

Duale Hochschule Baden-Württemberg Mannheim

**Project Thesis**

**Implementation Strategies for Machine  
Learning-Augmented Applications on Hybrid High  
Performance Computing Systems**

**Study Course Business Informatics**

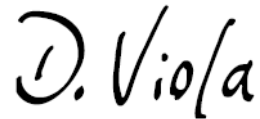
**Field of Study Data Science**

Author:	Dominic Viola
Company:	Hewlett Packard Enterprise
Department:	Center of Excellence
Course:	WWI-19-DSB
Program Director:	Prof. Dr. Bernhard Drabant
Scientific Advisor:	Prof. Dr. Holger Gerhards
Company Advisor:	Philipp Offenhaeuser
Processing Period:	July 16, 2021 – November 15, 2021

# Honorary Declaration

I hereby certify that I have written this paper entitled "*Implementation Strategies for Machine Learning-Augmented Applications on Hybrid High Performance Computing Systems*" independently and have not used any sources or aids other than those indicated. I also certify that the electronically submitted version is identical to the printed version.

Frankfurt, November 11, 2021

A handwritten signature in black ink that reads "D. Viola". The letters are cursive and fluid, with a large 'D' and a stylized 'Viola'.

Location, Date

Dominic Viola

# Abstract

High Performance Computing (HPC) is used to solve scientific and industrial challenges with massively parallel computing resources. Common HPC workflows include simulations, that rely on efficiency and high accuracy. This work explores the use of Machine Learning (ML) in HPC systems to optimize these workflows. HPC systems, which are used by a diverse array of organizations, have tight security restrictions. This complicates the implementation of ML applications. Approaches with the Message Passing Interface (MPI) and SmartSim for multi-node processing are considered. Furthermore, the usage of containers for secure and mobile implementations is explored. Implementation strategies are explored and tested with the Computational Fluid Dynamics (CFD) solver FLEXI and the Reinforcement Learning (RL) framework Relexi. The proposed implementation achieves parallel execution on multiple nodes of the HPC system Hawk. It exhibits a significant performance improvement over the previous single-node implementation.

# Contents

<b>Abstract</b>	<b>ii</b>
<b>List of Figures</b>	<b>v</b>
<b>Acronyms</b>	<b>vi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Aim . . . . .	2
1.3 Structure . . . . .	2
<b>2 Foundational Concepts</b>	<b>4</b>
2.1 High Performance Computing . . . . .	4
2.1.1 Message Passing Interface . . . . .	5
2.1.2 Containerization . . . . .	5
2.1.3 Usage of Graphics Processing Units . . . . .	7
2.2 Computational Fluid Dynamics . . . . .	7
2.3 Machine Learning . . . . .	10
2.3.1 Supervised Learning . . . . .	10
2.3.2 Unsupervised Learning . . . . .	11
2.3.3 Semi-Supervised Learning . . . . .	11
2.3.4 Reinforcement Learning . . . . .	11
2.3.5 Deep Learning . . . . .	12
<b>3 State-of-the-art</b>	<b>15</b>
3.1 Flexi Framework . . . . .	15
3.2 Relexi Algorithm . . . . .	16
<b>4 Available Development Environment</b>	<b>19</b>
4.1 Hardware . . . . .	19
4.2 Software . . . . .	20
<b>5 Implementation Strategies</b>	<b>21</b>
5.1 Distributed Machine Learning . . . . .	22
5.1.1 Model Parallelism . . . . .	22
5.1.2 Data Parallelism . . . . .	22
5.1.3 Parallel Optimization . . . . .	23
5.2 Distributed Simulations . . . . .	24
5.3 Message Passing . . . . .	24
5.4 SmartSim . . . . .	25

5.5	Comparison . . . . .	26
<b>6</b>	<b>Implementation and Testing</b>	<b>28</b>
6.1	Software Stack . . . . .	28
6.2	Selected Implementation Strategy . . . . .	28
6.3	Performance Testing and Results . . . . .	30
<b>7</b>	<b>Conclusion</b>	<b>33</b>
7.1	Discussion . . . . .	33
7.2	Outlook . . . . .	34
	<b>Bibliography</b>	<b>35</b>

# List of Figures

Figure 2.1 Singularity Container . . . . .	6
Figure 2.2 CPU and GPU Training Time Comparison . . . . .	8
Figure 2.3 Concept of an Artificial Neuron . . . . .	13
Figure 2.4 Feed-forward Neural Network with One Hidden Layer . . . . .	14
Figure 3.1 FLEXI Simulation . . . . .	16
Figure 3.2 Relexi Iteration . . . . .	18
Figure 5.1 Parameter Server in a Data Parallel Architecture . . . . .	23
Figure 6.1 Communication Overhead using SmartSim . . . . .	31
Figure 6.2 Weak Scaling . . . . .	32
Figure 6.3 Strong Scaling . . . . .	32

# Acronyms

<b>ANN</b>	Artificial Neural Network
<b>API</b>	Application Programming Interface
<b>CFD</b>	Computational Fluid Dynamics
<b>CPU</b>	Central Processing Unit
<b>DAG</b>	Directed Acyclic Graph
<b>FLOPS</b>	Floating Point Operations per Second
<b>FNN</b>	Feed-forward Neural Network
<b>GPU</b>	Graphics Processing Unit
<b>GUI</b>	Graphical User Interface
<b>HLRS</b>	Höchstleistungsrechenzentrum Stuttgart
<b>HPC</b>	High Performance Computing
<b>HPE</b>	Hewlett Packard Enterprise
<b>ML</b>	Machine Learning
<b>MPI</b>	Message Passing Interface
<b>NAS</b>	Network-Attached Storage
<b>OS</b>	Operating System
<b>RL</b>	Reinforcement Learning
<b>SIF</b>	Singularity Image File
<b>SSD</b>	Solid State Drive
<b>SSH</b>	Secure Shell
<b>TCP</b>	Transmission Control Protocol
<b>UDSS</b>	User Defined Software Stack

# 1 Introduction

## 1.1 Motivation

In the field of High Performance Computing (HPC) scientific and industrial problems are modeled and solved. This process is conventionally done by invoking algorithms and models that extensively rely on Central Processing Unit (CPU) operations. HPC systems offer high-end hardware distribution among many computing nodes. In recent years, a new trend has emerged to combine HPC and Machine Learning (ML) applications, to improve efficiency and performance in both fields [1] [2] [3]. ML applications can find patterns in the way simulations are executed, and hardware is utilized. Consequently, these patterns can be exploited to optimize the simulation. On the other hand, some machine learning applications may require large amounts of data or massive computational power, so that an HPC environment is required.

The usefulness of ML methods varies depending on the availability of data and the task at hand [4, p. 334]. One of these tasks is optimizing Computational Fluid Dynamics (CFD) simulations. The majority of current ML-augmented CFD solvers deploy supervised learning, which requires a large amount of labeled data [5]. In the field of fluid dynamics high fidelity data is expensive in respect to computation time and storage space. Because labeled data is often unavailable, supervised learning approaches are generally impractical. Moreover, labeling data implies that a target value can be assigned to input data deterministically. However, if the solution is non-deterministic, labeled data cannot be used. An alternative approach is Reinforcement Learning (RL) which employs an agent that trains a model interactively inside an environment, without the need for labeled data [6].

Section 2.1.3 shows that the performance of ML applications can be greatly increased with Graphics Processing Units (GPUs). Consequently, in addition to CPU resources, ML benefits from the addition of GPUs. However, there are multiple ways to combine CPUs and GPUs in an HPC system:

1. **Homogeneous clusters:** Each compute node in a cluster employs the same kind of resources. This either means, that all nodes hold the same CPU/GPU configuration,



that the resources of all nodes are from the same vendor or that each node has the same Operating System (OS).

2. **Heterogeneous clusters:** Compute nodes in a cluster employ different hardware or software configurations [7, p. 41] [8, p. 207]. This includes nodes with different CPU/GPU configurations, hardware from different vendors or differing OS'.
3. **Hybrid clusters:** Hybrid clusters in the context of this thesis are defined as nodes with same-vendor same-OS but different CPU/GPU configurations [9]. E.g., some nodes only use AMD CPUs while others use AMD CPUs in combination with GPUs.

This thesis explores the hybrid approach exclusively, employing a RL algorithm that optimizes the parameters of a subgrid-scale model. This model simulates the dynamics of small eddies in a CPU heavy CFD solver. The RL algorithm thereby increases the accuracy and efficiency of the CFD solver.

## 1.2 Aim

This paper aims to provide insight into the considerations before starting to train and deploy a ML model in an HPC environment. The overarching question explored by this thesis is: How should a machine learning model be deployed on a hybrid HPC system?

To explore this question, different approaches to distribute jobs for the most efficient computation are highlighted. A special focus is the connection of a GPU heavy RL agent and distributed CPU-extensive simulation environments.

## 1.3 Structure

*Chapter 2: Foundational Concepts* covers the basic knowledge necessary to understand the implementations proposed in this thesis. This chapter targets entry level HPC users.

In *chapter 3: State-of-the-art*, current approaches and technologies are examined. The findings from this chapter are considered when distribution models are proposed in chapter 5.

*Chapter 4: Available Development Environment* explores the capabilities and limitations of the hardware. Consequently, the recipients can compare their development environment and determine the viability of their implementation on their systems.

*chapter 5: Implementation Strategies* utilizes the insights from chapter 3 to derive implementation approaches for ML-augmented HPC. Afterwards the different methods are compared to conclude which is the best fit for a RL algorithm with CPU-extensive simulation environments.

*Chapter 6: Implementation and Testing* discusses which implementation strategy is the best fit for the available development environment. Performance tests show the viability of the selected implementation strategy.

In *chapter 7: Conclusion*, the procedure and findings of the project are evaluated and critically discussed.

Afterwards, the *chapter 7.2: Outlook* illustrates further potential of the project and possible future development activities.

## 2 Foundational Concepts

### 2.1 High Performance Computing

HPC had its beginning in 1964 with the supercomputer CDC6600, which could perform 3 million Floating Point Operations per Second (FLOPS) [10]. FLOPS measure the peak performance of computer systems [11]. HPC systems expand the boundaries of technological capabilities. Therefore, they enable scientific breakthroughs and innovations. For example, HPC systems enable complex scientific simulations that were previously impossible. Protein simulation for drug development and fluid simulation for climate prediction are some of the use cases.

To achieve high computational performance HPC systems employ massive parallelization. Parallelization is realized within a single machine, but also through distributing workloads in a network of machines. Execution and distribution of computations within a supercomputer's node network differs from standalone computers. Therefore, software had to be developed to cope with the added complexity of the distributed computer network. More on this can be found in subsection 2.1.1. Since there are many different computation nodes in an HPC system that might need the same files it would be impractical for each node to have a separate file system. Instead, Network-Attached Storage (NAS) is used, which can be accessed by all nodes [12].

Massive amounts of electricity are needed to power a supercomputer, therefore code execution must be as efficient as possible. For power efficient and performant programs, low-level programming languages like C, C++ or Fortran are used [13]. For the same reason abstraction layers like high-level languages and virtualization are typically avoided in an HPC setting [14, p. 4]. Nonetheless, Machine Learning workflows heavily rely on abstraction through frameworks, virtual environments, and high-level languages [15, p. 2] [16, p. 1]. Consequently, with advances in Machine Learning applications on HPC systems, efficient abstraction techniques also gain popularity. One such technology is highlighted in subsection 2.1.2.

### 2.1.1 Message Passing Interface

In a single commodity computer parallelization can be achieved using threading. Each application can spawn a process and that process can spawn multiple threads. Through multi-threading application logic can be split into multiple cohesive procedures.

HPC systems are complicated multi-user and multi-node environments. Consequently, the parallelization of an application on such a system is complicated. Another difficulty in designing a parallelization strategy would be the portability of the application. A program that utilizes optimal parallelization using nodes in one HPC environment might not run on another HPC system. As a solution for this problem, the Message Passing Interface (MPI) [17] standard was developed.

MPI is a standard for Application Programming Interfaces (APIs), that allow for the execution of code in a network of compute nodes. There are several implementations of this standard, such as OpenMPI [18], MPICH [19], and MPI for Python [20]. MPI handles the distribution and communication of software between processes. Where processes can be executed on many nodes and can further be distributed to different CPU cores. To execute software with MPI a copy of the application code is send to all ranks and executed on each rank. This means the application has to be aware of its execution context and change procedures accordingly.

The application specifies a communicator object that can exchange information within a set of processes. Processes can address each other by an identifier called rank. The code also specifies how different ranks exchange data. This communication can be point-to-point, meaning from one specific rank to another specific rank, or collective, meaning multiple ranks exchange data. Besides transferring data, processes can also synchronize and compute globally within the scope of their communicator group [21, p. 205].

### 2.1.2 Containerization

A container is a virtualized runtime environment for OS'. Their virtualization layer is on top of the host OS and all containers share the kernel of the host [22, p. 804]. Containerization utilizes resource isolation, so that each container has their own file system, dependencies and applications. This is especially useful in ML applications on HPC. The reason for this is that the common way to supply HPC users with software is unpractical for ML. Commonly software in HPC systems is supplied to users with local software environments,

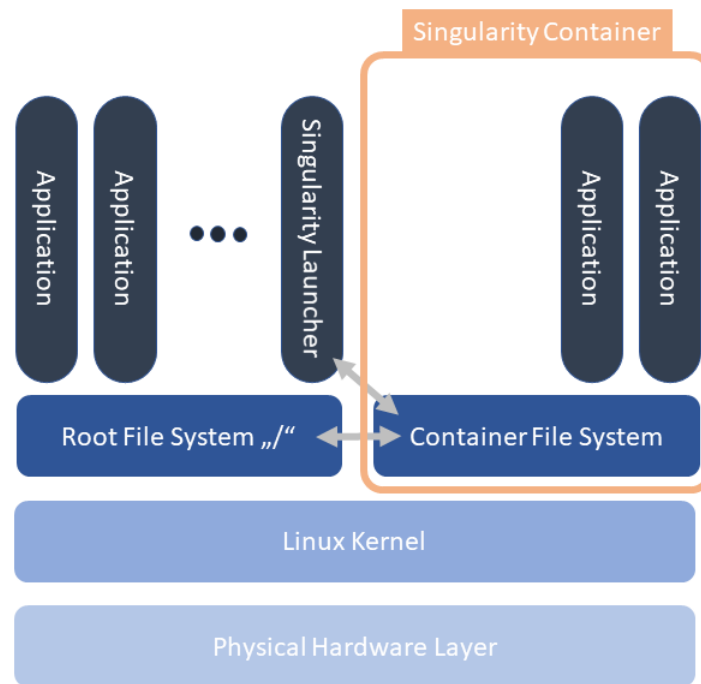


Figure 2.1: Singularity Container [32]

The Singularity container shares the same hardware and kernel as the host OS. The container can be configured to access parts of the file system and software on the host. It contains applications, that are isolated in the container environment.

whereas ML applications use complex software stacks [23]. These software stacks require compatible dependency versions and in some cases internet access [1]. This problem can be mitigated with a User Defined Software Stack (UDSS) inside a container [1].

Widespread file formats for containers are Singularity Image File (SIF) and Docker Image Manifest V2, Schema 2 [24]. Most commonly used container formats comply with the Open Container Initiative standard [25]. This makes it easier to convert container formats between one another. An example workflow of this would be to use Docker [26] container images, convert them to SIF and extend their functionality with a custom Singularity recipe. Recipes contain instructions like commands, file operations and environment variables to create a container with Singularity. Convertibility is especially useful considering that pre-build containers from multiple container registries can be shared, managed and modified [27][28][29]. In some HPC environments internet access is restricted due to security reasons, even in that case local container registries can be maintained by administrators to provide approved container images [30][31].

Singularity is an open source container system [33][34]. It is designed for multi-user en-

vironments such as HPC, while providing security and portability. An illustration of a Singularity container is presented in figure 2.1. As the grey arrows in the illustration suggest, the container can access features and partially the file system on the host. Moreover, users have the same user namespace inside and outside of the container [32, p. 3]. This enhances security and can prevent privilege escalation on the host system for example [35, p. 47 - 48].

### 2.1.3 Usage of Graphics Processing Units

Many sophisticated HPC systems employ accelerators like GPUs [36]. This is because GPUs use a massively parallel architecture that can be exploited for arithmetic and memory intensive operations [37]. There are some application procedures that can utilize this architecture to run faster than on CPU. ML algorithms for example can utilize massively parallel computing with GPUs [38]. An added benefit of using GPUs is that they allow for more compute power on a single compute node because they can be installed in addition to existing CPU resources.

To illustrate the difference in performance, a machine learning training test is conducted. Figure 2.2 shows the result of that test. It was conducted on a single node with two AMD EPYC 7702 64-Core CPUs and 8 Nvidia A100 GPUs each having 108 cores as well as 40GB of GPU memory. The program is configured to use one GPU only. The figure depicts the difference in training time between CPU and GPU execution. It can be observed that the training time is approximately constant and that the training time on GPU is about 17 times faster than on CPU.

## 2.2 Computational Fluid Dynamics

CFD uses computer simulations to model how fluids behave over time. This method is used to explore how fluids such as water and air behave in scenarios where physical experiments are expensive or impossible. Some use-cases of CFD include space flight engineering, climate modelling and modelling cardiovascular flows for medical applications [39, p. 6].

The modeling of fluids can be achieved with the Navier-Stokes equations which consist of three partial differential equations that describe the properties of a fluid [40, p. 141 -142]. They can be solved with numerical methods such as discretization procedures, as

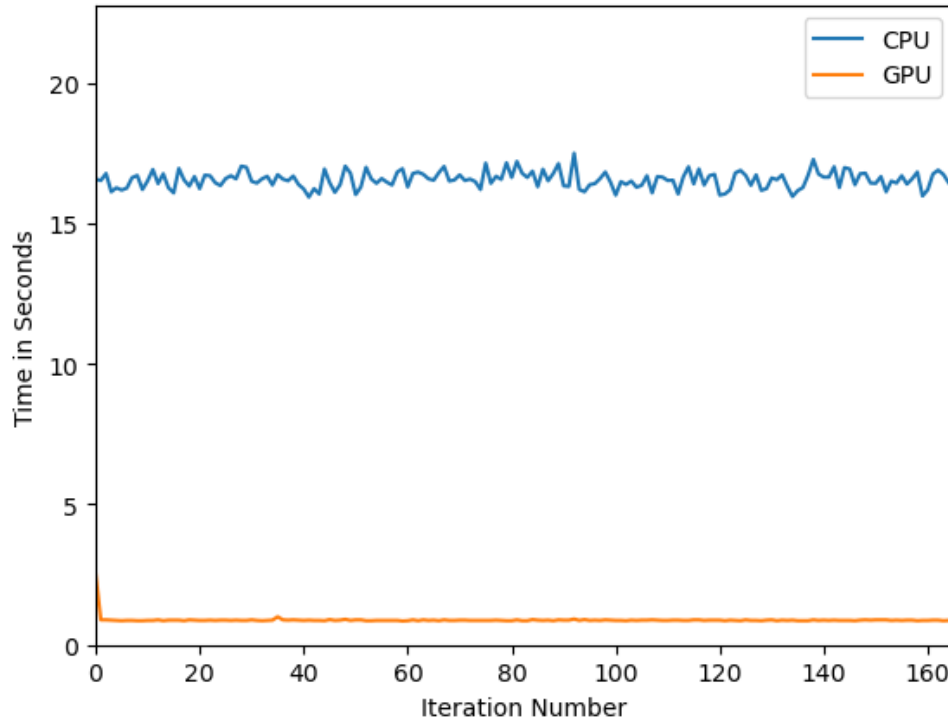


Figure 2.2: CPU and GPU Training Time Comparison

CPU performance fluctuates between 16 and 18 seconds per iteration. While GPU performance starts at approximately 2 seconds per iteration and then quickly drops to about 1 second per iteration, where it stays constant. Thus, the GPU computations are about 17 times faster, than the CPU training performance.

mentioned in section 3.1. The Foundations of computational fluid dynamics are relevant for this thesis because the ML-augmented application deployed in chapter 5 optimizes a CFD solver. It is therefore important to understand the data structures and procedure of a CFD workflow to assess how it can be distributed.

The Navier-Stokes Equations are crucial in CFD. The elements of Navier-Stokes should be understood to allow a high-level interpretation of a CFD simulation. The Navier-Stokes equations are a complex topic that can only be highlighted at a high abstraction level without leaving the scope of this thesis. A detailed explanation and derivation of the Navier-Stokes Equations can be found in the book “Computational fluid dynamics: An Introduction” by Anderson et al. [41, p. 15 – 51]. In an abstract way the Navier-Stokes equations can be described in the following way:

- **Continuity Equation:** It describes the mass conservation within the flow field. The mass can be shifted from one position in the flow field to another one, but the total

mass cannot increase or decrease.

- **Momentum Equation:** The Momentum Equation is based on Newton's Second Law, which states, that momentum is the product of mass and velocity. The momentum equation describes the relation between velocity, density, pressure, viscosity, and external forces such as gravitational and electromagnetic forces.
- **Energy Equation:** Kinetic and potential energy within the flow field can shift, but total energy within the flow field must be conserved. Whereby, kinetic energy results from velocity, mass, pressure, and viscosity. While potential energy results from temperature and heat conductivity.

For any given state of the fluid field, the Navier-Stokes equations describe the properties of that fluid in space and time. Therefore, it can be integrated with respect to time and space, which results in the fluid state at a later time.

In CFD a mesh, also known as a grid, discretizes a flow field into elements, also called cells. While the flow field holds information on the fluid state, the mesh discretizes the problem domain. The smaller those cells are, the more accurate the approximations will be. Without the discretization of the problem domain, the fluid dynamics simulation could not be solved. This is because there is currently no general solution for the Navier-Stokes Equations [42]. However, if certain boundary conditions are met solutions can be calculated. By dividing the continuous fluid field into discrete elements, these conditions are met. Thus, computers can compute the solutions based on the Navier-Stokes equations for each element and its neighbours. Each cell partially contributes to the dynamic of the whole fluid by having an impact on their neighbours. Consequently, the dynamic of the whole fluid can be approximated by the dynamic of all elements in the flow field.

The flow field holds information on the effects that impact each cell including the effects that they pose on their neighbours. Numerical methods like the Navier-Stokes equations are used to calculate the properties of the cells and update the flow field accordingly. This step is repeated multiple times either until equilibrium is reached or until a human-imposed time limit is met.

High computational power is needed for such simulations, as many elements need to be simulated. Thus, parallelization and distribution of the computation steps are preferable. In an HPC environment multiple compute nodes can be used to calculate a CFD simulation. This can be achieved using MPI as described in subsection 2.1.1.



One critical aspect to model reality accurately with CFD simulations are small eddies [43]. They are so small that it would take unreasonable compute resources to simulate them. They are, therefore, considered to be subgrid-scale, meaning smaller than the grid that is used to discretize the flow field. Such subgrid-scale eddies influence the properties of the fluid and, therefore, impact the physical behavior of the fluid. To achieve a physically accurate simulation these effects must be modeled. Subgrid scale models can use the viscosity of small eddies to predict their effect on the fluid [44]. Further information on this topic can be found in *section 3.2: Relexi Algorithm*.

## 2.3 Machine Learning

ML is concerned with teaching computers to take actions without being explicitly programmed to do so [45]. This is achieved by processing large amounts of data. An ML algorithm has the task to find patterns in that data. These patterns are then used to extract or infer relevant information to aid decision-making. ML use cases include computer vision, speech recognition, and robotics [46, p. 2].

Certain machine learning problems may need more processing power or data throughput than a single machine could handle. Therefore, distributed machine learning, which can utilize HPC hardware, emerged. Benefits of distributed machine learning include high data throughput, better accuracy, and shorter training time [47, p. 1196].

### 2.3.1 Supervised Learning

Supervised learning uses labeled data to train a model. This means for each data point there is one desired output value that the model should predict. The error between the model's prediction and the actual label is used to train the model [48, p. 9]. Important to note is that the trained model should have low variance and bias. Whereby low variance means that the algorithm identifies patterns in the training data. It can apply these patterns to unknown data without having memorized the training data. Low Bias means that the model is sufficiently complex to recognize patterns within the dataset [49, p. 1 – 2].

### 2.3.2 Unsupervised Learning

Contrary to supervised learning, unsupervised learning is not conducted with labeled data. Instead, the task in unsupervised learning is to find previously unknown patterns in data [50, p. 65581–65582]. Since no error can be calculated, the performance of the model is hard to determine. It is usually up to human intuition to decide if a model is performant. Similarity metrics, other ML algorithms or probability estimators like log-likelihood can aid that decision.

### 2.3.3 Semi-Supervised Learning

In semi-supervised learning a model is trained on both labeled and unlabeled data. A reason for this could be, that labeled data is expensive to come by so that only a portion of the dataset is labeled. By clustering labeled and unlabeled data and assigning labels based on the respective clusters, it is possible to use unlabeled data for supervised learning tasks. Another approach for labeling the unlabeled data would be to train a model on the labeled data. This model would then be able to annotate the unlabeled data [51, p. 374].

### 2.3.4 Reinforcement Learning

In reinforcement learning an agent interacts with an environment. Based on the observation of these interactions the agent gains experience. This experience is achieved with a reward function that provides the agent with feedback based on the environment state. A positive impact on the environment yields a high reward, while an interaction with a negative impact results in a low reward. The goal of the RL agent is to maximize its reward. Therefore, the agent must learn an optimized policy, so that its interactions yield a high reward [52, p. 3].

In the context of CFD, each environment is equivalent to one simulation. The agent monitors quality metrics like residuals that indicate if the agent-controlled model converges. Convergence is the process to find optimal model parameters so that such a desired output is achieved. From its observations, the agent learns a policy that ensures the highest possible convergence given certain human-imposed restrictions.

### 2.3.5 Deep Learning

Deep learning is concerned with research into Artificial Neural Networks (ANNs). These are networks that are modeled after biological brains and they can be used as models in the aforementioned learning techniques. An ANN consists of neurons that are interconnected.

In figure 2.3 the concept of an artificial neuron is illustrated. Each neuron can receive inputs either from the outside world or from other neurons. In a biological brain, electrical impulses between neurons have different intensities. This is emulated in artificial neurons by multiplying each input with a weight. In addition to the weighted inputs, a bias is used that provides some activation to the neuron regardless of the inputs. The bias is denoted by “b” in figure 2.3. In the next step the weighted inputs and the bias are summed up. This so-called pre-activation is then processed by the transfer function (activation function). The activation of the artificial neuron is its output and can be compared to the electrical signal send by a biological neuron [53, p. 3].

Artificial neural networks utilize multiple artificial neurons that can be assembled in different ways. One such configuration is a Feed-forward Neural Network (FNN). In it, neurons are assembled in layers, where neurons from each layer exclusively feed information to the neurons of the next layer without any loops [54, p. 4]. Layers are categorized in three groups, that are also illustrated in figure 2.4:

1. **Input Layer:** Neurons in this layer process the incoming data.
2. **Hidden Layers:** Neurons in a hidden layer receive input from neurons and deliver output to other neurons. If there are multiple hidden layers the network is called a deep neural network.
3. **Output Layer:** One or more neurons in the output layer modify their pre-activation, so that it can be interpreted by humans. As an example, assume there is an ANN that has one neuron in the output layer and the ANN should provide a probability as an output. Then the activation function of that neuron would squash the information in the interval  $[0, 1]$ .

As the name Feed-forward Neural Network implies, the information is passed forward through the system to receive a desired output. In the training of such a network, however, the weights within the network need to be adjusted, so that the model can converge. For

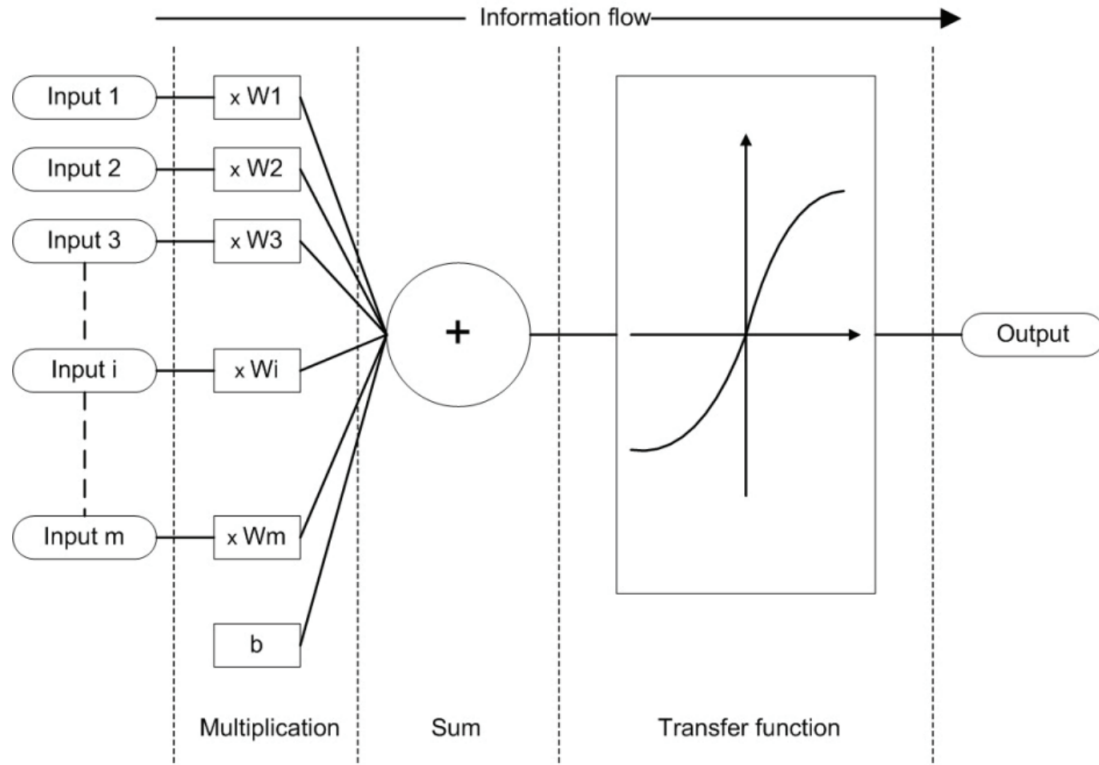


Figure 2.3: Concept of an Artificial Neuron [53]

Information flows into the artificial neuron in the form of multiple inputs. These inputs are each multiplied by a weighting factor and afterwards summed up together with a bias. This pre-activation is transformed by a transfer function. The resulting activation is the output of the artificial neuron.

this, a backward pass with backpropagation is used. A cost function measures the error that the model made in its prediction.

In backpropagation the error gradient, the derivative of the cost function, is used. These gradients show the direction towards the highest error. Beginning at the output layer, the weights within the network are updated in proportion to their contribution to the whole error. The individual weights are updated with a gradient descent algorithm [56]. Gradient descent updates the parameters in such a way, that the result of the cost function is minimized. There are multiple variants of gradient descent. One of them is stochastic gradient descent, that uses random samples of the dataset to calculate optimized parameters [57]. The formula to calculate the batch gradient descent is [58][59]:

$$w' = w - \eta \cdot \Delta w \quad (2.1)$$

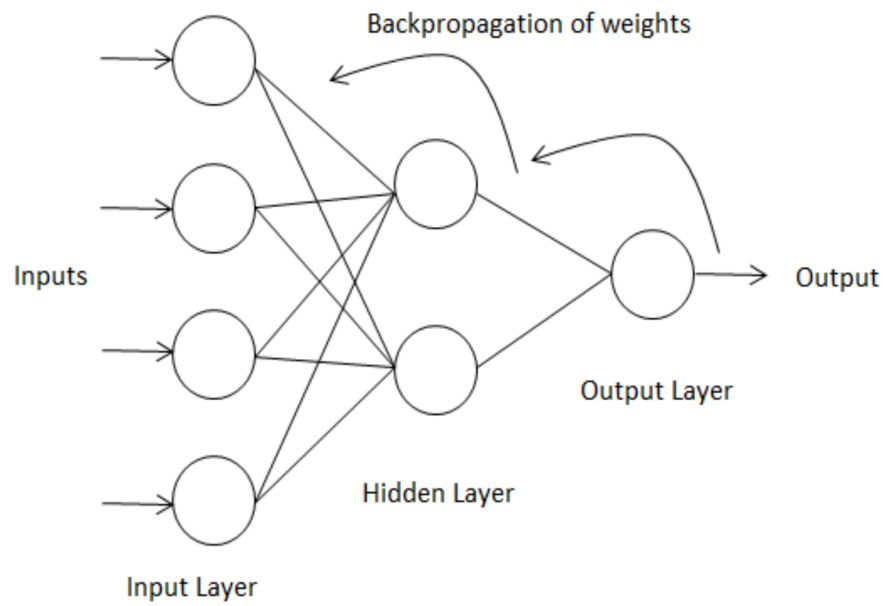


Figure 2.4: Feed-forward Neural Network with One Hidden Layer [55]

A Directed Acyclic Graph (DAG) of artificial neurons is divided into input, hidden and output layers. Inputs are processed by neurons in the input layer they feed their activations to the neurons in the hidden layer. Neurons in the hidden layer process these results and provide their activation to the output layer. The neuron in the output layer processes the data and provides a prediction as output.

Where  $w$  is a vector that holds all parameters of the model.  $\eta$  is the learning rate, that determines how strongly the parameters are updated.  $\Delta w$  is the parameter gradient. It is calculated by deriving the cost function with respect to the parameters  $w$  [60].  $w'$  then holds the updated model parameters such as weight factors in the case of ANNs.

Consequently, ANNs are versatile models that can be used and optimized for a variety of problems. In fact, the universal approximation theorem suggests that neural networks can approximate any continuous function [61][62]. ANNs are revisited in section 3.2, in which the deep reinforcement learning algorithm Relaxi is introduced.

## 3 State-of-the-art

### 3.1 Flexi Framework

FLEXI is an HPC framework for CFD applications [63][64]. It provides a state-of-the-art flow solver as well as pre- and post-processing tools that can create meshes and visualize simulation results. It is developed and maintained by scientists at the Numerical Research Group [65] of the Institute of Aerodynamics and Gas Dynamics - University of Stuttgart. Numerical methods use approximation models to solve problems that cannot be solved otherwise. To approximate sufficiently accurate solutions a high spatial and temporal resolution is necessary. This requires large computational power and generates massive amounts of data. Therefore, FLEXI is built with multi-node parallelization in mind, this allows for state-of-the-art accuracy [63].

The FLEXI framework simulates the fluid field in the following steps, which are also illustrated in figure 3.1:

1. The fluid field state data and the mesh file, are loaded into the program.
2. Parameter values like the number of time steps and the viscosity of small eddies are read from a configuration file that the user modifies.
3. The parameters are used to calculate the forces and the subsequent changes in the flow state for configured time steps in a time frame.
4. The flow field is discretized into cells as defined in the mesh file. For each element, the Navier-Stokes Equations are solved. The flow state is updated depending on the results.
5. Repeat step 4 until the configured maximum time is reached and the simulation terminates.

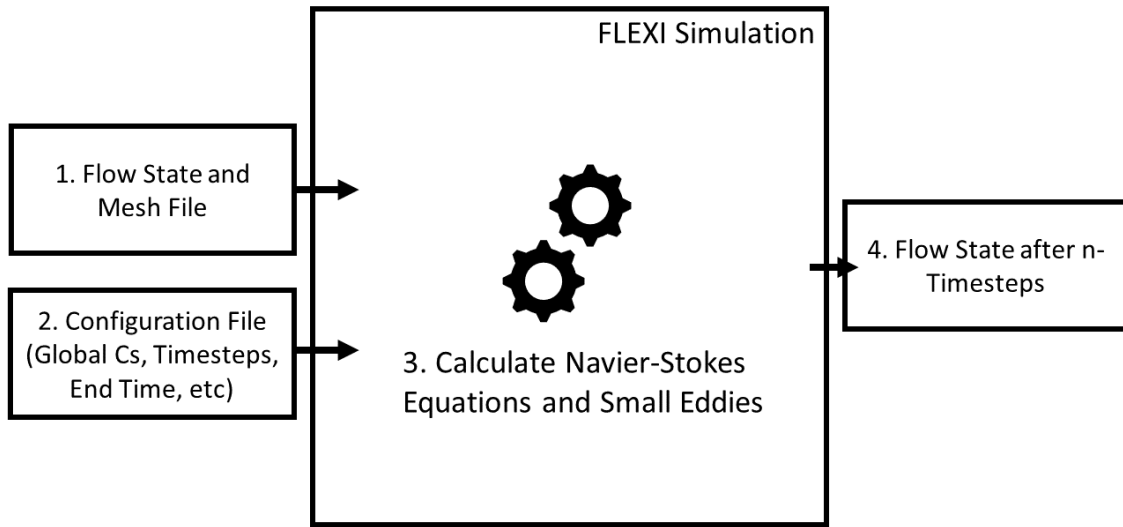


Figure 3.1: FLEXI Simulation

(1.) A FLEXI simulation takes the flow state and a mesh file as input to construct the simulation environment. (2.) The configuration file provides information on how to simulate the fluid dynamic over time by parameterizing FLEXI. (3.) The simulation steps are conducted by calculating the Navier Stokes Equations and by modeling small eddies. (4.) If the End Time is reached the flow state after n-timesteps is provided as output.

## 3.2 Relexi Algorithm

Relexi is a RL framework that is currently under development as part of Marius Kurz's doctoral study at the Numerical Research Group [65].

As described in section 3.1, the FLEXI framework accepts a variety of parameters that can tune the accuracy of the simulation results. To achieve accurate results as fast as possible and with efficient power consumption, good parameter tuning is essential. However, good parameter selection requires expertise and domain knowledge, added that optimal parameters can be unintuitive.

In the past supervised learning was used to obtain optimized parameters for specific problems FLEXI is meant to solve. Yet, it is still expensive to produce labeled data for the supervised learning algorithm as mentioned in chapter 1.1. Therefore, the Relexi algorithm uses an RL approach to the problem. The benefit of this approach lies in the generalization that reinforcement learning provides. The reinforcement learning agent can learn a generalized policy that can determine optimal parameters based on any given simulation scenario.

Whereas supervised learning can only optimize the parameters for one scenario and only if labeled data is available for that problem. Nonetheless, to achieve a high-quality parameter determination policy the reinforcement agent needs a well designed reward function. This is a challenging task, especially considering that a multitude of metrics is available to determine the reward function.

Relaxi focuses on an essential part of a FLEXI simulation, which is optimizing the viscosity parameter that is used to model the effects of small eddies on the flow field. This viscosity parameter is from now on denoted as  $C_s$ . In CFD simulations small eddies are often too small to be simulated with reasonable time and compute resources. Instead, the viscosity ( $C_s$ ) of such eddies is used to model their effect on the rest of the system. This is crucial to achieve physically accurate simulations. In a common FLEXI simulation  $C_s$  is provided in the configuration file and applied to all cells. With Relaxi, however,  $C_s$  is provided for each cell and updated over time.

Relaxi uses a deep neural network as its policy to decide the  $C_s$  values. It will use the state of a fluid field and determine which parameters will be optimal for the state. The training of the neural network is performed in multiple iterations. At the end of each iteration the Policy Gradient Theorem is applied to determine the gradient of the reward function [66]. Based on the gradient, backpropagation is used to update the weights within the neural network.

The process of a Relaxi training iteration is illustrated in figure 3.2. One iteration is structured as follows:

1. Based on the configuration multiple FLEXI simulations with different  $C_s$  parameters are spawned in separate environments. Their  $C_s$  parameters are determined by the RL policy.
2. Each FLEXI instance simulates the same fluid field. Each simulation follows the steps as described in section 3.1, with the addition that  $C_s$  is provided for each element and it is updated by the neural network policy over time.
3. The performance metrics for all FLEXI simulations are observed by the agent. The agent generates a reward based on the performance of each environment.
4. The Policy Gradient Theorem is used to estimate gradients from the reward [6]. The weights of the policy are then updated using backpropagation with those gradients.



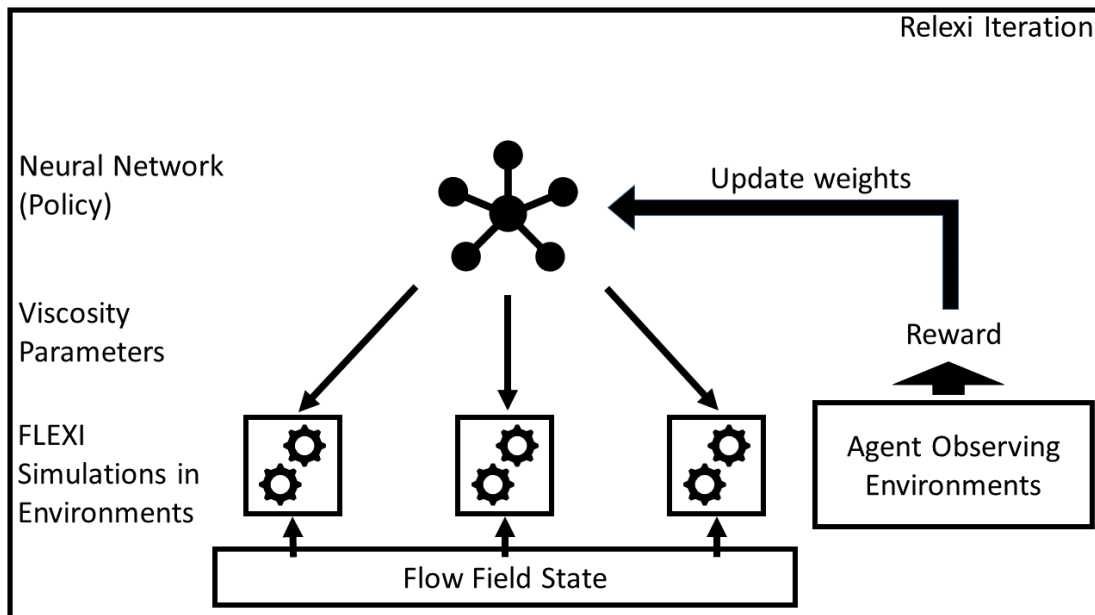


Figure 3.2: Relexi Iteration

Multiple FLEXI simulations are started in individual environments, that are observed by the RL agent. These simulations load the same flow state, mesh file, and configuration parameters, except the viscosity parameter. This parameter is provided by the policy. Depending on the state of the environments the agent awards a reward for each environment, that is used to update the weights in the policy with backpropagation.

These iteration steps repeat until the model converges. To determine the performance of the policy evaluation steps are conducted regularly. Each evaluation step is conducted with a fluid field simulation that was not used to update the weights of the policy. In that way the performance of the policy on unknown data is tested. It also ensures that the policy is finding logical patterns instead of memorizing training data.

## 4 Available Development Environment

The development environment used in the context of this work consists of a Hewlett Packard Enterprise (HPE) Apollo 9000 system and a recent extension for ML applications the Apollo 6500 Gen10 Plus, the system is also called Hawk [67]. It is on-site at Höchstleistungsrechenzentrum Stuttgart (HLRS) and has a peak performance of 26 peta-FLOPS [67]. As of June 2021, it is the 18th most powerful computer in the world according to TOP500 [68].

As mentioned before, HPC systems are multi-node systems. This means operating on an HPC system requires a different workflow, than a one-node system like a laptop for example. On Hawk there are some nodes that are used for login purposes and others that can only be addressed by a batching system for heavy workloads [69].

### 4.1 Hardware

The node-to-node interconnect consists of a 9-D hypercube topology with a bandwidth of 200Gbit/s. The interconnect is used for connecting multiple systems including the HPE Apollo 9000, the HPE Apollo 6500 Gen10 Plus and a parallel filesystem. A list of all the connected systems can be found here [70].

The parallel file system can be used to store data persistently, while some compute nodes have ephemeral Solid State Drive (SSD) storage, meaning that it is deleted after the user disconnects from it. The parallel file system is a distributed system that can be accessed by all compute nodes and enables parallel file operations. Therefore, it could be used to exchange data between nodes during the execution of multi-node applications.

The hardware specification of the Hawk system is displayed in table 4.1. The table separates the core system HPE Apollo 9000 and the HPE Apollo 6500 Gen10 Plus ML extension because they have different compute resources. Nodes from both systems can be addressed by the batch system to form a hybrid cluster.

Table 4.1: Compute Resources [67][68]

	HPE Apollo 9000	HPE Apollo 6500 Gen10 Plus
Number of nodes	5,632	24
CPUs per node	2	2
Cores per CPU	64	64
CPU frequency	2.25 GHz	2.0 GHz
CPU architecture	AMD EPYC (“Rome”)	AMD EPYC (“Rome”)
GPUs per node	-	8
GPU type	-	NVIDIA A100
GPU Memory	-	20 x 40GB and 4 x 60 GB
Average Memory per node	256 GB	1,000 GB
Node to node interconnect	Mellanox Dual Rail Infini-Band HDR200	Mellanox Dual Rail Infini-Band HDR200

## 4.2 Software

The OS on the Hawk nodes is CentOS Linux 8.1 without a Graphical User Interface (GUI) [68]. Precompiled and optimized applications are provided to users through the “module” command line tool, that can be accessed on all nodes. This includes applications for running and compiling software, math libraries, session management tools, and container tools. The available software to run containers is Singularity, as described in section 2.1.2.

The default MPI implementation is HPE MPI, even though other implementations are available through the module tool [68]. The software to request and schedule node clusters is called batch system. Hawk uses the PBS Pro batch system.

Due to security reasons, direct access from within hawk to the internet is not possible. However, local mirrors provide access to some repositories and programming language libraries. If packages outside of this scope are needed it is also possible to use a Secure Shell (SSH)-tunnel to get access to required software [71]. The usage of SSH-tunnels under the restrictions of the HLRS firewall.

# 5 Implementation Strategies

This chapter highlights different strategies to deploy ML augmented HPC applications, considering the capabilities of HPC systems. Challenges arise from the difference in programming paradigms between performance-critical HPC applications and rapidly changing ML applications. HPC programs utilize low-level programming to maximize performance, while ML workloads are subject to changes, as part of the fields rapid development [72, p. 1–2].

For the HPC-ML implementation it is important to distinguish online and offline learning. In online learning the ML algorithm learns from streaming data, processing one record at a time. Therefore, the learning process can take place while the data is generated by another program. This allows for parallel execution and real-time training, but comes at the cost of communication overhead. In practice, the HPC application generates data during a simulation run, while the ML workflow reads the streaming data and optimizes the HPC application at runtime.

On the contrary, the offline approach doesn't have to rely on communication to receive data. Instead, the data generation takes place first and the generated dataset is used for training afterwards [73, p. 1]. In this process the data is loaded into memory as a batch and training takes place over the entire batch. This means, that the HPC simulation runs first and generates data during execution. After the simulation terminates an ML application uses this data to optimize the simulation.

The online approach tightly integrates ML and HPC applications, while offline learning leaves both applications mostly decoupled. According to Schrittwieser et al. offline reinforcement learning can utilize pre-existing datasets and is useful if environment interactions are costly [74]. They argue further that online learning can provide state-of-the-art performance.

## 5.1 Distributed Machine Learning

### 5.1.1 Model Parallelism

In model parallelism the ML model is split into separate components that contain parts of the model's logic [75, p. 1]. These submodels are distributed among nodes. The submodels then communicate results to one another, so that they function as one distributed model [76, p. 6]. That way the model can scale beyond the limitations of one compute node. However, this comes at the cost of communication latency and possible connection issues that have to be considered in the model logic.

Jaderberg et al. point out, that the training process of an ANN has sequential processes, which result in challenges for model parallelism [77]. The computation of forward and backward passing as well as the update of weights can lead to locking in the distributed model. This means that all submodels need to synchronize at the end of these processes, which makes the model's speed dependent on the slowest submodel. Jaderberg et al. solved this issue with the communication protocol Decoupled Neural Interface and Synthetic Gradients [77]. To utilize the parallel compute resources of HPC systems efficiently, such challenges must be considered in the implementation. This leads to a longer development time.

### 5.1.2 Data Parallelism

With a data-parallel approach multiple instances of an identical ML model are deployed to multiple nodes. Each model processes different data as inputs. This is achieved by partitioning the input space into discrete subspaces. This means multiple nodes ingest subsets of the whole dataset [78, p. 2 – 3]. Figure 5.1 illustrates the data-parallel architecture, the components and processes are explained hereinafter.

In data parallelism, each model is trained with a subset of the whole dataset. This enables fast processing of large datasets, as the bandwidth can scale with the number of nodes. The model instances then exchange their parameters with a parameter server. This server collects, computes, and serves parameters to worker nodes that run models [79]. Therefore, models are executed independently and there is no direct communication between them [80, p. 2], which reduces communication overhead. Nonetheless, all models synchronize parameters through the parameter server. This accelerates convergence because all model

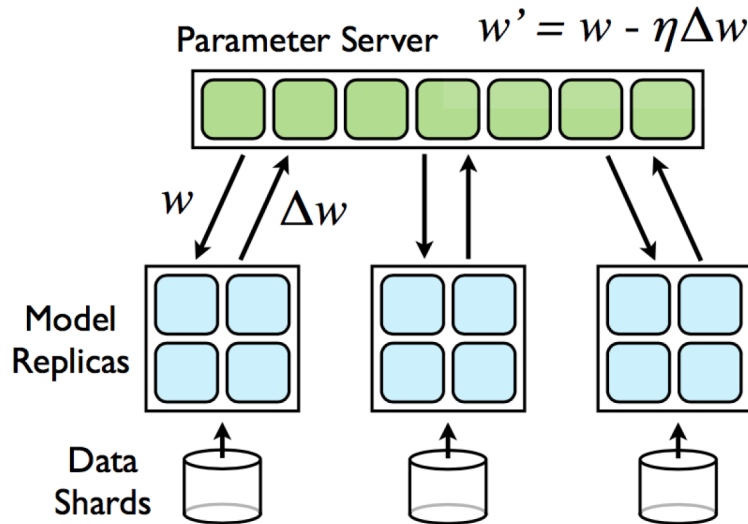


Figure 5.1: Parameter Server in a Data Parallel Architecture [58]

Data shards feed subsets of the whole dataset to replicated model instances. These models exchange their weight gradients with the parameter server. The server computes updated weights with a gradient descent algorithm. The updated weights are then served to the model instances.

instances learn by collective parameter optimization. The parameter optimization by the parameter server can be achieved with error gradients (weight gradients) and the gradient descent algorithm, introduced in section 2.3.5 [79].

### 5.1.3 Parallel Optimization

Besides architectures to train and deploy models in a parallel manner, ML models can also be optimized in parallel. The multi-node characteristic of HPC systems can be used to optimize hyperparameters for ML models. Parallel hyperparameter search splits the interval of reasonable hyperparameters into subsets. Each model is then parameterized with a different subset of hyperparameters [81, p. 44].

Similar to data parallelism hyperparameters are assessed by a single node, however, parallel hyperparameter search doesn't rely on collective hyperparameter updates. Instead, each model instance is initialized with different hyperparameters and their respective configurations are then evaluated. The evaluation is conducted in parallel by algorithms such as SMAC [82] or ASHA [83] for example. By searching through hyperparameter subspaces in parallel, optimal hyperparameters can be determined. This approach outperforms the parameterization by human experts [84, p. 8].

## 5.2 Distributed Simulations

Distributing simulations is preferable if the ML algorithm has to optimize a few parameters of the simulation and therefore is computationally light weight. HPC applications are heavy CPU applications that usually take a long time to compute. In that case, multiple simulations per ML application are needed to ensure that the ML application gains sufficient data to learn from.

Parallelization within one compute node can be viable for the development and debugging of the application. However, it underutilizes distributed computing resources in an HPC system. Compared to multi-node simulation environments the single-node approach lacks in compute resources with only one compute node. With multi-node simulation environments, a large number of simulations can be processed in parallel. This utilizes the distributed nature of HPC systems and generates more data for an RL agent or any other kind of ML application. Due to environment distribution, the application also scales better, so that the degree of distribution can vary depending on the underlying hardware or the amount of data necessary.

In the case of offline learning, where the data is first generated by the simulations and then processed by the ML algorithm, it can be preferable to use multiple simulations. Generating data from multiple differently parameterized simulations at the same time will speed up the data generation process. Thus, learning with the ML algorithm can start sooner.

## 5.3 Message Passing

The aforementioned distribution models scale to multiple nodes that require communication between them, as they do not share the same memory. In section 2.1.1 MPI was introduced to mitigate this issue.

MPI is a widespread standard to program parallel applications for systems that don't have a shared memory space. MPI applications are performant and portable to other systems that use MPI [85]. The development with MPI is different from the development of applications with a shared memory space. This means existing applications that were originally developed for shared memory space systems need to be rewritten for MPI support. MPI provides primitives that abstract the complexity of interprocess communication. Nonetheless, the implementation is a time-consuming process.

There are downsides to distributing applications with MPI. For example, if multiple MPI applications are started at once all applications can be terminated if one rank returns an error. That is because MPI interprets that error as fatal and stops all applications in the same communicator group the fatal rank belongs to. However, such errors can be the result of unreasonable decisions of the ML algorithm during the learning process.

## 5.4 SmartSim

SmartSim is an orchestration framework, that allows for distributed program execution on HPC systems [13]. It has a specific focus on combining typical CPU extensive HPC codes with ML applications. Data between applications is exchanged without using the file system, this is achieved by using a Redis in-memory database that can be distributed across nodes. The Redis database enables asynchronous data exchange and real-time training, inference, and analysis on data within the database.

Data is represented as key-value pairs that can be sent to and retrieved from a database. It is also possible to poll for data and execute code after it is received. Data can be stored in tensors and multiple tensors can be combined into a dataset. Such data types are sent with the Transmission Control Protocol (TCP) between clients and databases. Thus, SmartSim is based on the client-server architecture. Due to this architecture HPC and ML workflows are loosely coupled, as they don't communicate directly. This means existing standalone applications require few modifications to work together compared to a MPI approach, which uses direct communication. SmartSim provides modules for C, C++, Fortran and Python to provide the aforementioned functionalities for clients, while the server-side logic is built into the framework.

SmartSim comes with built-in logic for multiple database instances and tensor distribution between them. Thus, large distributed applications can scale without compromising performance by memory shortage and bandwidth restrictions. Another feature that SmartSIM provides is to commit TorchScript to the database that is then executed within the database. TorchScript can be used for pre- and post-processing of data inside the database.



## 5.5 Comparison

In the implementation of a ML-augmented HPC simulation, there are four criteria that should be considered to arrive at an implementation strategy and an appropriate technology to implement that strategy. These considerations concern:

1. Software Stack: How are the necessary dependencies installed on the HPC system?
2. Data Ingestion: How does the ML application receive simulation data?
3. Machine Learning Distribution: Which method is used to distribute the ML workload to multiple nodes?
4. Simulation Distribution: How many different simulations should run in parallel?

There are two options to deploy the necessary software stack for the simulation and ML code. One option is to use containers to deploy the software and the other option is to install it on bare-metal. The container approach has the advantage, that there is no restriction to the software versions and combinations that can be installed. With the bare-metal approach there might be software versions that are incompatible with pre-installed applications in the HPC environment. This also implies, that the user has the privileges to install and run custom software. Users must also be able to use SSH-tunnels or similar methods to get the software onto the HPC system. However, the bare-metal approach has better performance than containerization because there is no virtualization layer involved. Containers have the advantage of portability, yet driver software for the HPC hardware might be necessary inside the container. If that is the case, then MPI and SmartSim could both be used.

Data ingestion into the ML algorithm can be online or offline. Online learning is time-efficient, however, it comes with the downside of communication overhead. The communication logic needs to be integrated into simulations and ML. Offline learning on the other hand doesn't need communication and it can use pre-existing datasets. However, data generation and training cannot be done in parallel. SmartSim communication can be integrated into simulation and ML codes with little effort, compared to MPI. SmartSim also allows to attach new simulations and ML applications to the infrastructure at run-time, MPI doesn't have this functionality. Consequently, SmartSim is well suited for online learning.

The distribution of the machine learning algorithm can be realized with data parallelism, model parallelism, a combination of both or not at all, if one compute node has enough resources. If large amounts of data need to be ingested and high data throughput is essential the data-parallel method should be used. SmartSim can store ML models in an in-memory database, this enables fast ML model distribution for the data-parallel approach. If the ML model is large, like a complex ANN with many neurons, the model parallel approach is advisable. To implement this model parallelism fast direct communication between nodes can be achieved with MPI. The combination of both approaches is used for high data throughput and a large model. In this case the ML model could be implemented as a MPI application and multiple instances of it can be launched with SmartSim. If the ML model is light weight and the input data can be handled by one node and implementation time is sparse, the single node approach can be used.

Multiple simulations can be distributed to multiple nodes, one simulation can scale among nodes or both. A simulation should be scaled with the complexity of the simulated problem and required simulation time. In combination with online learning, the number of simulations should depend on the data that the ML algorithm can ingest. If SmartSim is used, the ML model can start and stop simulations as needed and attach them to the infrastructure.

# 6 Implementation and Testing

## 6.1 Software Stack

Machine Learning applications typically rely on specific versions of libraries and other dependencies, that are available for download through package managers. However, the available development environment, as well as some other HPC environments, do not have internet access. Therefore, an approach was necessary that could ensure that all dependencies are present without a direct internet connection. The first challenge to implement a scalable ML-augmented application is consequently to get a UDSS on the HPC system.

The initial approach was using singularity containers that are built on a computer with internet access so that all dependencies could be downloaded to the container. Afterwards the built container image is copied onto the HPC environment, where the container is started. Then a script is executed that would configure and compile dependencies according to the available hardware in the HPC environment. As discussed in section 5.5, MPI from inside a container requires driver packages for the interconnect hardware. If the MPI version inside and outside the container match, the application can reach other nodes outside the container. Due to time constraints and unforeseen delays in the Hawk ML extension deployment, this obstacle could not be overcome. Thus, the Container approach could only be tested on a single node, not with MPI parallelization between multiple nodes.

The second approach was using a bare-metal installation, meaning that the UDSS is downloaded directly to the Hawk hardware with SSH-tunnels, as described in section 5.5. The pre-installed software on the HPC environment, like Python version and compilers for Fortran and C, were compatible with the UDSS. Consequently, this approach allows deploying scalable applications with native MPI support. Although, this solution offers less portability, than the container approach.

## 6.2 Selected Implementation Strategy

The ML application Relexi and the CFD application FLEXI are deployed on bare-metal. Relexi follows the RL approach with environment interaction and therefore requires online

learning. There is no pre-existing dataset to train Relexi on, consequently offline RL is not possible [73]. The communication of the FLEXI simulation state to Relexi is realized with SmartSim. Reasons to use SmartSim are that simulations can be added and removed at runtime, that it can be integrated with few changes to the existing codebases, and that the resulting architecture is modular. Furthermore, SmartSim is designed for distributed use in HPC systems and provides MPI support.

Relexi is efficient if it is executed on GPU hardware, while the simulations are CPU-heavy. Thus, using multiple CPU simulation nodes to generate data and only one GPU ML node to process the data is viable. While parallelization of the ML application is possible, it is sufficient to process the data without ML distribution. This has the advantage, that Relexi can incorporate the code to start the SmartSim infrastructure. Moreover, it can start and stop FLEXI instances as needed and connect them to the infrastructure. The communication in the online learning process is currently implemented in a blocking manner. Blocking means both FLEXI and Relexi wait for the communication to finish, before they advance their respective program logic.

The SmartSim infrastructure can launch applications via MPI. Because it is integrated into Relexi, different simulation distribution strategies can be controlled by Relexi. The amount and CPU cores of simulation environments can be determined in a configuration file.

The communication between FLEXI and Relexi is through the Redis database, while the communication within a FLEXI instance is achieved with MPI. Each FLEXI instance gets a tag assigned by Relexi to avoid collisions of tensor keys in the database. The following data is exchanged:

1. When a FLEXI instance is started, the initial state throughout all ranks is gathered with MPI and then sent to the database. Relexi reads that tensor and prepares the training process for it.
2. After a configurable number of timesteps, the ranks in a FLEXI instance gather the current state and send it to the database. The FLEXI instance blocks at this point and awaits the updated Cs parameters from Relexi. Relexi reads the state and sends the updated Cs to the FLEXI instance via the database. FLEXI scatters the new parameters throughout its ranks and continues the next timestep with the new Cs.
3. After each FLEXI timestep a tensor is exchanged that indicates if FLEXI is about to terminate. Relexi then determines if a new FLEXI instance should be started.

## 6.3 Performance Testing and Results

This chapter will test the communication overhead and the scalability of the selected implementation strategy. The performance is measured with the FLEXI performance index (PID). The PID is a normalized measure of compute time. The PID is calculated like so [64, p. 40]:

$$PID = \frac{\text{wall clock time} \cdot \#cores}{\#DOF \cdot \#time\ steps \cdot \#RK\ stages} \quad (6.1)$$

Where the numerator consists of the wall clock time that is the elapsed time frame and  $\#cores$  that are the number of CPU cores used. In the denominator  $\#DOF$  is the number of degrees of freedom calculated,  $\#time\ steps$  is the number of FLEXI simulation steps and  $\#RK\ stages$  is the number of approximation steps calculated with the Runge-Kutta method. Runge-Kutta is a numerical method to solve differential equations. In the context of this section, the wall clock time also incorporates the time of all communications between Relexi and FLEXI, as described in the enumeration in section 6.2.

The performance of the resulting architecture is tested for the communication overhead and the scaling efficiency of one simulation environment.

Figure 6.1 shows the communication overhead of a single FLEXI environment, that uses one CPU core. The blocking communication is realized via tensors in the in-memory Redis database. This database is hosted on the same node as Relexi within the HPE Apollo 6500 Gen10 Plus extension. The FLEXI simulation is conducted on a node in the HPE Apollo 9000 main system. The communication overhead is about 2.4%. However, the overhead can be reduced by using threading to pre- and post-process data in a parallel manner. Also, with non-blocking communication more time could be utilized for computation, which could further increase performance.

The conducted scaling tests in figures 6.2 and 6.3 show the efficiency of the implemented strategy, described in section 6.2. With weak scaling, the compute resources are scaled along with the problem complexity, so that the workload per CPU core stays constant. In contrast, strong scaling only scales the compute resources, while the problem complexity stays constant. In both tests, the performance of one node is used as baseline efficiency at 100%. The native FLEXI scaling efficiency is super-linear, this means that the parallel efficiency rose over 100% in some test cases [64, p. 40 – 42]. Whereas, the weak and

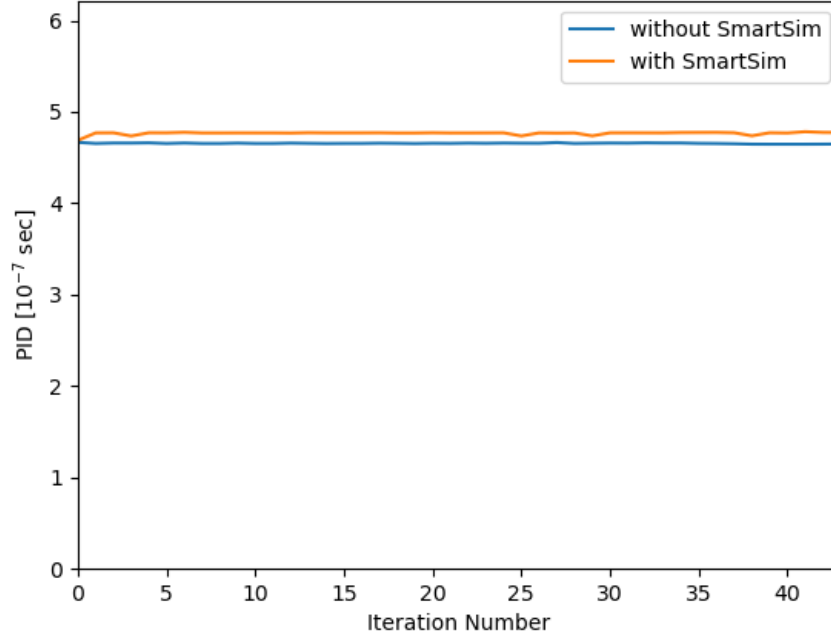


Figure 6.1: Communication Overhead using SmartSim

Shown is the performance measure of a FLEXI application without any communication compared with a FLEXI application that communicates with the SmartSim infrastructure. Both simulations were conducted with the same mesh and with one CPU core. The SmartSim infrastructure was executed on a different node. The performance measure includes the time to communicate the solution with the SmartSim infrastructure. The overhead is on average about 2.4%.

strong scaling efficiency of the proposed implementation strategy continually declines. This can be attributed to the blocking communication with Relexi and the sequential execution of Relexi logic. Regardless of the parallelization of FLEXI there is only one sequential Relexi instance.

Amdahl's law describes the parallelization speedup of a program as a function of the parallelizable fraction of the program [86]. The non-parallelizable fraction of the program is sequential. According to Amdahl's law, the speedup is limited by the sequential fraction of the program [87].

The principle of Amdahl's law can be applied to the weak and strong scaling results of the proposed implementation. The scaling efficiency declines, due to the unparallelized Relexi procedures.

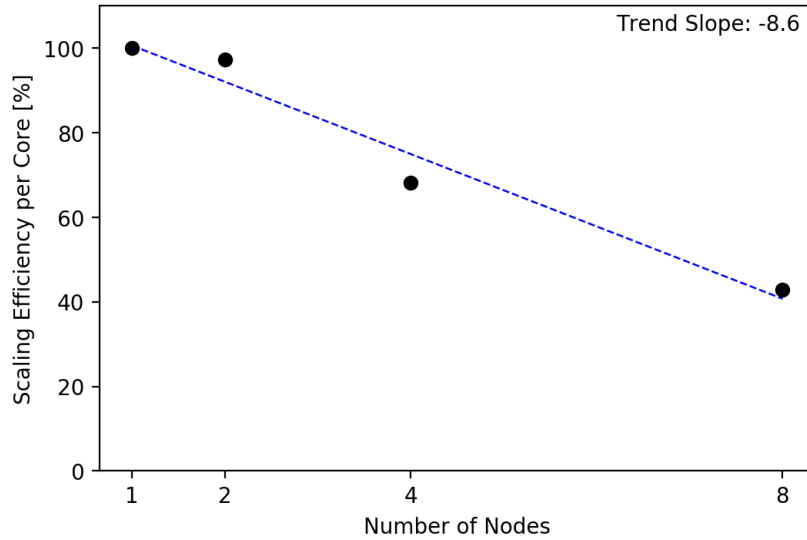


Figure 6.2: Weak Scaling

One FLEXI instance was scaled across 1, 2, 4, and 8 nodes, that each used 128 CPU-cores. The mesh was scaled along with the nodes so that each core had to solve the differential equations for 4 cells. FLEXI communicated with Relexi through the SmartSim infrastructure, which was running on a different node. For each number of nodes, the mean from 20 simulation timesteps is depicted. The scaling efficiency is based on the PID. One node is the baseline efficiency and resembles 100%.

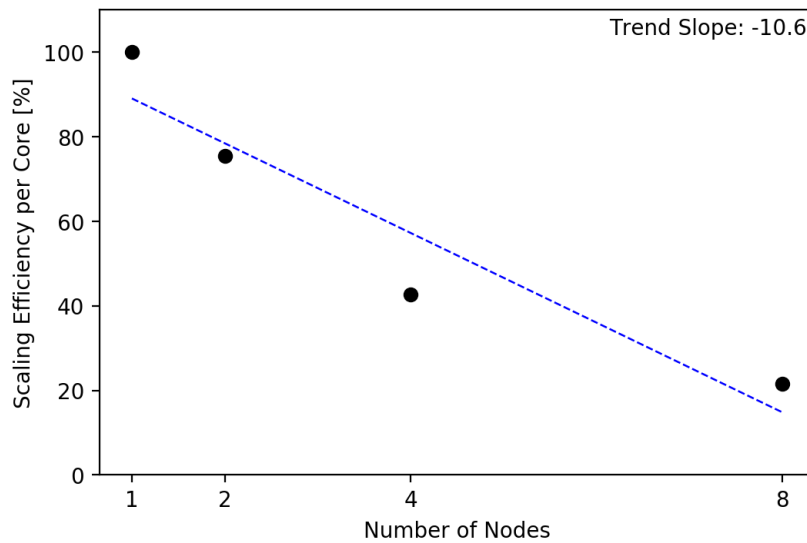


Figure 6.3: Strong Scaling

One FLEXI instance was scaled across 1, 2, 4, and 8 nodes, that each used 128 CPU-cores. The mesh had a constant size of 4,096 cells. FLEXI communicated with Relexi through the SmartSim infrastructure, which was running on a different node. For each number of nodes, the mean from 20 simulation timesteps is shown. The scaling efficiency is based on the PID. The performance of one node is used as the baseline efficiency and resembles 100%.

# 7 Conclusion

This thesis explored different aspects of implementing a ML-augmented HPC application. Such aspects include the software stack, the distribution of the ML and HPC applications and their communication. Different technologies were examined, their capabilities were explored and their constraints were highlighted. Afterwards, a implementation strategy was proposed and tested.

## 7.1 Discussion

In chapter 6.3 it was shown that the implemented strategy can scale to multiple nodes to use the massively parallel resources that HPC systems possess. The strategy has scope for improvement, when it comes to scaling efficiency. However, the proposed implementation strategy was selected as a proof-of-concept, that represents a significant improvement over the previous state-of-the-art. Due to time constraints and delays in the Hawk ML extension, further scaling optimizations were not possible.

Nonetheless, performance can be improved by using parallel Relaxi procedures with threading. Without changing the implementation strategy, pre- and post-processing of data can be parallelized. Moreover, threading also enables a technique called latency hiding, that can further increase performance. Communication latency occurs when data is transferred between nodes or different hardware components, like CPUs and GPUs. With latency hiding, the latency time is overlapped with continued computation to improve communication performance [88]. Additionally, Relaxi inference can be distributed between multiple GPUs on a node. This can reduce inference time and, therefore, speed up the calculation of a parameter update.

Although there are performance improvements, the proposed strategy delivers a significant improvement over the preceding state-of-the-art. The previous implementation could only be executed on a single node. In contrast the proposed implementation strategy can utilize multiple nodes with over 100 cores each and achieve a speedup.

In a context of the demonstrated project, the proposed strategy is a good fit. Yet, the best implementation strategy depends on the hard- and software in the HPC environment, pre-



existing applications and the desired integration of ML and HPC application. Hereinafter, exemplary implementation contexts are highlighted.

One context could be that portability of the software is not a concern, a HPC application exists and a ML algorithm is not implemented, yet. If ML and HPC applications should be tightly integrated, than a MPI implementation on bare-metal could be used to get optimal performance. If there is a vast amount of parameters that the ML algorithm should optimize, than the model parallel approach can be used.

Another context could be that pre-existing HPC and ML applications should be connected and scaled. Furthermore, the HPC environment provides optimized containers. The project specification requires rapid development on the HPC system to train the ML model at scale. The model should be portable, so that local development is also possible. In this scenario containers are preferable for portable use, without the risk to interfere with pre-installed software. Modifying the existing applications to communicate with the SmartSim infrastructure makes the development fast, because few code changes have to be made. SmartSim's loose coupling makes it possible to switch out the HPC application and simulate its behaviour for local ML development.

Consequently, there is no single best implementation strategy. Instead, the implementation context dictates which strategy fits best.

## 7.2 Outlook

In section 3.2, Relexi was introduced as a ML Framework that is currently under development. The proposed improvements in section 7.1 could be integrated into Relexi in the course of development. The performance of the final implementation will certainly surpass the performance presented in this thesis. When the development is finished, Relexi and its source code will be published by Marius Kurz. This thesis showed that ML and HPC workflows have different and in some cases contradictory principles. Nonetheless, the use of ML on HPC systems is a logical consequence of ever increasing data volumes, that are to be analysed. HPC applications generate large amounts of data that cannot be processed manually. Therefore, ML applications with high computational power are needed to extract information from this data pool. In addition, the strive towards ever more efficient HPC applications drives their optimization by ML applications forwards. Thus, the Center of Excellence at HPE will further investigate how ML and HPC workflows can be combined.

