processing systems*. 2012, pp. 1223–1231 (cit. on p. 36).

[DHS11] J. Duchi, E. Hazan, Y. Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159 (cit. on p. 36).

[DPG+14] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, Y. Bengio. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *Advances in neural information processing systems*. 2014, pp. 2933–2941 (cit. on p. 36).

[Flo86] B. E. Flores. "A pragmatic view of accuracy measurement in forecasting". In: *Omega* 14.2 (1986), pp. 93–98 (cit. on p. 27).

[Gat14] B. Gatzhammer. "Efficient and flexible partitioned simulation of fluid-structure interactions". PhD thesis. Technische Universität München, 2014 (cit. on p. 17).

[GB10] X. Glorot, Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256 (cit. on pp. 32, 40).

[GBB11] X. Glorot, A. Bordes, Y. Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323 (cit. on p. 34).

[GMH13] A. Graves, A.-r. Mohamed, G. Hinton. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649 (cit. on p. 31).

[HBM03] D. M. Hawkins, S. C. Basak, D. Mills. "Assessing model fit by cross-validation". In: *Journal of chemical information and computer sciences* 43.2 (2003), pp. 579–586 (cit. on p. 26).

[IS15] S. Ioffe, C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015) (cit. on p. 40).

[KAH+01] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, M. Gittings. "Predictive performance and scalability modeling of a large-scale application". In: *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. IEEE. 2001, pp. 39–39. DOI: 10.1145/582034.582071 (cit. on p. 23).

[KB15] D. Kingma, J. Ba. "Adam: a method for stochastic optimization (2014)". In: *arXiv preprint arXiv:1412.6980* 15 (2015) (cit. on pp. 37, 38).

[KHZR12] H. Klimach, M. Hasert, J. Zudrop, S. Roller. "Distributed Octree Mesh Infrastructure for Flow Simulations". In: *European Congress on Computational Methods in Applied Sciences and Engineering* (2012), pp. 1–15 (cit. on p. 42).

[Kie53] J. Kiefer. "Sequential minimax search for a maximum". In: *Proceedings of the American Mathematical Society, 4 (3)*. 1953, pp. 502–506 (cit. on p. 30).

[KSH12] A. Krizhevsky, I. Sutskever, G. E. Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105 (cit. on p. 31).

[MAP+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on p. 48).

[MP43] W. S. McCulloch, W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133 (cit. on p. 31).

[OZW08] M. S. Oyamada, F. Zschornack, F. R. Wagner. "Applying neural networks to performance estimation of embedded software". In: *Journal of Systems Architecture* 54.1-2 (2008), pp. 224–240 (cit. on p. 23).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 47).

[RCSW17] P. Reisert, A. Calotoiu, S. Shudler, F. Wolf. "Following the blind seer–creating better performance models using less information". In: *European Conference on Parallel Processing*. Springer. 2017, pp. 106–118 (cit. on p. 27).

[RRWN11] B. Recht, C. Re, S. Wright, F. Niu. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in neural information processing systems*. 2011, pp. 693–701 (cit. on p. 36).

[Rud16] S. Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747 (cit. on pp. 35–38).

[RZL17] P. Ramachandran, B. Zoph, Q. V. Le. "Searching for activation functions". In: *arXiv preprint arXiv:1710.05941* (2017) (cit. on p. 34).

[SCH+15] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, F. Wolf. "Exascaling your library: Will your implementation meet your expectations?" In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM. 2015, pp. 165–175 (cit. on p. 26).

[SDSM18] E. Strohmaier, J. Dongarra, H. Simon, M. Meuer. *Top500 Supercomputer Sites November 2018*. https://www.top500.org/lists/2018/11/. 2018 (cit. on p. 15).

[SHK+14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958 (cit. on p. 39).

[Shu18] S. Shudler. "Scalability Engineering for Parallel Programs Using Empirical Performance Models". PhD thesis. Technische Universität München, 2018 (cit. on p. 26).

[TN17] M. Toussaint, D. Nguyen-Tuong. *Lecture Slides: Introduction to Machine Learning*. 2017 (cit. on p. 25).

[TPS+19] A. Totounferoush, N. E. Pour, J. Schröder, S. Roller, M. Mehl. “A new load balancing approach for coupled multi-physics simulations”. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2019, pp. 676–682. DOI: DOI10.1109/IPDPSW.2019.00115 (cit. on pp. 51, 55).

[Tri13] A. Tripathi. “Six Ways to Count the Number of Integer Compositions”. In: *Crux Mathematicorum* 39.2 (2013), pp. 84–88 (cit. on p. 22).

[Uek16] B. W. Uekermann. “Partitioned fluid-structure interaction on massively parallel systems”. PhD thesis. Technische Universität München, 2016 (cit. on p. 21).

[Wer74] P. Werbos. “Beyond Regression:”New Tools for Prediction and Analysis in the Behavioral Sciences”. In: *Ph. D. dissertation, Harvard University* (1974) (cit. on p. 31).

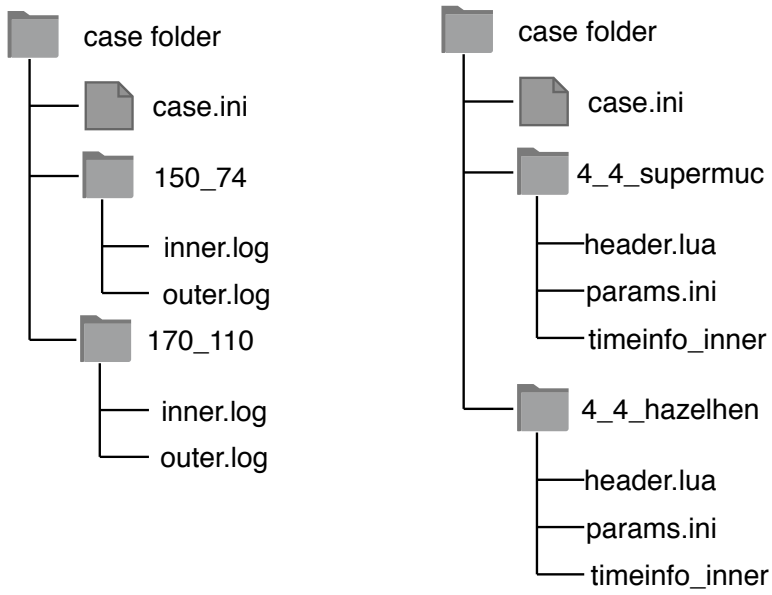
[Zei12] M. D. Zeiler. “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (2012) (cit. on p. 37).

All links were last followed on May 23, 2019.

A Usage instructions

The implementation described in Section 4.2 is provided as command line controlled Python application. To use it, one has to provide the files containing the parameters and run-time measurements in a certain folder structure and then execute the program.

Folder structure



(a) Coupled gaussian pressure pulse
2 solvers, 1 parameter

(b) Monolithic gaussian pressure pulse
1 solver, 3 parameters

Figure A.1: Folder structures for the coupled and monolithic gaussian pressure pulse.

In general, there is a top-level folder (case folder), which contains one subfolder for each measurement and a *case.ini* describing some general properties of the problem. For the coupled gaussian pressure pulse (i.e. two solvers and only taking the core amount into account) this is depicted in Figure A.1a. Each sample folder follows the naming scheme $p_1\_p_2$, where p_1 and p_2 are the amount of cores assigned to the involved domains. For three solvers this would be $p_1\_p_2\_p_3$. This serves as a sanity check and is compared to the maximum rank found in the log file for each domain. The *case.ini* file defines the domains, dimensions and relevant files for each sample. For the above example it is shown in Listing A.1. The sections of this file define the domains, except the DEFAULT section which is used for general parameters, e.g. the input dimension of the problem $d = 1$. For each

Listing A.1 *case.ini* file for coupled gaussian pressure pulse.

```
[DEFAULT]
d=1

[Inner]
events=inner.log

[Outer]
events=outer.log
```

Listing A.2 *case.ini* file for monolithic gaussian pressure pulse.

```
[DEFAULT]
d=3
mesh_header=header.lua
additional=params.ini

[Inner]
timeinfo=timeinfo_inner
```

domain, the user needs to provide a file containing the run-time information, i.e. either a *preCICE*-event-log *preCICE*-event-json or an *Ateles*-timeinfo-file. The first two use the *events* keyword and the latter the *timeinfo* keyword.