# Bibliography

- [1] David Brayford and Sofia Vallecorsa. “Deploying Scientific AI Networks at Petaflop Scale on Secure Large Scale HPC Production Systems with Containers”. In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '20: Platform for Advanced Scientific Computing Conference (Geneva, Switzerland). New York, NY, USA: ACM, 2020, pp. 1–8. ISBN: 9781450379939. DOI: 10.1145/3394277.3401850. URL: <https://arxiv.org/ftp/arxiv/papers/2005/2005.10676.pdf> (visited on 09/20/2021).
- [2] Shinyoung Ahn et al. “ShmCaffe: A Distributed Deep Learning Platform with Shared Memory Buffer for HPC Architecture”. In: *2018 IEEE 38th International Conference on Distributed Computing Systems*. 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS) (Vienna). IEEE International Conference on Distributed Computing Systems et al. Piscataway, NJ: IEEE, 2018, pp. 1118–1128. ISBN: 978-1-5386-6871-9. DOI: 10.1109/ICDCS.2018.00111. URL: <http://www.cs.ucf.edu/~mohaisen/doc/icdcs18a.pdf> (visited on 09/21/2021).
- [3] Mark Endrei et al. “Statistical and machine learning models for optimizing energy in parallel applications”. In: *The International Journal of High Performance Computing Applications* 33.6 (2019), pp. 1079–1097. ISSN: 1094-3420. DOI: 10.1177/1094342019842915.
- [4] Steven L. Brunton, Maziar S. Hemati, and Kunihiko Taira. “Special issue on machine learning and data-driven methods in fluid dynamics”. In: *Theoretical and Computational Fluid Dynamics* 34.4 (2020), pp. 333–337. ISSN: 0935-4964. DOI: 10.1007/s00162-020-00542-y. URL: <https://link.springer.com/article/10.1007/s00162-020-00542-y> (visited on 11/08/2021).
- [5] Kai Fukami, Koji Fukagata, and Kunihiko Taira. “Assessment of supervised machine learning methods for fluid flows”. In: *Theoretical and Computational Fluid Dynamics* 34.4 (2020), pp. 497–519. ISSN: 0935-4964. DOI: 10.1007/s00162-020-00518-y. URL: <https://arxiv.org/pdf/2001.09618.pdf> (visited on 10/21/2021).
- [6] Suraj Pawar and Romit Maulik. “Distributed deep reinforcement learning for simulation control”. en. In: *Machine Learning: Science and Technology* 2.2 (2021), p. 025029. ISSN: 2632-2153. DOI: 10.1088/2632-2153/abdaf8. URL: <https://iopscience.iop.org/article/10.1088/2632-2153/abdaf8/meta> (visited on 11/11/2021).
- [7] J. D. Teresco, J. Faik, and J. E. Flaherty. “Resource-Aware Scientific Computation on a Heterogeneous Cluster”. In: *Computing in Science and Engineering* 7.2 (2005), pp. 40–50. ISSN: 1521-9615. DOI: 10.1109/MCSE.2005.38.

- [8] F. Kon et al. “2K: a distributed operating system for dynamic heterogeneous environments”. In: *Proceedings / the Ninth International Symposium on High-Performance Distributed Computing. Pittsburg, Pennsylvania, USA, August 1 - 4, 2000*. Proceedings the Ninth International Symposium on High-Performance Distributed Computing (Pittsburgh, PA, USA, ). IEEE Computer Society. Los Alamitos, Calif.: IEEE Computer Soc, 2000, pp. 201–208. ISBN: 0-7695-0783-2. DOI: 10.1109/hpdc.2000.868651.
- [9] Roberto Capuzzo-Dolcetta, Mario Spera, and Davide Punzo. “A fully parallel, high precision, N-body code running on hybrid computing platforms”. In: *Journal of Computational Physics* 236 (2013), pp. 580–593.
- [10] Sumit Narayan. “Supercomputers: past, present and the future”. In: *XRDS: Crossroads, The ACM Magazine for Students* 15.4 (2009), pp. 7–10. URL: <https://dl.acm.org/doi/fullHtml/10.1145/1558897.1558900> (visited on 09/23/2021).
- [11] Jaegeun von Han, Jaegeun von Han, and Bharatkumar Sharma. *Learn CUDA Programming*. 2019. ISBN: 9781788991292.
- [12] Dongfang Zhao and Ioan Raicu. “Distributed file systems for exascale computing”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12), doctoral showcase*, pp. 267–276. URL: [https://www.researchgate.net/publication/258244870\\_Distributed\\_file\\_systems\\_for\\_exascale\\_computing](https://www.researchgate.net/publication/258244870_Distributed_file_systems_for_exascale_computing) (visited on 11/11/2021).
- [13] Sam Partee et al. *Using Machine Learning at Scale in HPC Simulations with SmartSim: An Application to Ocean Climate Modeling*. 13.04.2021. URL: <https://arxiv.org/pdf/2104.09355> (visited on 11/08/2021).
- [14] Victor R. Basili et al. “Understanding the High-Performance-Computing Community: A Software Engineer’s Perspective”. In: *IEEE Software* 25.4 (2008), pp. 29–36. ISSN: 0740-7459. DOI: 10.1109/MS.2008.103.
- [15] Nicolas Vasilache et al. *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*. 13.02.2018. URL: <https://arxiv.org/pdf/1802.04730.pdf> (visited on 09/23/2021).
- [16] Yuan Tang. *TF.Learn: TensorFlow’s High-level Module for Distributed Machine Learning*. 13.12.2016. URL: <https://arxiv.org/pdf/1612.04251.pdf> (visited on 09/23/2021).
- [17] MPI Forum. *MPI Documents*. 13.09.2021. URL: <https://www.mpi-forum.org/docs/> (visited on 09/29/2021).

- [18] The Open MPI Project. *Open MPI: Open Source High Performance Computing*. 23.09.2021. URL: <https://www.open-mpi.org/> (visited on 09/23/2021).
- [19] MPICH Contributors. *MPICH / High-Performance Portable MPI*. 23.09.2021. URL: <https://www.mpich.org/> (visited on 09/23/2021).
- [20] Lisandro Dalcin. *MPI for Python documentation*. 14.08.2021. URL: <https://mpi4py.readthedocs.io/en/stable/> (visited on 09/23/2021).
- [21] Hesham el Rewini and Mostafa Abd-El-Barr. *Advanced computer architecture and parallel processing*. Wiley series on parallel and distributed computing. Hoboken, NJ: Wiley-Interscience, 2005. 272 pp. ISBN: 9780471478386. DOI: 10.1002/0471478385. URL: <https://onlinelibrary.wiley.com/doi/book/10.1002/0471478385> (visited on 11/09/2021).
- [22] Sachchidanand Singh and Nirmala Singh. "Containers & Docker: Emerging roles & future of Cloud technology". In: *2016 2nd International Conference 21.7.2016 - 23.7.2016*, pp. 804–807. DOI: 10.1109/ICATCCT.2016.7912109.
- [23] Lucas Benedicic et al. "Sarus: Highly Scalable Docker Containers for HPC Systems". In: *High Performance Computing. ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers*. Ed. by Michèle Weiland et al. Springer eBook Collection 11887. Springer International Publishing and Imprint Springer, 2019, pp. 46–60. ISBN: 978-3-030-34356-9. URL: [https://link.springer.com/chapter/10.1007/978-3-030-34356-9\\_5](https://link.springer.com/chapter/10.1007/978-3-030-34356-9_5) (visited on 11/11/2021).
- [24] Microsoft, Inc. *Supported content formats - Azure Container Registry*. 20.09.2021. URL: <https://docs.microsoft.com/en-us/azure/container-registry/container-registry-image-formats> (visited on 09/20/2021).
- [25] The Linux Foundation. *Open Container Initiative*. 20.09.2021. URL: <https://opencontainers.org/> (visited on 09/20/2021).
- [26] Docker, Inc. *Docker*. 17.09.2021. URL: <https://www.docker.com/> (visited on 09/20/2021).
- [27] Docker, Inc. *Docker Hub Container Image Library*. 20.09.2021. URL: <https://hub.docker.com/> (visited on 09/20/2021).
- [28] Google Cloud. *Container Registry*. 6.09.2021. URL: <https://cloud.google.com/container-registry> (visited on 09/20/2021).
- [29] Amazon Web Services, Inc. *Amazon Elastic Container Registry*. 28.08.2021. URL: <https://aws.amazon.com/de/ecr/> (visited on 09/20/2021).

- [30] GitLab B.V. *GitLab Container Registry*. 20.09.2021. URL: [https://docs.gitlab.com/ee/user/packages/container\\_registry/](https://docs.gitlab.com/ee/user/packages/container_registry/) (visited on 09/20/2021).
- [31] Vanessa Sochat. *Singularity Registry (HPC)*. 3.09.2021. URL: <https://singularity-hpc.readthedocs.io/en/latest/> (visited on 09/20/2021).
- [32] Carlos Arango, Rémy Dérnat, and John Sanabria. *Performance Evaluation of Container-based Virtualization for High Performance Computing Environments*. 2017. URL: <https://arxiv.org/pdf/1709.10140.pdf> (visited on 09/29/2021).
- [33] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. “Singularity: Scientific containers for mobility of compute”. In: *PLOS ONE* 12.5 (2017), e0177459. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0177459. eprint: 28494014. URL: <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0177459> (visited on 11/11/2021).
- [34] HPCng. *Source Code of Singularity*. *GitHub*. 17.09.2021. URL: <https://github.com/hpcng/singularity> (visited on 09/17/2021).
- [35] Anton Semjonov. *Security analysis of user namespaces and rootless containers*. en. 2020. DOI: 10.15480/882.3089. URL: <https://tore.tuhh.de/bitstream/11420/7891/1/thesis-r231-g18e9edc.pdf> (visited on 09/20/2021).
- [36] Michael Feldman. *New GPU-Accelerated Supercomputers Change the Balance of Power on the TOP500 | TOP500*. TOP500. 9.11.2021. URL: <https://www.top500.org/news/new-gpu-accelerated-supercomputers-change-the-balance-of-power-on-the-top500/> (visited on 11/09/2021).
- [37] Volodymyr V. Kindratenko et al. “GPU clusters for high-performance computing”. In: *2009 IEEE International Conference on Cluster Computing and workshops. (Cluster 2009) ; New Orleans, Louisiana, USA, 31 August - 4 September 2009*. 2009 IEEE International Conference on Cluster Computing and Workshops (New Orleans, LA, USA, ). Institute of Electrical and Electronics Engineers. Piscataway, NJ: IEEE, 2009, pp. 1–8. ISBN: 978-1-4244-5011-4. DOI: 10.1109/CLUSTER.2009.5289128.
- [38] Michal Marks and Ewa Niewiadomska-Szynkiewicz. “Hybrid CPU/GPU platform for high performance computing”. In: *Proceedings / 28th European Conference on Modelling and Simulation ECMS 2014*. 28th Conference on Modelling and Simulation. Ed. by Flaminio Squazzoni and Evtim Peytchev. ECMS, European Council for Modelling and Simulation, and European Conference on Modelling and Simulation. 2014, pp. 508–514. ISBN: 9780956494481. DOI: 10.7148/2014-0508.

- [39] Bin Xia and Da-Wen Sun. "Applications of computational fluid dynamics (cfd) in the food industry: a review". In: *Computers and Electronics in Agriculture* 34.1-3 (2002), pp. 5–24. ISSN: 0168-1699. DOI: 10.1016/S0168-1699(01)00177-6. URL: <https://www.sciencedirect.com/science/article/pii/S0168169901001776> (visited on 11/08/2021).
- [40] P. T. Williams and A. J. Baker. "INCOMPRESSIBLE COMPUTATIONAL FLUID DYNAMICS AND THE CONTINUITY CONSTRAINT METHOD FOR THE THREE-DIMENSIONAL NAVIER-STOKES EQUATIONS". In: *Numerical Heat Transfer, Part B: Fundamentals* 29.2 (1996), pp. 137–273. ISSN: 1040-7790. DOI: 10.1080/10407799608914980.
- [41] Anderson, John David and Wendt, J. *Computational Fluid Dynamics. An Introduction*. Vol. 206. Springer, 1995. ISBN: 978-3-540-85055-7. URL: <https://link.springer.com/content/pdf/10.1007/978-3-540-85056-4.pdf> (visited on 11/11/2021).
- [42] Clay Mathematics Institute. *Navier–Stokes Equation Millennium Problem*. 24.09.2021. URL: <https://www.claymath.org/millennium-problems/navier%E2%80%9393stokes-equation> (visited on 09/24/2021).
- [43] O. Agullo et al. "Large eddy simulation of decaying magnetohydrodynamic turbulence with dynamic subgrid-modeling". In: *Physics of Plasmas* 8.7 (2001), pp. 3502–3505. ISSN: 1070-664X. DOI: 10.1063/1.1372337. URL: [https://web.archive.org/web/20180720234119id\\_/https://hal-amu.archives-ouvertes.fr/hal-01625879/document](https://web.archive.org/web/20180720234119id_/https://hal-amu.archives-ouvertes.fr/hal-01625879/document) (visited on 11/09/2021).
- [44] F. Ducros, F. Nicoud, and T. Poinso. *Wall-adapting local eddy-viscosity models for simulations in complex geometries*. 1998. URL: [https://imag.umontpellier.fr/~nicoud/pdf/icfd\\_wale.pdf](https://imag.umontpellier.fr/~nicoud/pdf/icfd_wale.pdf) (visited on 11/09/2021).
- [45] Batta Mahesh. *Machine Learning Algorithms - A Review*. 2019. DOI: 10.21275/ART20203995. URL: [https://www.researchgate.net/publication/344717762\\_Machine\\_Learning\\_Algorithms\\_-\\_A\\_Review](https://www.researchgate.net/publication/344717762_Machine_Learning_Algorithms_-_A_Review) (visited on 11/11/2021).
- [46] Tom M. Mitchell. *The Discipline of Machine Learning*. Vol. 9. Carnegie Mellon University, 2006. URL: <http://ra.adm.cs.cmu.edu/anon/usr0/ftp/anon/ml/CMU-ML-06-108.pdf> (visited on 10/31/2021).
- [47] Alex Galakatos, Andrew Crotty, and Tim Kraska. "Distributed Machine Learning". In: *Encyclopedia of Database Systems*. New York, NY: Springer New York, 2018, pp. 1196–1201. DOI: 10.1007/978-1-4614-8265-9\_80647.
- [48] Osvaldo Simeone. *A Brief Introduction to Machine Learning for Engineers*. 2018. URL: <http://arxiv.org/pdf/1709.02840v3> (visited on 11/11/2021).

- [49] Brady Neal et al. *A Modern Take on the Bias-Variance Tradeoff in Neural Networks*. 2018. URL: <https://arxiv.org/pdf/1810.08591> (visited on 11/11/2021).
- [50] Muhammad Usama et al. "Unsupervised Machine Learning for Networking: Techniques, Applications and Research Challenges". In: *IEEE Access* 7 (2019), pp. 65579–65615. ISSN: 2169-3536. DOI: 10.1109/access.2019.2916648.
- [51] Jesper E. van Engelen and Holger H. Hoos. "A survey on semi-supervised learning". In: *Machine Learning* 109.2 (2020), pp. 373–440. ISSN: 0885-6125. DOI: 10.1007/s10994-019-05855-6.
- [52] Csaba Szepesvári. "Algorithms for reinforcement learning". In: *Synthesis lectures on artificial intelligence and machine learning* 4.1 (2010), pp. 1–103.
- [53] Andrej Krenker, Janez Bester, and Andrej Kos. "Introduction to the Artificial Neural Networks". In: *Artificial Neural Networks - Methodological Advances and Biomedical Applications*. InTech, 2011. DOI: 10.5772/15751.
- [54] Jürgen Schmidhuber. "Deep learning in neural networks: an overview". In: *Neural networks : the official journal of the International Neural Network Society* 61 (2015), pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003. eprint: 25462637.
- [55] Yana Mazwin Mohmad Hassim and Rozaida Ghazali. "Training a Functional Link Neural Network Using an Artificial Bee Colony for Solving a Classification Problems". In: *Journal of Computing* (2012). URL: <https://arxiv.org/pdf/1212.6922> (visited on 11/11/2021).
- [56] M. Riedmiller and H. Braun. "A direct adaptive method for faster backpropagation learning: the RPROP algorithm". In: *IEEE International Conference on Neural Networks*. IEEE, 1993. DOI: 10.1109/icnn.1993.298623.
- [57] Léon Bottou. "Stochastic Gradient Descent Tricks". In: *Neural networks: tricks of the trade*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. 2nd ed. Vol. 7700. Lecture Notes in Computer Science 7700. Berlin and Heidelberg: Springer, 2012, pp. 421–436. ISBN: 978-3-642-35288-1. DOI: 10.1007/978-3-642-35289-8\_25.
- [58] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *Advances in Neural Information Processing Systems*. Vol. 25. URL: <https://papers.nips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf> (visited on 09/21/2021).
- [59] Martin Zinkevich et al. *Parallelized stochastic gradient descent*. 4th ed. Vol. 1. NIPS. 2010. URL: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.374.1458&rep=rep1&type=pdf> (visited on 11/10/2021).

- [60] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 15.09.2016. URL: <https://arxiv.org/pdf/1609.04747.pdf> (visited on 10/21/2021).
- [61] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: <https://link.springer.com/article/10.1007/bf02551274> (visited on 11/11/2021).
- [62] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 08936080. DOI: 10.1016/0893-6080(89)90020-8.
- [63] *Flexi – High Performance Open Source CFD*. 3.08.2021. URL: <https://www.flexi-project.org/> (visited on 08/03/2021).
- [64] Nico Krais et al. *FLEXI: A high order discontinuous Galerkin framework for hyperbolic-parabolic conservation laws*. 7.10.2019. URL: <https://arxiv.org/pdf/1910.02858> (visited on 11/09/2019).
- [65] Institute of Aerodynamics and Gas Dynamics. *Numerics Research Group*. University of Stuttgart. 9.09.2021. URL: <https://www.iag.uni-stuttgart.de/en/working-groups/numerical-methods/> (visited on 09/09/2021).
- [66] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: (1999). URL: <https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf> (visited on 09/17/2021).
- [67] HLRS High Performance Computing Center Stuttgart. *HPE Apollo (Hawk)*. 11.08.2021. URL: <https://www.hlrs.de/systems/hpe-apollo-hawk/> (visited on 08/11/2021).
- [68] Erich Strohmaier et al. *Hawk - Apollo 9000*. TOP500. 30.10.2021. URL: <https://www.top500.org/system/179880/> (visited on 10/30/2021).
- [69] HLRS High Performance Computing Center Stuttgart. *HPE Hawk access*. 20.07.2021. URL: [https://kb.hlrs.de/platforms/index.php/HPE\\_Hawk\\_access](https://kb.hlrs.de/platforms/index.php/HPE_Hawk_access) (visited on 10/24/2021).
- [70] HLRS High Performance Computing Center Stuttgart. *Systems*. 31.10.2021. URL: <https://www.hlrs.de/systems/> (visited on 10/31/2021).
- [71] HLRS Platforms. *Secure Shell ssh*. 20.07.2021. URL: [https://kb.hlrs.de/platforms/index.php/Secure\\_Shell\\_ssh#if\\_you\\_can.27t\\_get\\_a\\_connection](https://kb.hlrs.de/platforms/index.php/Secure_Shell_ssh#if_you_can.27t_get_a_connection) (visited on 10/28/2021).



- [72] Jordan Ott et al. *A Fortran-Keras Deep Learning Bridge for Scientific Computing*. 14.04.2020. URL: <https://arxiv.org/pdf/2004.10652> (visited on 11/08/2021).
- [73] Scott Fujimoto and Shixiang Shane Gu. *A Minimalist Approach to Offline Reinforcement Learning*. 12.06.2021. URL: <https://arxiv.org/pdf/2106.06860.pdf> (visited on 11/04/2021).
- [74] Julian Schrittwieser et al. *Online and Offline Reinforcement Learning by Planning with a Learned Model*. 13.04.2021. URL: <https://arxiv.org/pdf/2104.06294> (visited on 11/11/2021).
- [75] Zhihao Jia, Matei Zaharia, and Alex Aiken. *Beyond Data and Model Parallelism for Deep Neural Networks*. 14.07.2018. URL: <https://arxiv.org/pdf/1807.05358> (visited on 11/11/2021).
- [76] Adrián Castelló et al. "Analysis of model parallelism for distributed neural networks". In: *Proceedings of the 26th European MPI Users' Group Meeting*. the 26th European MPI Users' Group Meeting (Zürich, Switzerland). Ed. by Torsten Hoefler. Ed. by Jesper Larsson Träff. ACM Digital Library. New York, NY, United States: Association for Computing Machinery, 2019, pp. 1–10. ISBN: 9781450371759. DOI: 10.1145/3343211.3343218. URL: [https://www.researchgate.net/publication/335377467\\_Analysis\\_of\\_model\\_parallelism\\_for\\_distributed\\_neural\\_networks](https://www.researchgate.net/publication/335377467_Analysis_of_model_parallelism_for_distributed_neural_networks) (visited on 11/08/2021).
- [77] Max Jaderberg et al. *Decoupled Neural Interfaces using Synthetic Gradients*. 2016. URL: <https://arxiv.org/pdf/1608.05343> (visited on 11/11/2021).
- [78] Hao Li et al. "Malt: distributed data-parallelism for existing ml applications". In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–16.
- [79] Yuchen Fan et al. "Model Aggregation Method for Data Parallelism in Distributed Real-Time Machine Learning of Smart Sensing Equipment". In: *IEEE Access* 7 (2019), pp. 172065–172073. ISSN: 2169-3536. DOI: 10.1109/access.2019.2955547.
- [80] Kailai Xu, Weiqiang Zhu, and Eric Darve. *Distributed Machine Learning for Computational Engineering using MPI*. 2.11.2020. URL: <https://arxiv.org/pdf/2011.01349> (visited on 11/11/2021).
- [81] M. Todd Young et al. "Distributed Bayesian optimization of deep reinforcement learning algorithms". In: *Journal of Parallel and Distributed Computing* 139 (2020), pp. 43–52. ISSN: 07437315. DOI: 10.1016/j.jpdc.2019.07.008. URL: <https://www.sciencedirect.com/science/article/pii/S0743731519300231> (visited on 11/11/2021).

- [82] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. “Sequential Model-Based Optimization for General Algorithm Configuration”. In: *Learning and intelligent optimization*. Ed. by Carlos A. Coello Coello. Vol. 6683. Lecture Notes in Computer Science 6683. Berlin and Heidelberg: Springer, 2011, pp. 507–523. ISBN: 978-3-642-25565-6. DOI: 10.1007/978-3-642-25566-3\_40.
- [83] Liam Li et al. *A System for Massively Parallel Hyperparameter Tuning*. Version 5. 2018. URL: <https://arXiv.org/1810.05934> (visited on 10/31/2021).
- [84] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. “Practical bayesian optimization of machine learning algorithms”. In: *Advances in neural information processing systems* 25 (2012).
- [85] J. Piernas, A. Flores, and J. M. García. “Analyzing the performance of MPI in a cluster of workstations based on fast ethernet”. In: *Recent advances in parallel virtual machine and message passing interface*. Ed. by Marian Bubak. Vol. 1332. Lecture Notes in Computer Science 1332. Berlin and Heidelberg: Springer, 1997, pp. 17–24. ISBN: 978-3-540-63697-7. DOI: 10.1007/3-540-63697-8\_65.
- [86] Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)* (Atlantic City, New Jersey, ). New York, New York, USA: ACM Press, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: <https://dl.acm.org/doi/pdf/10.1145/1465482.1465560> (visited on 11/09/2021).
- [87] Stijn Eyerman and Lieven Eeckhout. “Modeling critical sections in Amdahl’s law and its implications for multicore design”. In: *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10* (New York, New York, USA). New York, New York, USA: ACM Press, 2010. DOI: 10.1145/1815961.1816011.
- [88] V. Strumpen and T. L. Casavant. “Exploiting communication latency hiding for parallel network computing: model and analysis”. In: *Proceedings of 1994 International Conference on Parallel and Distributed Systems*. IEEE Comput. Soc. Press, 1994. DOI: 10.1109/icpads.1994.590409.