For $d > 1$, we need to edit the *case.ini* file and provide files containing the additional parameters. Figure A.1b shows the new folder structure and Listing A.2 is the new *case.ini* file. There is no longer a forced naming scheme for the sample folders, in this case $p_1_l_$system was chosen. The new *case.ini* file provides the filenames for the header file and the file containing the new parameters (*params.ini*). Also it uses an *Ateles* timeinfo file instead of the *preCICE* event file. The *params.ini* can be used to provide arbitrary many additional inputs for the sample, e.g. the clock rate and cores per node as shown in Listing A.3.

Running the application

To run the application the user most always provide a mode (*regression*, *NN* or *parse*) and the path to a case folder. For example,

```
./main.py parse --case ../case_folder/
```

will parse the files in *case_folder* and output each sample with the corresponding run-time. A more advanced call may look like this:

```
./main.py regression --case ../case_folder/ --plot --optimize --optimizeP 560
```

Listing A.3 Example *params.ini* file defining a clock rate value.

```
[DEFAULT]
clock_rate=2600
cores_per_node=28
```

This will create regression models for all domains defined by the case, plot the models and run the optimization for 560 cores.

There is a wide range of options to control the search space of the regression and other behavior of the program. A full help text, including all parameters, can be printed using

```
./main.py --help
```

Listing A.4 Full help text of the program

```
usage: main.py [-h] [--case CASE] [--plot] [--optimize] [--sort] [--log]
               [--plot_1d] [--no_progress] [--plot_len PLOT_LEN]
               [--splitp SPLITP] [--optimizeP OPTIMIZEP]
               [--no_assume_monotonic] [--alpha ALPHA] [--J_start J_START]
               [--J_end J_END] [--J_step J_STEP] [--I_start I_START]
               [--I_end I_END] [--I_step I_STEP] [--n {1,2,3,4,5,6,7,8,9}]
               [--epochs EPOCHS]
               {regression,NN,parse}

Predict and optimize the runtime for solvers.

positional arguments:
  {regression,NN,parse}
                        Which learner to use.

optional arguments:
  -h, --help            show this help message and exit
  --case CASE           Path to case folder.
  --plot                Show plots of the predicted run times
  --optimize            Perform optimization.
  --sort                Sort the files into correct folders before parsing.
  --log                 Applies the logarithm to the run-time of the data
                        points. This helps to increase the accuracy.
  --plot_1d             Will plot the best single parameter models vor multi-
                        dimensional regression. Or plot 1d slices for NN
                        learning.
  --no_progress         Disables progress bars.
  --plot_len PLOT_LEN   Plot will be generated up to the amximum p value in
                        the data plus this.
  --splitp SPLITP       p to split the dataset into training and test data
                        (less than p is training data).
  --optimizeP OPTIMIZEP
                        Number of cores to use for the optimization.
  --no_assume_monotonic
                        If this is set, the optimizer will check all possible
                        core assignments, instead of only those with exactly
                        optimizeP cores.
  --alpha ALPHA         Regularization parameter for the Ridge regression.
  --J_start J_START     Beginning of range for j exponents.
  --J_end J_END         End of range for j exponents.
  --J_step J_STEP       j exponents step size.
  --I_start I_START     Beginning of range for i exponents.
  --I_end I_END         End of range for i exponents.
  --I_step I_STEP       i exponents step size.
  --n {1,2,3,4,5,6,7,8,9}
                        Number of terms for the sum
  --epochs EPOCHS       Number of epochs for the NN optimizer.
```

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature