Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Parallel Machine Learning of Fluid-Structure Interaction

Moritz Widmayer

Course of Study:	Informatik
Examiner:	Prof. Dr. Miriam Schulte
Supervisor:	Amin Totounferoush, M.Sc.
Commenced:	February 1, 2022
Completed:	August 1, 2022

Acknowledgement

I want to express my gratitude to Prof. Dr. Miriam Schulte for allowing me to complete and write my master's thesis at her department of the IPVS. I also want to say thank you to my thesis supervisor, Amin Totounferoush, for all of his time and work in helping me complete the thesis. I appreciate my family and friends' encouragement to go higher as well as their support during difficult times and want to give special thanks to my parents for always standing behind me. Finally, I want to thank Tabea Schoch for inspiring me and providing me with a cause to keep going.

Abstract

This thesis applies machine learning (ML) methods to the numerical simulation of a fluid-structure interaction problem involving an elastic tube containing a liquid. A neural network, utilizing fully connected and recurrent layers, is trained on the simulation data such that the prediction during inference generates the subsequent time step to a given input batch. To make use of the parallelization technique presented in [1], we partition the tube and train a separate neural network on each partition independently. Due to low accuracy and continuity of poor quality in the predictions, we further make use of having adjacent partitions overlap such that information can be shared.

Our results suggest using multiple layers in the neural network architecture is superior to only having single layers. Additionally, the application of L2 regularization in form of weight decay has no detrimental effect on the error of the predictions. The overlapping technique demonstrates a highly valuable method of increasing the continuity of the predictions whereas we find no significant difference regarding the amount of overlap used. Finally, the implemented parallelization approach can make better use of machines providing a higher number of CPU cores when compared to the multithreading offered by the used ML library directly.

Contents

1 Introduction

1.1 Overview

In this thesis, we use machine learning (ML) methods to improve the numerical simulation of a fluid-structure interaction (FSI) problem. While the origins of ML methods can be traced back to the 1950s [2] truly viable and broad usage of deep learning has only started to appear in the previous decade [3]. Since then ML methods have been widely studied and deployed commercially but also in many fields of science [4].

One of the scientific use cases of ML methods that shows promising results is their application in physical simulations [5]. Data generated through these simulations can be used for training of various classes of ML methods such as neural networks [6]. These methods are then used to either completely replace classical simulations [7], enhance their results [8] or improve their computational performance [9].

Availability of a large amount of data, either from simulations or even directly originating out of experiments, facilitates using such data-driven ML methods in simulation science [10]. On the other hand, the introduction of physics-informed neural networks (PINN) [11] enabled knowledge-driven ML methods to perform well on low data regime applications.

This thesis aims to improve the numerical simulation of a three-dimensional FSI problem. To do this, the resulting data of the simulation will be split up and a separate neural network will be trained on each partition. Parallelization of the training process will be applied as a speed-up technique. An approach incorporating overlapping of the partitions will also be examined to improve the prediction accuracy of the neural networks.

Chapter 2 describes the steps followed to produce the desired predictions of the simulation data. The results of these methods are analyzed an evaluated in chapter 3. Lastly, final thoughts and outlook are presented in chapter 4.

1.2 Motivation

Modelling the behavior of physical systems the classical way, i.e., using the mathematical equations that were derived from experiments, poses several drawbacks. For instance, some insight could be unidentified and therefore missing in these equations. This would lead to computations that might approximate the results of the performed experiments but don't extrapolate well to further simulations due to information that has not been integrated into these potentially simplified equations. Furthermore, the mathematical equations can demand complex numerical solvers which is connected to the issue of available resources not being sufficient to accommodate the required computational work.

These problems can be mitigated by using strictly data-driven ML methods which require access to data of physical experiments and simulations. By using all available information as training data set, all knowledge contained in these results is used to learn the ML model. This can lead to applications that lie outside of the performed cases that produced the training data as insight could be learned that wasn't previously known. Additionally, while the training of ML models can be very computationally demanding, the inference of the resulting fully trained models is very cheap. However, the main difficulty of this approach lies within the training process which has to be implemented such that it can yield predictions of sufficient accuracy. Consequently, this might limit the applicability of this method if the prediction accuracy isn't acceptable.

Lastly, there is the possibility of combining ML methods with classical solvers. While this approach is the most complex to implement, it also has the possibility to use the already known physical laws in junction with newly learned knowledge. This can improve the simulation performance by improving on the prediction accuracy. Moreover, by introducing an appropriate parallelization technique, we remove one of the obstacles of combining ML methods with classical parallel solvers.

1.3 Objectives

We use a three-dimensional FSI problem consisting of a fluid and a solid domain [12] to collect the required data to train the neural network with. The required data will be collected by solving the system of equations of the coupled FSI problem.

To account for mutual interaction of the fluid and the solid domain, and to avoid iterative solution procedure (in contrast to the approach presented by Totounferoush et. al [7]), we train only a single neural network for the whole domain. Since the respective problem is time-dependent, using recurrent neural networks (RNN) is necessary to learn the time dependency. In addition, convolutional layers can be used to account for the spatial connectivity imposed by the physical system of the problem. However, since the training data are collected from a simulation using unstructured grids, applying convolution is not easily feasible. As an alternative, more general fully connected feed-forward layers (FC) can be used.

There are problems however with FC layers with the main one being the large number of weights that they use. Since the problem domain is large, the training process does not fit into the memory of a single CPU. As a remedy, this project aims to use the parallelization technique introduced in [1] to parallelize the training.

Since this approach involves separating the data into partitions, the continuity on the interface of these has to be investigated in order to verify the applicability of the results. Accordingly, the possible discontinuities in partitions' intefaces must be handled appropriately. For this, improvements on the means of partitioning and handling each partition individually or combining several partitions can be implemented. For this purpose, we will investigate a partitions overlapping technique to avoid such issues.

2 Methodology

This chapter describes the individual steps that are performed for the implementation of this thesis. The very first procedure is the generation and aggregation of data from the FSI problem simulation which is specified in section 2.1. Next, in section 2.2, the architecture design of the deployed neural network is described as well as the refinements of their particular applications that are performed. Section 2.3 lines out the means of parallelizing the training of the neural network.

2.1 Data Collection

To train the neural network, training data is required. The steps followed to produce the training data set outlined in this section. First, section 2.1.1 describes the setup of the FSI problem simulation. This setup is used to produce the required training and inference data. Then, in section 2.1.2, the means of extracting the data from the simulation output as well as how the data is normalized, partitioned, and clustered into batches are shown to produce the final training data set.

2.1.1 Fluid-Structure Interaction Simulation

The FSI simulation test case provided by [13] is used for the data production. The tutorial that is used to set up the simulation can be found in [12]. This tutorial follows a partitioned approach where individual solvers for the fluid and solid sub-domains are used. In this approach, another library is necessary to couple these single-physics solvers. In the current test case, the OpenFOAM solver is used for the fluid and CalculiX for the solid domain. As explained by the tutorial, version 2112 of OpenFOAM is downloaded and installed from the OpenFOAM Ubuntu repository [14]. The coupling of the two solvers is performed by preCICE which is installed via its binary packages as described in [15]. Additionally, the adapters for each solver for preCICE had to be installed with the installation instructions of the OpenFOAM adapter located at [16] and the ones for the adapter for CalculiX in [17].

After having all required software installed and properly configured, the FSI simulation is run. The simulation consists of a total of 100 time steps. Each time step size equals 0.01 seconds which accumulates to a simulation period of 0.1 seconds. Initially, a pressure is prescribed at the tube inlet boundary that lasts for three time steps and propagates subsequently throughout the tube. This result can be described as a wave and visualized by creating an animation showing the deformation of the tube. The resulting files are in Visualization Toolkit (VTK) file format which is described in [18]. Since the data is arranged in an unstructured grid, the file extension `.vtu` is used which however doesn't change the way we handle the data as compared to `vtk` files. In figure 2.1, the 90th time step of the fluid and the solid participant is depicted in ParaView [19] with a warp by vector filter using a scale factor of ten added to the displacements of the points and coloring depending on the magnitude of the local point displacement.

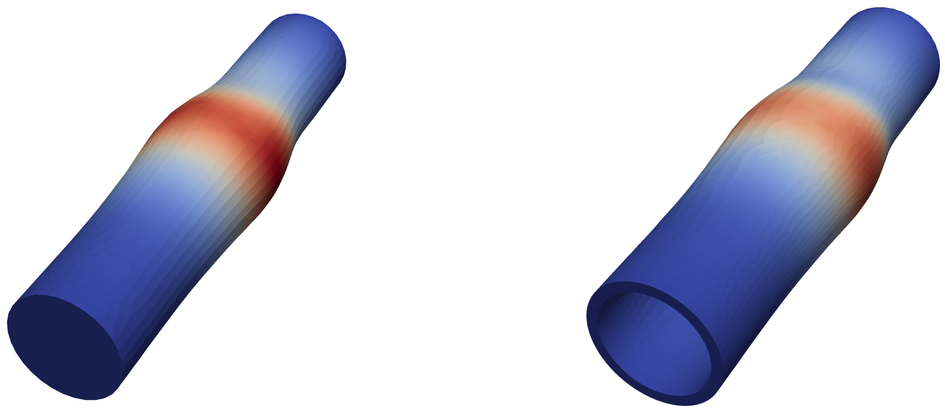


Figure 2.1: The original data of the fluid (left) and solid (right) participant of the FSI simulation at time step $t = 90$ with displacements scaled by a factor of ten and colored depending on the displacements.

2.1.2 Data Preprocessing

To obtain the data required to train the neural network, several steps have to be performed. These preprocessing steps are made up of data extraction, partitioning, normalization, and batching. In section 2.1.2.1, the initial way of obtaining the data from the simulation is described as well as the way we organize this data. Afterward, in section 2.1.2.2, the normalization of the data is explained which is a common practice when working with neural networks to speed up the training process. Next, in section 2.1.2.3, we explain how we implement the partitioning of the data, which is required later on for the parallelization technique from [1]. Lastly, in section 2.1.2.4, we describe our way of organizing the training data in batches which is a necessary step to later use recurrent neural network layers.

2.1.2.1 Data Extraction

For reading and later also writing of the `vtk` files, the python package meshio [20] is used. It provides the function `meshio.read()` that allows reading in the data of the `vtk` files as a mesh. This function is called for each time step and both participants, respectively, resulting in the 200 meshes being read into the memory. From these meshes, the required point data is extracted where the solid domain provides only displacement data for each point and the fluid domain produces displacement, force, and pressure data.

In total, the data is made up of 100 time steps t, each consisting of $14{,}670$ points p with 6598 points coming from the fluid domain and 8072 points of the solid domain. Each point has an initial location made up of three values for the x, y, and z directions, respectively.

$$p = (p_x, p_y, p_z)^\top.$$

Displacement data D_p^t is extracted from all points of both domains and since each point displacement consists of three values, one for each spatial dimension, the data looks as follows:

$$D_p^t = \left((D_x, D_y, D_z)^\top\right)_p^t,\ 0 \le p \le 14{,}669,\ 0 \le t \le 99.$$

Force data F_p^t is only extracted from the boundary points of the fluid dimension since the forces inside are not computed and therefore default to zero. Each force data point consists of three dimensions and with 1986 points used the data is organized as follows:

$$F_p^t = \left((F_x, F_y, F_z)^\top\right)_p^t,\ 0 \le p \le 1985,\ 0 \le t \le 99.$$

Finally, the pressure data P_p^t is extracted from each of the 6598 points of the fluid domain and consists only of single values leading to the following shape:

$$P_p^t,\ 0 \le p \le 6597,\ 0 \le t \le 99.$$

In total, there are $5{,}665{,}500$ data values to train the neural network on with each time step t of data organized in a single vector:

$$v^t = \begin{pmatrix} D_0^t \\ D_1^t \\ \vdots \\ D_{14{,}669}^t \\ F_0^t \\ F_1^t \\ \vdots \\ F_{1985}^t \\ P_0^t \\ P_1^t \\ \vdots \\ P_{6597}^t \end{pmatrix}$$

.

2.1.2.2 Normalization

Normalization of the data is performed linearly on all data points over all time steps. We used separate normalization maximums for displacement, force, and pressure data, respectively. Since the displacement data is made up of positive and negative values, normalization is carried out to the interval $[-1,\ 1]$ by dividing each value of each displacement D_p^t of partition p and time step t by the displacement maximum $D_{\max}$ which yields the normalized displacement data $\hat{D}_p^t$.

$$\begin{aligned} D_{\max} &= \max\left(\left\{\left|(D_d)_p^t\right| \;\middle|\; d \in \{x,\ y,\ z\},\ 0 \le t \le 99,\ 0 \le p \le 14{,}669\right\}\right) \\ \hat{D}_p^t &= \left(\frac{(D_x)_p^t}{D_{\max}}, \frac{(D_y)_p^t}{D_{\max}}, \frac{(D_z)_p^t}{D_{\max}}\right)^\top \end{aligned}$$

Normalization of the force data is performed in the same manner as previously outlined for the displacement data with its separate maximum value and a smaller number of points.

$$\begin{aligned} F_{\max} &= \max\left(\left\{\left|(F_d)_p^t\right| \;\middle|\; d \in \{x,\ y,\ z\},\ 0 \le t \le 99,\ 0 \le p \le 1985\right\}\right) \\ \hat{F}_p^t &= \left(\frac{(F_x)_p^t}{F_{\max}}, \frac{(F_y)_p^t}{D_{\max}}, \frac{(F_z)_p^t}{F_{\max}}\right)^\top \end{aligned}$$

Finally, pressure data is normalized to the interval $[0, 1]$ as this data doesn't contain any negative values but the general normalization computations stay the same.

$$P_{\text{max}} = \max\left(\left\{P_p^t \,\middle|\, 0 \leq t \leq 99,\ 0 \leq p \leq 6597\right\}\right)$$
$$\hat{P}_p^t = \left(\frac{P_p^t}{P_{\text{max}}}\right)$$

2.1.2.3 Partitioning

The partitioning of the data is required to utilize the parallelization technique described in [1].

The approach of partitioning the data is based on the idea of cutting up the tube with its solid and fluid domain into slices and treating each slice as its separate partition that a neural network is assigned to. A visual example of this partitioning technique is depicted in figure 2.2. To achieve this, first, the partition allocation has to be created, which holds a value for each data point containing the assigned partition for that point. Each partition spans the same length in space along the z-axis of the tube, i.e., the direction the tube is oriented in. Due to the unstructured nature of the grid in that the data points are arranged, the partitions have differing numbers of values. In particular, the first and last partitions contain the highest number of data points.

To create the partition allocation, first the values of only the z-axis of the data points is extracted. From this array, the physical length in the z-direction of a partition l_{part} is calculated by dividing the difference of the greatest and the smallest z-axis value by the number of partitions n_{part}.

$$l_{\text{part}} = \frac{(p_z)_{\text{max}} - (p_z)_{\text{min}}}{n_{\text{part}}}$$

Now, the partition allocation a_p for each point p is calculated as

$$a_p = \min\left(\frac{\lfloor p_z - (p_z)_{\text{min}} \rfloor}{l_{\text{part}}},\ n_{\text{part}} - 1\right).$$

This results in each point p being allocated to a partition a_p such that

$$a_p \leq \frac{p_z}{l_{\text{part}}} \leq a_p + 1.$$

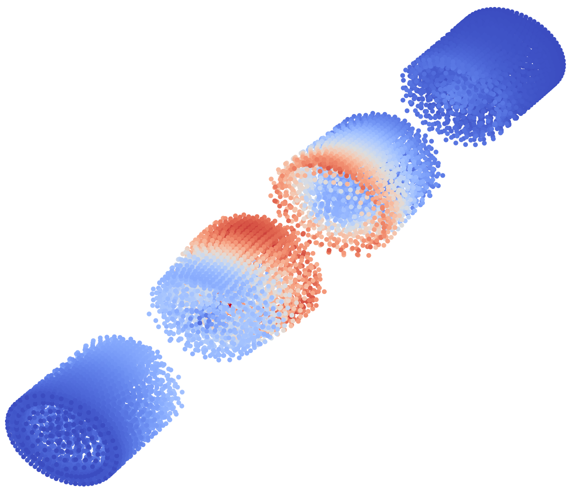


Figure 2.2: A visual example of the partitioned tube at $t = 90$ with four spatially separated partition slices. Each point is plotted with its displacement scaled by a factor of ten and colored according to the respective magnitude of the displacement.

2.1.2.4 Batching

The final step of the preprocessing of the data is batching which creates the batches B^n consisting of multiple subsequent time steps. This is required for the recurrent neural network layer as the input is made up of three time steps $(t-2,\ t-1,\ t)$ whereas the output is only the following time step $(t+1)$. This yields a total of 97 batches of the following shape:

$$B^n = \begin{pmatrix} D^n & D^{n+1} & D^{n+2} \\ F^n & F^{n+1} & F^{n+2} \\ P^n & P^{n+1} & P^{n+2} \end{pmatrix} = \begin{pmatrix} B_0^n & B_1^n & B_2^n \end{pmatrix},\ 0 \le n \le 97.$$

2.2 Machine Learning

After preparing the training data, the neural network can start the learning process. For the ML element of this thesis, we use the open-source ML framework PyTorch [21]. This section covers the architecture of the used neural network as well as the optimizations that are performed to improve the accuracy of the predictions. In section 2.2.1, the general neural network model that is deployed for each partition is described. After that, in section 2.2.2, we specify the technique of using overlapping partitions as input to the neural network to enhance the predictions.

2.2.1 Single Partition Neural Network

This section covers the implementation of the neural network model and training. In section 2.2.1.1, the architecture of the neural network is described and the process of how data is inferred through the network is detailed. Afterward, in section 2.2.1.2, the training process is shown. Lastly, in section 2.2.1.3, the inference method is described, as well as the postprocessing steps.

2.2.1.1 Neural Network Model Architecture

The general outline of the architecture used for the neural network model is depicted in figure 2.3. All hidden layers are of the same number of dimensions as the length of the desired final output vector. We first pass the input data batch B_0 through five fully connected layers $\ell \in [0,\ 4]$ using `torch.nn.Linear()` [22]. As activation function we apply the rectified linear unit (ReLU) with `torch.nn.relu()` [23] which is defined as

$$\text{ReLU}(x) = (x)^+ = \max(0, x).$$

This yields B_5 with weights W_ℓ and biases b_ℓ.

$$B_{\ell+1} = \text{ReLU}\left(B_\ell W_\ell^\top + b_\ell\right),\ 0 \le \ell \le 4.$$

Afterward, the data gets passed through the recurrent neural network layer using `torch.nn.RNN()` [24]. Here, PyTorch deploys a multi-layer Elman recurrent neural network that was first proposed in [25]. We're using three recurrent layers $\ell \in [5, 7]$, i.e., three recurrent neural networks stacked together where each layer takes the output of the previous one as input. Again, we apply ReLU as non-linearity between layers. Each layer takes the three time steps of each batch time stepwise and computes the hidden states $h^1,\ h^2,\ h^3$, for the next layer or as the output of the recurrent neural network. This procedure is depicted in figure 2.4 where the handling of a batch of three time steps by a recurrent neural network is shown. This yields B_8 with weights on the input W_i, weights on the hidden state W_h, biases on the input b_i, and biases on the hidden state b_h.

$$\begin{aligned} h_{\ell+1}^{t+1} &= \text{ReLU}\left(B_\ell^t W_i^\top + b_i + h_\ell^t W_h^\top + b_h\right),\ 5 \le \ell \le 7,\ 0 \le t \le 2 \\ B_8 &= \begin{pmatrix} h_8^1 & h_8^2 & h_8^3 \end{pmatrix}. \end{aligned}$$

To get the output of the recurrent layer into the desired output shape of one single time step, we deploy a final single fully connected layer $\ell = 8$. Prior to this however, we reshape B_8, a two-dimensional matrix containing a vector for each time step, into $\hat{B}_8$, a single vector with the three time steps stacked on top of one another, using `torch.reshape` [26].

$$\hat{B}_8 = \begin{pmatrix} h_8^1 \\ h_8^2 \\ h_8^3 \end{pmatrix}$$

$$B_9 = \hat{B}_8 W_8^\top + b_8.$$

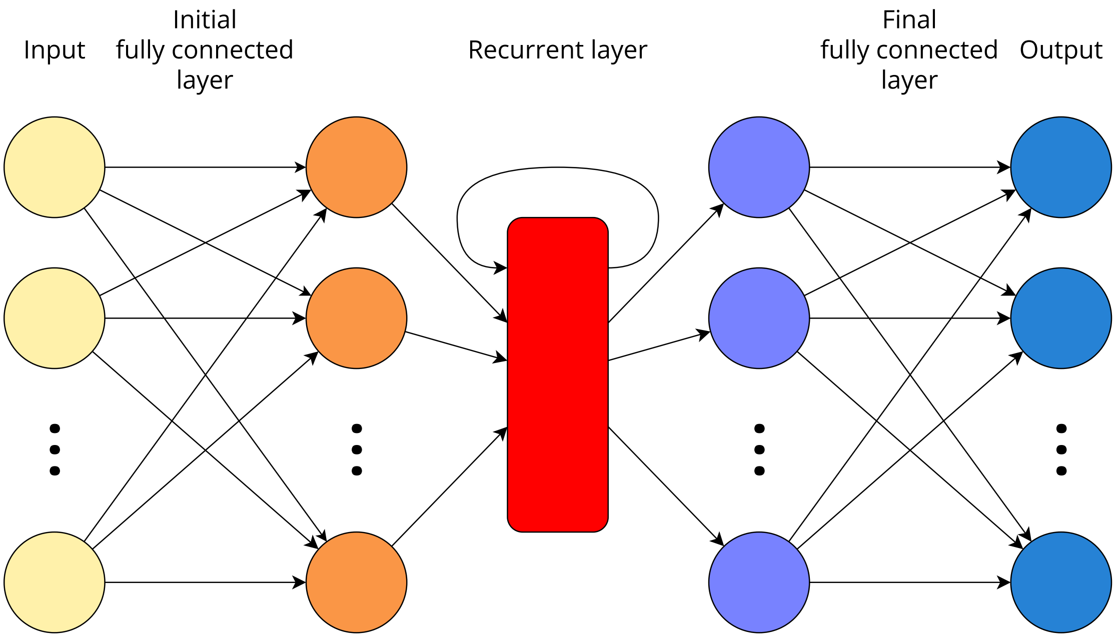


Figure 2.3: A general outline of the architecture of the neural network model. First, the input data is passed through several fully connected layers. Then, the batch of three time steps is given to the recurrent layer time step by time step. Finally, the data gets reshaped to the desired output dimensions in the last fully connected layer.

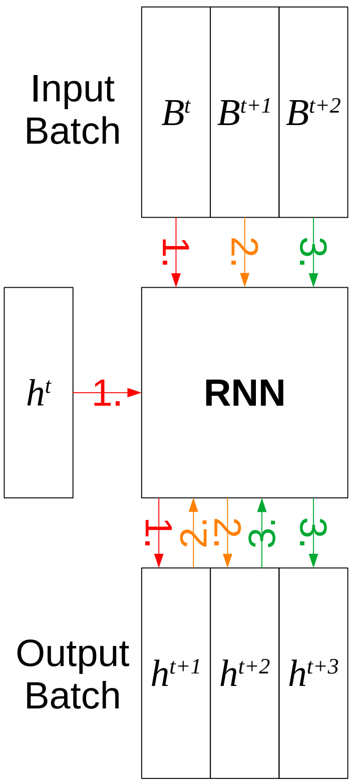


Figure 2.4: Method of processing data through a recurrent neural network (RNN). The input batch B with time steps $t,\ t+1,\ t+2$ is provided to the RNN time stepwise with their respective hidden states h and h^t being the initial hidden state. The colors of the arrows signal the recurrent steps performed by the RNN layer. The output batch is constructed from the three predicted hidden states $h^{t+1},\ h^{t+2},\ h^{t+3}$.

2.2.1.2 Training

For the training of the single partition neural network we use `torch.nn.MSELoss()` as the criterion, which computes the loss $L(x, y)$ as the mean squared error (MSE) of input x and target y.

$$L(x, y) = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - y_i)^2 \, .$$

The Adam algorithm, as introduced in [27] and implemented via `torch.optim.Adam()` [28], is used as the chosen optimizer of the training process. Here, we also define the learning rate `lr` and the L2 regularization as `weight_decay`.

$$\begin{aligned}\texttt{lr} &= 10^{-5}\\ \texttt{weight\_decay} &= 10^{-6}.\end{aligned}$$

The training is executed with a set number of epochs. Since we use different machines to perform the training, we adapt the number of epochs depending on the time it takes to carry out the entire training. For training runs that test the accuracy of the results, we use a CUDA-enabled [29] machine and used 10^4 epochs, while for training runs testing the parallelization implementation, we use the neon machine (s. table 2.1) with a significantly lower number of training epochs.

	CUDA-enabled machine	**neon**
CPU	AMD Ryzen 5 3600X	Intel Haswell
GPU	Nvidia GeForce RTX 2070 Super	-
RAM	16GB	504GB

Table 2.1: The specifications of the machines used in the training runs.

2.2.1.3 Sampling

After the training on each partition of data is completed, the inference step of the implementation is launched. For this, the very first batch of time step data for a given partition is given to the trained neural network model of that respective partition. The model now produces the succeeding time step of that partition. This time step data is then appended to the used batch, while the first time step is omitted such that the batch consists of three consecutive time steps again. Figure 2.5 visualizes the method of inference. This process is now repeated until all time steps are predicted. To return the data to the non-normalized domain, we post-process each data point p of every time step t by multiplying its predicted normalized values by the respective data maximum.

$$\begin{aligned}D_p^t &= \left((\hat{D_x})_p^t \cdot D_{\max},\ (\hat{D_y})_p^t \cdot D_{\max},\ (\hat{D_z})_p^t \cdot D_{\max}\right)\\ F_p^t &= \left((\hat{F_x})_p^t \cdot F_{\max},\ (\hat{F_y})_p^t \cdot F_{\max},\ (\hat{F_z})_p^t \cdot F_{\max}\right)\\ P_p^t &= \left(\hat{P}_p^t \cdot P_{\max}\right)\end{aligned}$$

During this normalization process, the data for each point is also returned to their original location in the data array that was previously lost due to the partitioning step. Finally, the predicted values are written out as `vtk` files using `meshio.Mesh()` to create the mesh and `meshio.write()` to write the mesh to the `vtk` file.

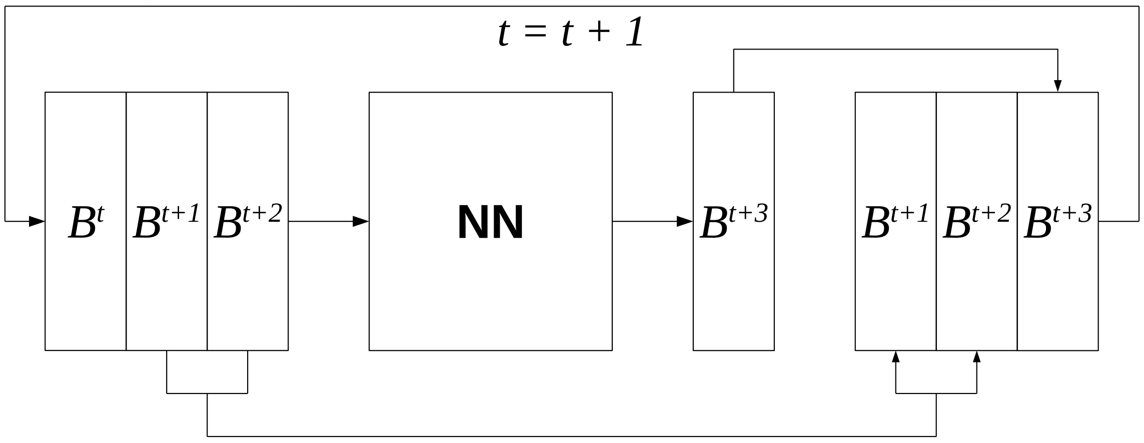


Figure 2.5: Depicted is the inference of one time step $t+3$ through the neural network given the batch consisting time steps $t,\ t+1,\ t+2$. The resulting time step is then appended to the latter two time steps of the input batch.

2.2.2 Overlapping Partitions

Due to the partitioning and especially separate and isolated handling of each partition on their own, the neural networks don't share any information with one another. This doesn't act as a problem for partitions close to the inlet of the tube, as the first three time steps of the simulation already introduce the initial impulse. However, partitions further down the tube, that lie outside of that range, don't receive any information as to when, i.e., at which time step, they're meant to continue the pressure wave. To combat this issue of missing information on the propagating wave, we explore the concept of implementing overlapping partitions. First, in section 2.2.2.1, we detail the initial approach of fully overlapping adjacent partitions to train the neural network. Afterward, in section 2.2.2.2, we describe the more sophisticated approach of only partially overlapping partitions.

2.2.2.1 Full Overlap

For the implementation of fully overlapping partitions, we increase the size of the input tensor given to each neural network. This is done by stacking the vectors of the preceding ($a_p - 1$) and following ($a_p + 1$) partition of each partition a_p with the vectors of its partition. For the first ($a_p = 0$) and last ($a_p = n_{\text{part}}$) partitions, only the following or preceding partitions are added, respectively.

$$
\begin{aligned}
B^0 &= \begin{pmatrix} B_0^0 & B_1^0 & B_2^0 \\ B_0^1 & B_1^1 & B_2^1 \end{pmatrix} \\
B^p &= \begin{pmatrix} B_0^{a_p-1} & B_1^{a_p-1} & B_2^{a_p-1} \\ B_0^{a_p} & B_1^{a_p} & B_2^{a_p} \\ B_0^{a_p+1} & B_1^{a_p+1} & B_2^{a_p+1} \end{pmatrix}, \ 1 \le a_p < n_{\text{part}} \\
B^{n_{\text{part}}} &= \begin{pmatrix} B_0^{n_{\text{part}}-1} & B_1^{n_{\text{part}}-1} & B_2^{n_{\text{part}}-1} \\ B_0^{n_{\text{part}}} & B_1^{n_{\text{part}}} & B_2^{n_{\text{part}}} \end{pmatrix}
\end{aligned}
$$

Doing this significantly increases the number of values in the input tensor and therefore also the size of the neural network model. Also, since this approach doesn't treat each partition completely separate from one another anymore, it's not possible to compute inference for just one partition at a time. Rather, we have to calculate the predictions time stepwise, such that each partition can receive the results of their neighbors to compute the next time step.

2.2.2.2 Partial Overlap

To decrease the size of the input tensor while still maintaining the additional information, we explore the idea of partially overlapping the partitions. The visualization of this approach is depicted in figure 2.6, where 4 partitions are shown with the overlapping areas colored in red for the back and blue for the front of the partition's first and last 20% of length in the z-direction. This is achieved by first computing separate partition allocations for the front and back parts of each partition. Thus, we calculate for each point p by its z-value p_z whether it's either part of the front (a_f) or back (a_b) of its partition with the overlapping percentage o and partition length l_{part}.

$$
\begin{aligned}
\frac{p_z \bmod l_{\text{part}}}{l_{\text{part}}} &\le o \rightarrow a_f = a_p \\
\frac{p_z \bmod l_{\text{part}}}{l_{\text{part}}} &\ge 1 - o \rightarrow a_b = a_p
\end{aligned}
$$

After these additional partition allocations are computed, the batches are assembled similarly as for the fully overlapping partitions. The difference is, that only points of the previous and following partition are used, if they are in the front or back of their partition slice, respectively.

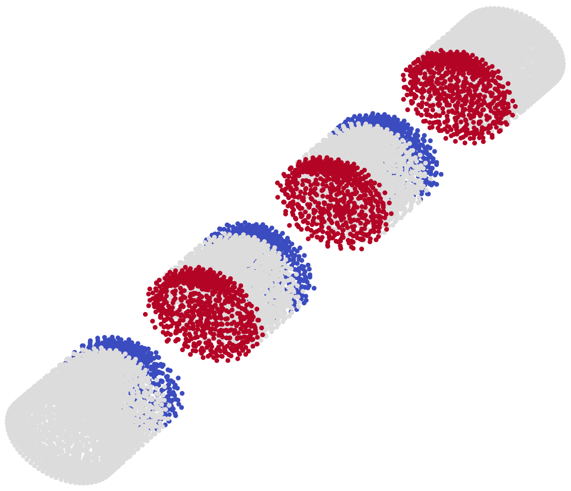


Figure 2.6: Visualization of partial overlapping of 20% on four partitions where red points are in the overlapping area for the previous partition and blue points are overlapped to the succeeding partition.

2.3 Parallelization

For the parallelization implementation of this thesis, we use the Python package MPI for Python (`MPI4py`) [30]. Due to the initial partitioning of the data, parallelization is implemented in a very straightforward way. First, the main process, where `rank` $= 0$, reads, normalizes, and allocates all the data in the same manner as previously realized. For the partition allocation, we use the same number of partitions as there are threads.(`size`):

$$n_{\text{part}} = \texttt{size}.$$

Then, the partition data is distributed across the threads by calling `comm.scatter()` for the displacement, force, and pressure data, respectively. Figure 2.7 depicts this distribution for four partitions, where each partition $1 \leq a \leq 4$ is given to its own thread p_a. Now, each thread carries out the training in parallel on its given partition data in the same way as it was done sequentially. After the training, the inference and the subsequent postprocessing of the prediction data are performed by each thread in parallel. Only now the implementation returns to sequential actions, as the main thread collects the prediction data of each sub-thread by using `comm.gather()`. Finally, it returns the data to the order it was in before the partitioning step, such that it can write out the predictions in `vtk` files.

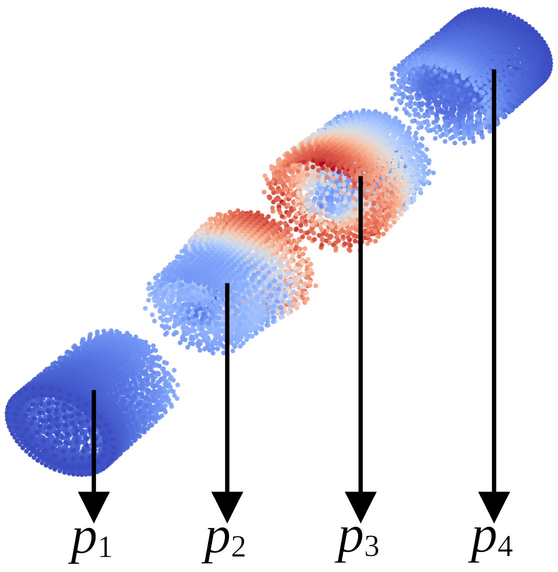


Figure 2.7: Distribution of four partitions to their respective thread $p_n,\ 1 \leq n \leq 4$.

Since this concurrent implementation requires the entire data, including input and target batches together with the neural network models, to be held in memory at the same time, it is not possible to use the CUDA-enabled machine shown in table 2.1. Instead, for testing this parallelization, we use the neon machine which provides plenty of memory space to carry out the training in parallel.

3 Results

This chapter covers and discusses the results that are produced by our applied methods. First, section 3.1 describes the achieved accuracy of the machine learning and partitioning approaches. Section 3.2 evaluates the outcome of our implementation for parallelizing the training process.

3.1 Accuracy

This section details the results of the neural network predictions in terms of the achieved accuracy. Section 3.1.1 compares neural networks of different hyper-parameters to justify the neural network structure used. Afterward, in section 3.1.2, we compare the achieved improvements in prediction accuracy when utilizing various percentages of overlap between partitions.

3.1.1 Neural Network Hyper-Parameters

For the comparison of hyper-parameters, we focus on two aspects of the neural network. First, there is the number of layers used for each part of the network which will be detailed in section 3.1.1.1. Secondly, in section 3.1.1.2, we look at the L2 regularization introduced via the `weight_decay` parameter of the PyTorch optimizer class `Adam`. Figure 3.1 depicts the comparison of these aspects of the neural network. Each combination of layering and regularization utilization is measured by the final MSE (blue) and total training time in seconds (orange). The training runs for these comparisons were performed on the CUDA-enabled machine (table 2.1). Each training run is performed for 1000 epochs and uses a learning rate of 10^{-5}.

3.1.1.1 Layers

As previously established, the neural network is structured into three parts (figure 2.3).

1. initial fully connected layers
2. recurrent layers
3. final fully connected layers

For the layering of the neural network, we first have the single layer setup, where each part of the network only contains one layer. The setup using multiple layers contains five layers for the first, three layers for the second, and one layer for the final part.

For the single layer variant, we achieve an MSE of about 2.01×10^{-4} with an average training time of around 334 seconds. In contrast, using multiple layers, the final MSE is circa 0.82×10^{-4} and the required training time is roughly 661 seconds. The observation in consideration of the different layering we can make is the inverse correlation of MSE and training time, i.e., the training runs that took more time to complete the 1000 epochs achieved a lower MSE and vice versa. This is to be expected, as the multiple layer setup of the neural network performs more optimization steps per epoch and thereby takes more time while lowering the MSE to a greater extent than the single layer variant. Our later training runs perform more epochs than these tests, which promises to approach convergence more closely. As we expect the minimum achievable MSE to be lower with multiple layers for each part of the neural network, we use this setup for later training runs.

3.1.1.2 Regularization

The tested regularization parameters are `weight_decay` $= 0$ for no regularization and `weight_decay` $= 10^{-6}$ for the runs utilizing regularization. If regularization proves to be applicable for the data used, it has the potential to reduce overfitting and, in our case, reduce high local error occurring in some points of the predictions.

The training time appears to be slightly affected detrimentally by the use of regularization as it increased the total training time by about 4.7%. As for the MSE, regularization doesn't seem to be influential in any consistent way. Due to these findings, we chose to keep using regularization because the effect in training time is only marginal while the potential to avoid local spikes in error is promising for later training runs.

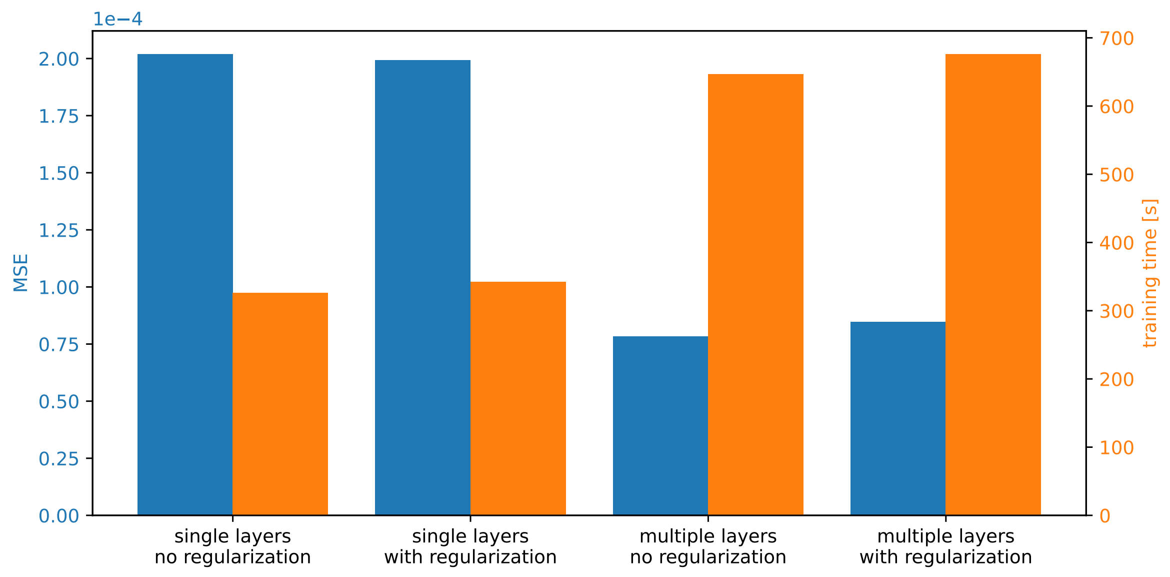


Figure 3.1: Comparison of the final MSE (blue) and total training time (orange) of neural networks using various hyper-parameters. Layering is differentiated by either only involving a single layer for the initial fully connected, the recurrent, and the final fully connected parts, respectively, or consisting of five initial fully connected layers, three recurrent layers, and one final fully connected layer. For runs with or without regularization a weight decay of 10^{-6} or 0 is used, respectively.

3.1.2 Overlapping Partitions

This section covers the results produced by the approach utilizing overlapping partitions in comparison to the initial method of keeping the partitions completely separate from one another. All training runs are performed with $10{,}000$ epochs and a learning rate of 10^{-5} As detailed in section 3.1.1.2, L2 regularization is deployed by setting `weight_decay` $= 10^{-6}$. The used percentages to determine the overlapping area are 5%, 10%, 20%, and 100%, respectively. Here, 100% is the initial technique of working with overlapping partitions, which uses the entire data set of neighboring partitions as input to the neural network. Figure 3.2 depicts the comparison in terms of MSE (blue) and total training time (orange) of different percentages of overlap. This is more closely explored in section 3.1.2.1. Afterward, in section 3.1.2.2, the predictions of the different overlapping percentages are compared by the continuity of the displacements from partition to partition.

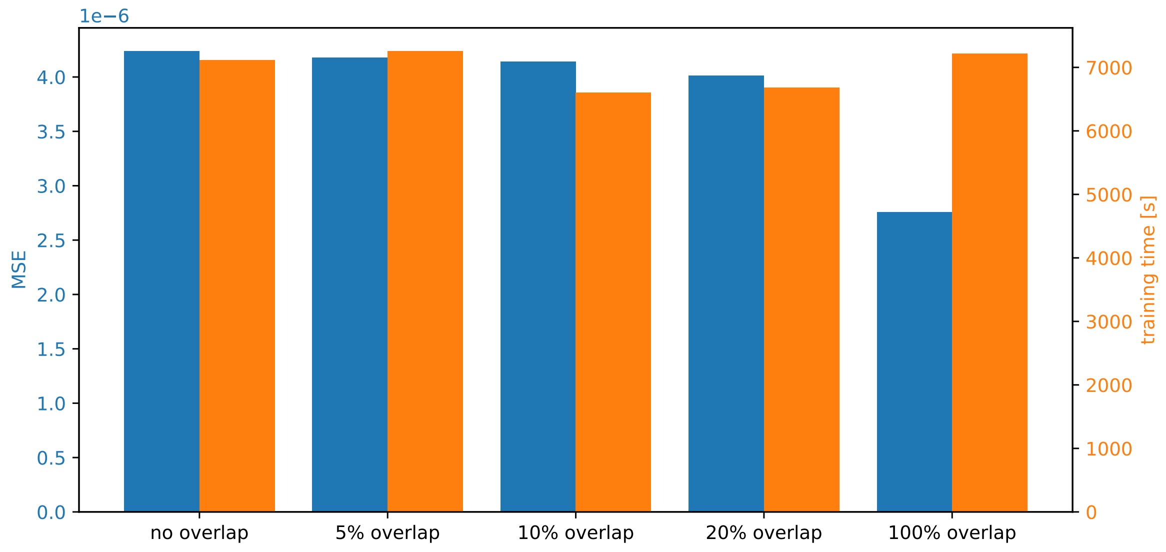


Figure 3.2: Comparison of the final MSE (blue) and total training time (orange) of neural networks using different percentages of overlap.

3.1.2.1 MSE & Training Time

Concerning training time, there is no significant change in any direction when increasing the overlap. This could be attributed to the number of dimensions in the hidden layers of the neural networks. Since these are set equal to the size of the output vector, the size of the vectors in the input batch does not pose a serious growth in the number of calculations performed per epoch. For example, the input batch of the fully overlapping approach is about three times as big as the input batch of the method using no overlap. This only affects the very first layer of the neural network, as here three times as many weights and biases are required now, but all other layers stay of the same size. With this possible explanation, the fluctuation in training time might be attributed mainly to inconsistencies in load or scheduling.

On the other hand, regarding the MSE, there is a visible decline as the overlap percentage increases. In fact, MSE is decreasing strictly monotonous with a greater overlap, i.e., each training run using more overlapping area produced a lower MSE than any run with less overlap. Especially when comparing the 20% to the full overlap approach, the difference in MSE is visible. As MSE is not subjected to fluctuations of load on the machine, in contrast to training time, the increase in neuron connections in the very first layer of the network might influence the resulting MSE directly in a positive way. These results indicate that increasing the number of dimensions in the hidden layers could prove to be helpful to lower the MSE further.

3.1.2.2 Continuity

Due to the partitioning of the data set, we also examine for possible discontinuities at the interface of partitions, since each partition is handled separately. Figure 3.3 depicts plots for comparison created with ParaView of the original simulation data (top left), as well as predictions using the various overlapping percentages of 0% (top right), 5% (center left), 10% (center right), 20% (bottom left), and 100% (bottom right. For these, only the solid domain is used at time step $t = 90$ and a warp by vector filter of a scale factor of 10 is applied to the displacements. What can be noticed, is the poor prediction when using no overlap in contrast to the greatly improved accuracy of the plots using even small overlap amounts. While there is a visible jump in quality going from 0% to 5% overlap, there is only a very marginal improvement noticeable when increasing the overlapping area further. This suggests that some overlap is required to train the neural network appropriately.

Figures 3.4, 3.5, and 3.6 show a comparison of the various overlapping prediction variants ($0\%,\ 5\%,\ 10\%,\ 20\%,\ 100\%$) against the original simulation data for time steps $t \in \{10,\ 50,\ 90\}$, respectively. On these plots, it's clear that the non-overlapping predictions are vastly inferior to any overlapping prediction. Especially when moving forward in time, the discrepancy becomes greater while, in contrast, the predictions with overlap only differ slightly from the original data. Furthermore, there's only very little difference in the predictions with overlap. This again suggests that some overlap may be necessary for predictions of high quality in accuracy.

3.2 Parallelization

For parallelization, we compare three variants of deploying multithreading in the training process. On the one hand, we use our parallel implementation which handles the concurrency with MPI4py [30] and treats each partition of the tube as its thread. Then, there is the multithreading provided by PyTorch which can be controlled through `torch.set_num_threads()` [31]. Lastly, we also take a hybrid approach to the comparison where parallel partitions and PyTorch's multithreading are used to the same degree, i.e., using n parallel partitions each with n threads available to PyTorch resulting in n^2 threads in total. All training runs are performed on the neon machine 2.1 with 1000 epochs.

Figure 3.7 shows the total training time each parallelization variant required to finish. We can see that for PyTorch's multithreading there seems to be a limit as to the number of CPU cores that can be utilized to decrease the training time. For the runs using up to 16 threads the time gets lowered but increasing the number of available cores does not further reduce the training time. For the parallel partitions and the hybrid approach, this does not occur, as each training run using more threads also required less time for the training.

In figure 3.8, the speedup $s = \frac{t_1}{t_n}$ of each variant is plotted for the various numbers of threads, where t_n is the training time using n threads. The multithreading of PyTorch is seen to gain only for a small number of threads and then flattens off when making more threads available. For the other two variants, there is also a non-linear relationship of speedup to the number of threads, however, even for a greater number of threads, an increase in speedup can be observed with only a little difference between them.

Finally, figure 3.9 compares the efficiency $e = \frac{s_n}{n}$ of each parallel implementation where s_n is the speedup for n threads. Again, PyTorch's multithreading shows a weaker capability of scaling well for a higher number of threads, as the efficiency drops to only about 8% when using 64 threads. On the other hand, the parallel partitions and hybrid approach achieve efficiencies of 32% and 34%, respectively.

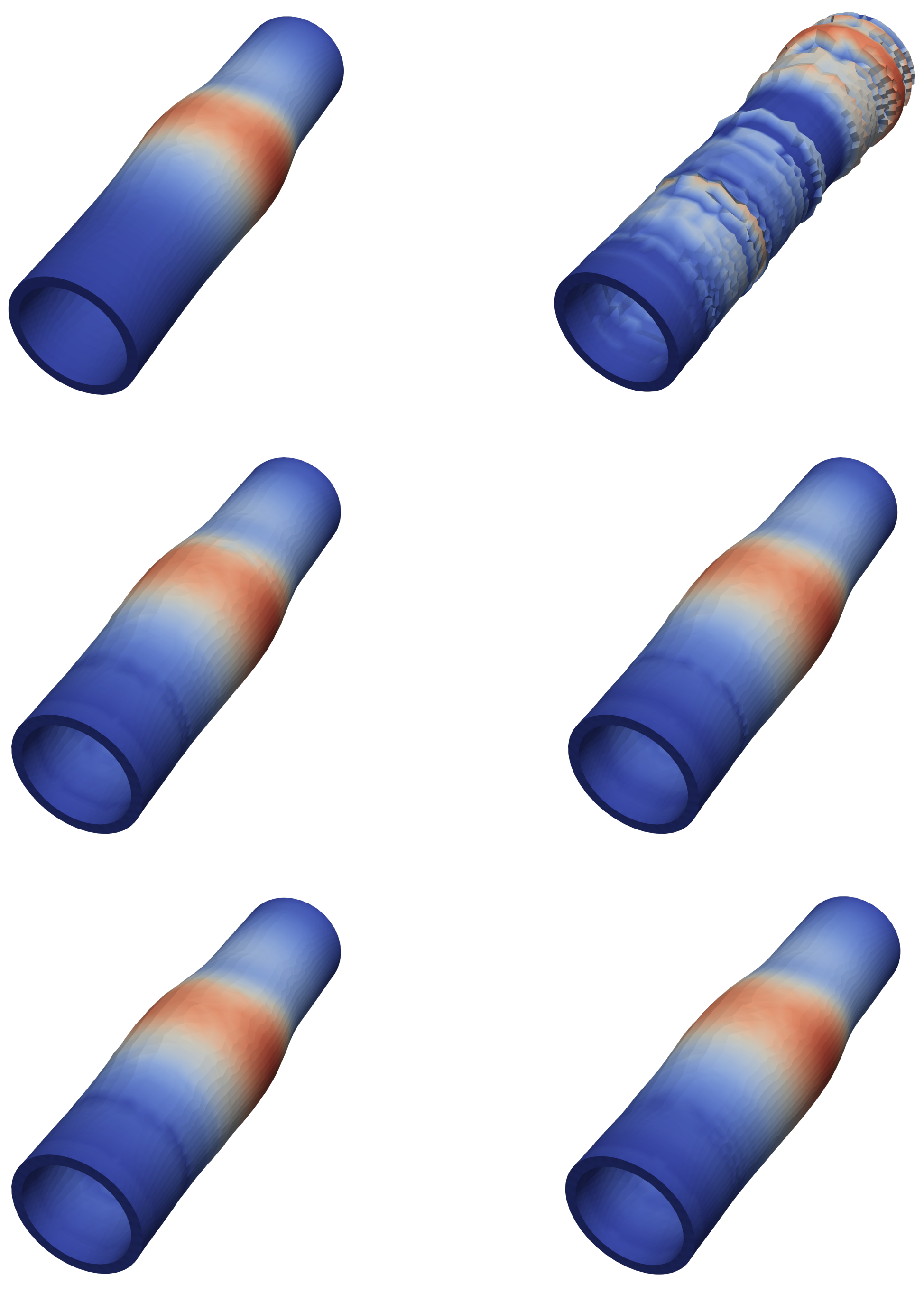
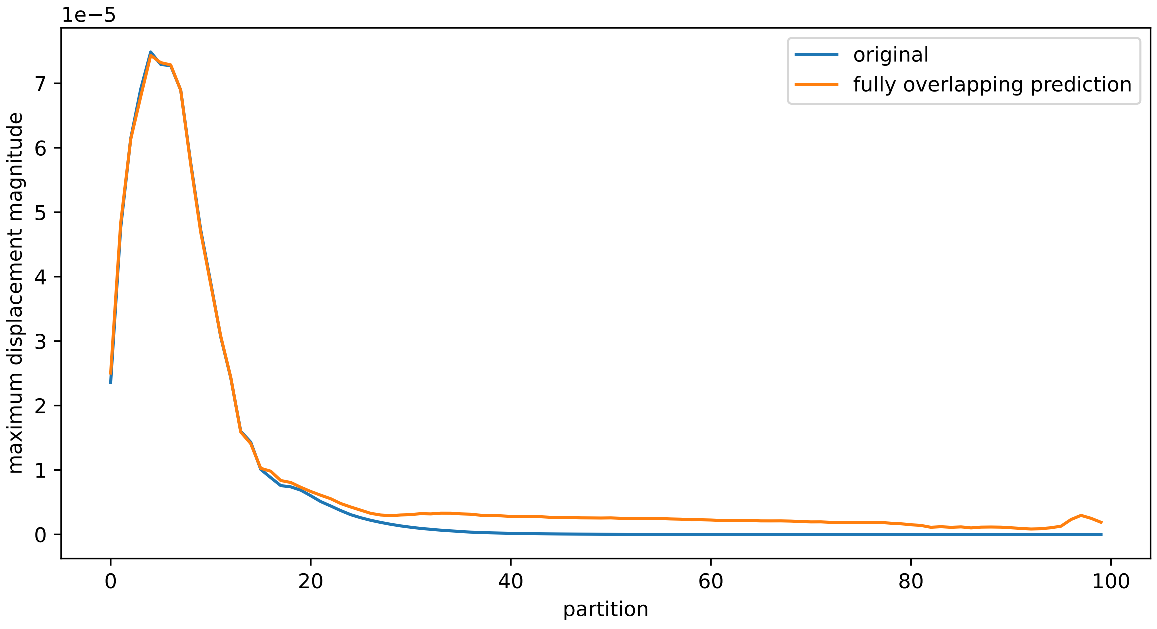


Figure 3.3: ParaView plots of original data (top left), 0% (top right), 5% (center left), 10% (center right), 20% (bottom left), and 100% (bottom right) overlap area, respectively. All plots are of the solid domain at time step $t = 90$ with a warp by vector filter using a scale factor of 10 on the displacements.



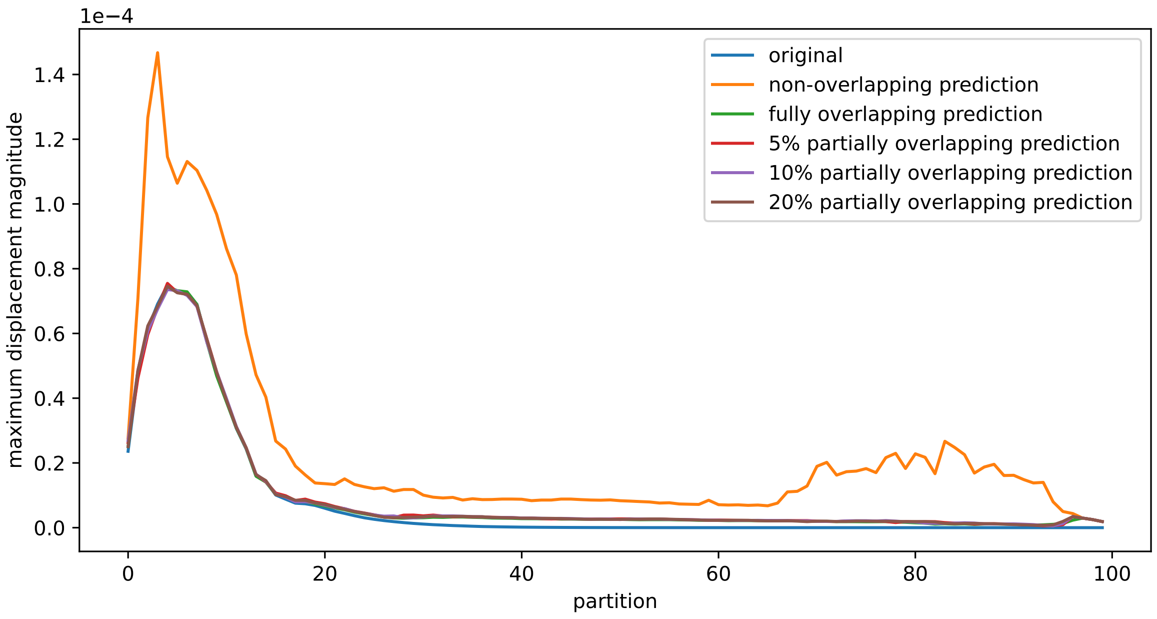
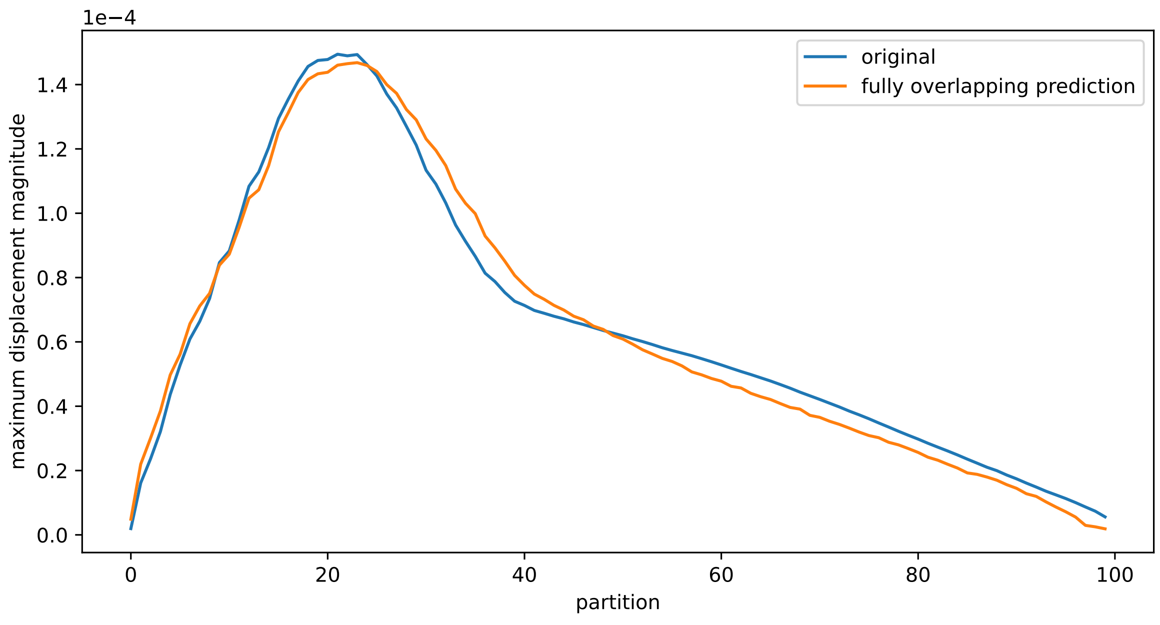


Figure 3.4: Comparison of displacements at time step $t = 10$ for the original simulation data and the non-overlapping, fully overlapping, and 5%, 10%, and 20% partially overlapping predictions, respectively. The top plot shows only the original and fully overlapping prediction data while the bottom one compares all variants. Partitions of the tube lie on the x-axis with the highest absolute displacement of any point per partition plotted against on the y-axis.



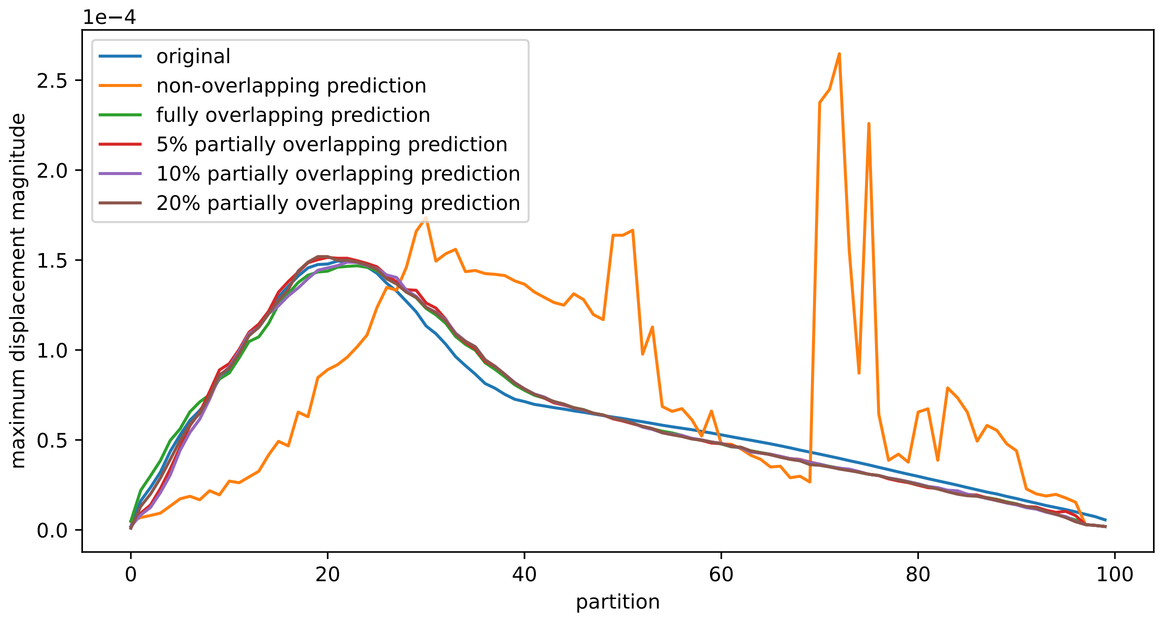
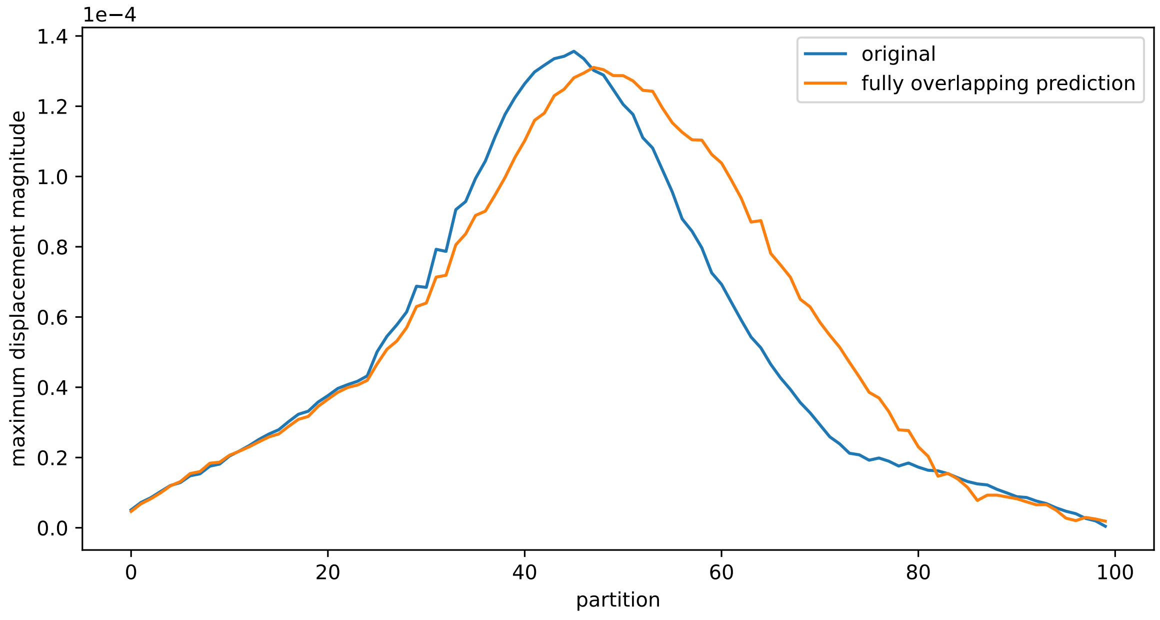


Figure 3.5: Comparison of displacements at time step $t = 50$ for the original simulation data and the non-overlapping, fully overlapping, and 5%, 10%, and 20% partially overlapping predictions, respectively. The top plot shows only the original and fully overlapping prediction data while the bottom one compares all variants. Partitions of the tube lie on the x-axis with the highest absolute displacement of any point per partition plotted against on the y-axis.



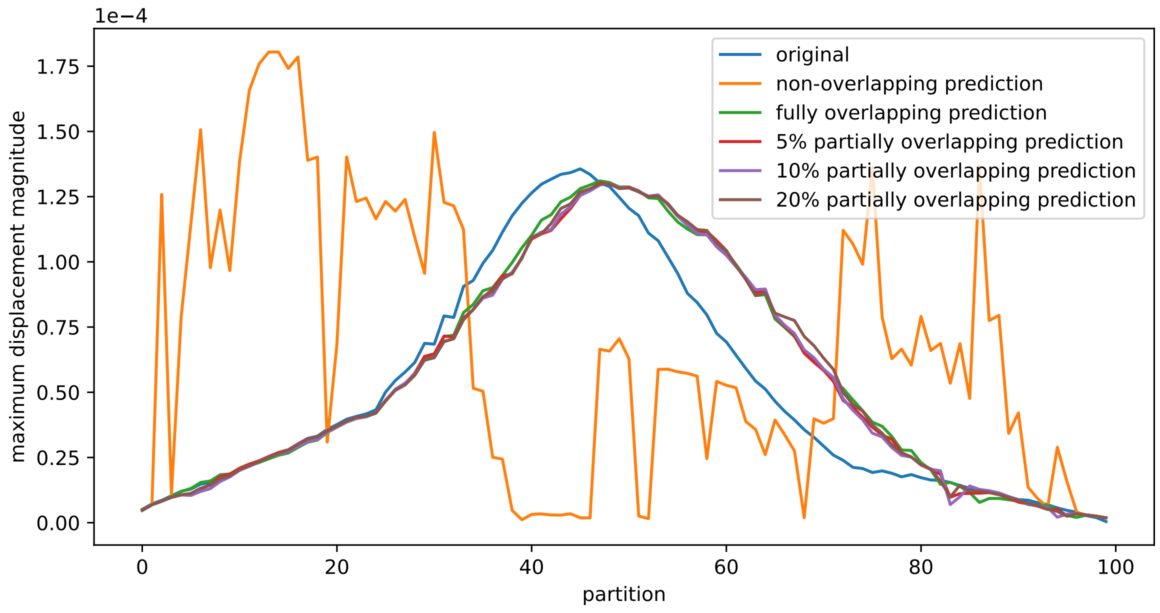


Figure 3.6: Comparison of displacements at time step $t = 90$ for the original simulation data and the non-overlapping, fully overlapping, and 5%, 10%, and 20% partially overlapping predictions, respectively. The top plot shows only the original and fully overlapping prediction data while the bottom one compares all variants. Partitions of the tube lie on the x-axis with the highest absolute displacement of any point per partition plotted against on the y-axis.

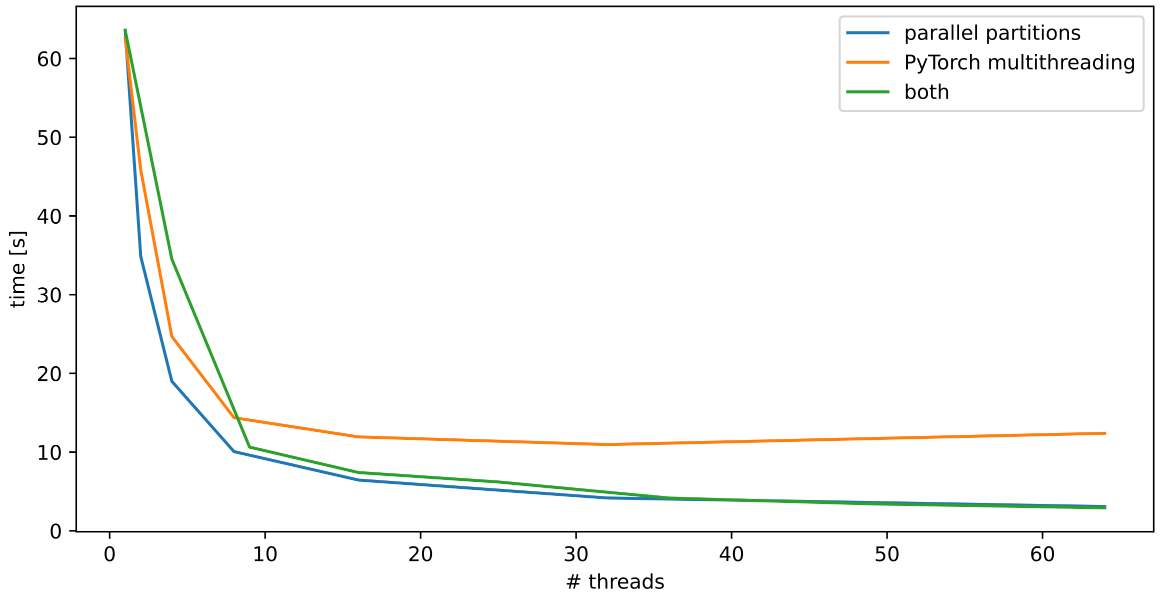


Figure 3.7: Training time in seconds of training partitions in parallel, multithreading of PyTorch, and using both implementations to the same degree.

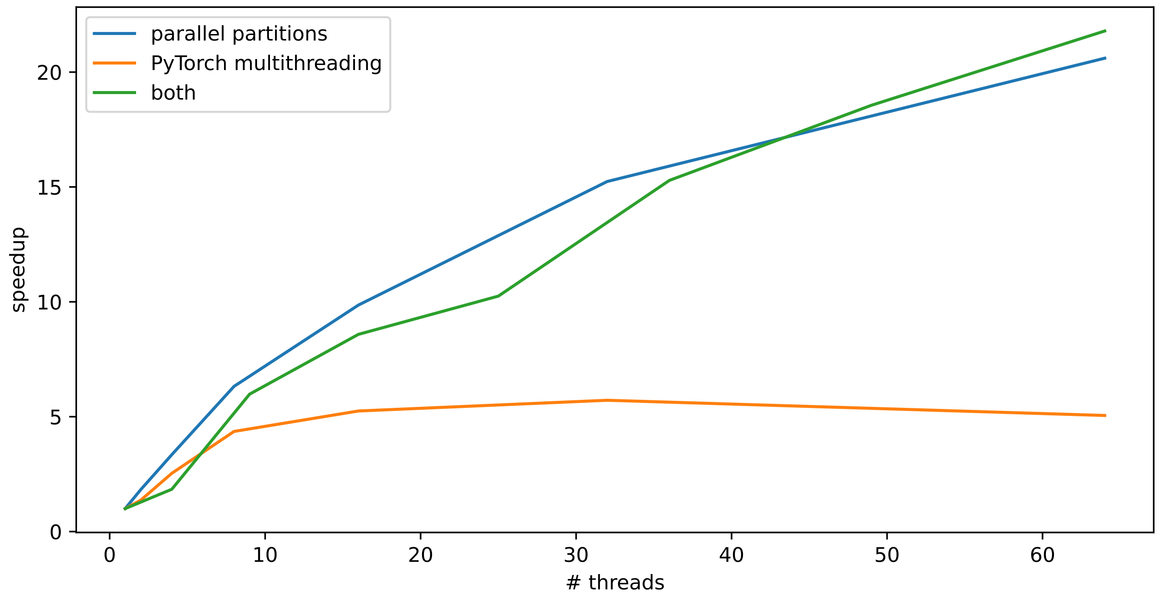


Figure 3.8: Speedup $s = \frac{t_1}{t_n}$, with the training time using a single thread t_1 and the training time using n threads t_n, of training partitions in parallel, multithreading of PyTorch, and using both implementations to the same degree.

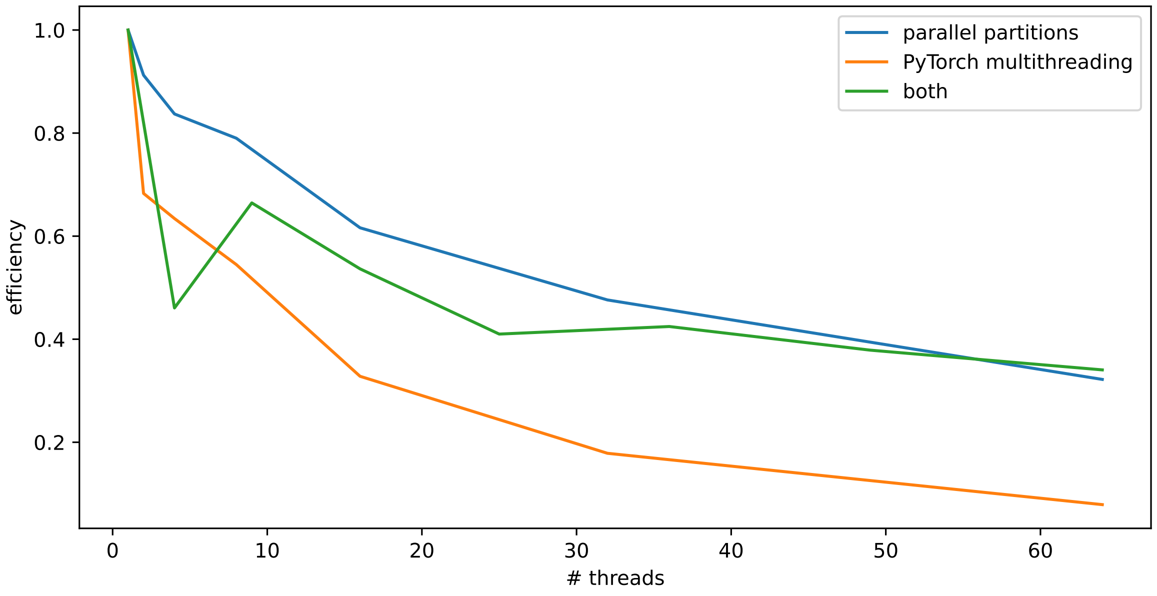


Figure 3.9: Efficiency $e = \frac{s_n}{n}$, with the speedup s_n using n threads, of training partitions in parallel, multithreading of PyTorch, and using both implementations to the same degree.

4 Conclusion

In this thesis, we deployed neural networks to enhance the numerical simulation of a three-dimensional FSI problem. We collected and prepared data produced by the original simulation such that a recurrent neural network could use it as training data. To apply parallelization to the training process, we partitioned the tube such that each partition was able to be trained independently and therefore in parallel. Furthermore, we improved the prediction accuracy of the neural networks by implementing overlap to the partitions whereby information was shared between them.

Our results in terms of neural network architecture revealed that using multiple layers for the initial fully-connected and recurrent parts increases the training time but also achieves a lower error. Using L2 regularization by adding weight decay during training didn't increase the MSE while only adding to the training time marginally. We used these findings for further tests concerning partitioning.

For comparisons of partitioning methods, we found that training time didn't change when deploying any amount of overlap compared to the non-overlapping variant. However, MSE was reduced the more overlap was used. When comparing continuity between partitions, we found a considerable improvement for even little overlap in contrast to using no overlap, especially for later time steps of the simulation.

Finally, we tested the implementation of parallelization for the partitioned tube and compared the results to the multithreading provided by PyTorch directly. Our results showed that for a higher number of available threads, using parallel partitions or a hybrid approach achieved a greater speedup and efficiency compared to utilizing only PyTorch's multithreading.

Our results show promise for the approach of using ML methods on FSI simulation data, however, there is the possibility for better results when using neural networks of an even greater number of hidden layers or dimensions of the hidden layers. Overlapping of partitions turned out to be considerably necessary to achieve favorable results of predictions, yet it's not entirely clear what the precise amount of overlap required is. The use of parallel partitions proved to be able to decrease training time when compared to multithreading of PyTorch. Based on these results, further research can build upon the presented methods to improve the accuracy of predictions and reduce training time by optimizing parallelization.

Bibliography

[1] A. Totounferoush, N. E. Pour, S. Roller, M. Mehl. “Parallel Machine Learning of Partial Differential Equations.” In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2021, pp. 698–703 (cit. on pp. 5, 11, 15, 17).

[2] A. L. Samuel. “Some studies in machine learning using the game of checkers. II—recent progress.” In: *Computer Games I* (1988), pp. 366–400 (cit. on p. 9).

[3] Y. LeCun, Y. Bengio, G. Hinton. “Deep learning.” In: *nature* 521.7553 (2015), pp. 436–444 (cit. on p. 9).

[4] M. I. Jordan, T. M. Mitchell. “Machine learning: Trends, perspectives, and prospects.” In: *Science* 349.6245 (2015), pp. 255–260 (cit. on p. 9).

[5] F. Noé, A. Tkatchenko, K.-R. Müller, C. Clementi. “Machine learning for molecular simulation.” In: *arXiv preprint arXiv:1911.02792* (2019) (cit. on p. 9).

[6] G. Carleo, I. Cirac, K. Cranmer, L. Daudet, M. Schuld, N. Tishby, L. Vogt-Maranto, L. Zdeborová. “Machine learning and the physical sciences.” In: *Reviews of Modern Physics* 91.4 (2019), p. 045002 (cit. on p. 9).

[7] A. Totounferoush, A. Schumacher, M. Schulte. “Partitioned Deep Learning of Fluid-Structure Interaction.” In: *arXiv preprint arXiv:2105.06785* (2021) (cit. on pp. 9, 10).

[8] T. M. Deist, A. Patti, Z. Wang, D. Krane, T. Sorenson, D. Craft. “Simulation-assisted machine learning.” In: *Bioinformatics* 35.20 (2019), pp. 4072–4080 (cit. on p. 9).

[9] B. Lattimer, J. Hodges, A. Lattimer. “Using machine learning in physics-based simulation of fire.” In: *Fire Safety Journal* 114 (2020), p. 102991 (cit. on p. 9).

[10] M. Buchanan. “The power of machine learning.” PhD thesis. Nature Publishing Group, 2019 (cit. on p. 9).

[11] M. Raissi, P. Perdikaris, G. E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations.” In: *Journal of Computational physics* 378 (2019), pp. 686–707 (cit. on p. 9).

[12] *Elastic tube 3D*. URL: https://precice.org/tutorials-elastic-tube-3d.html. (accessed: Jun 6, 2022) (cit. on pp. 10, 13).

[13] *Elastic tube 3D*. URL: https://github.com/precice/tutorials/tree/master/elastic-tube-3d. (accessed: Jun 6, 2022) (cit. on p. 13).

[14] *OpenFOAM debian*. URL: https://develop.openfoam.com/Development/openfoam/-/wikis/precompiled/debian#openfoam-ubuntu-repository. (accessed: Jun 6, 2022) (cit. on p. 13).

[15] *Installing preCICE*. URL: https://precice.org/installation-overview.html. (accessed: Jun 6, 2022) (cit. on p. 13).

[16] *Get the OpenFOAM adapter*. URL: https://precice.org/adapter-openfoam-get.html. (accessed: Jun 6, 2022) (cit. on p. 13).

[17] *Get the CalculiX adapter*. URL: https://precice.org/adapter-calculix-get-adapter.html. (accessed: Jun 6, 2022) (cit. on p. 13).

[18] *VTK file formats*. URL: https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf. (accessed: Jun 6, 2022) (cit. on p. 14).

[19] *ParaView*. URL: https://www.paraview.org/. (accessed: Jun 6, 2022) (cit. on p. 14).

[20] *meshio*. URL: https://pypi.org/project/meshio/. (accessed: Jun 6, 2022) (cit. on p. 15).

[21] *PyTorch*. URL: https://pytorch.org/. (accessed: Jul 7, 2022) (cit. on p. 18).

[22] *PyTorch Linear*. URL: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html. (accessed: Jul 7, 2022) (cit. on p. 19).

[23] *PyTorch ReLU*. URL: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html. (accessed: Jul 7, 2022) (cit. on p. 19).

[24] *PyTorch RNN*. URL: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html. (accessed: Jul 7, 2022) (cit. on p. 19).

[25] J. L. Elman. "Finding structure in time." In: *Cognitive science* 14.2 (1990), pp. 179–211 (cit. on p. 19).

[26] *PyTorch torch.reshape*. URL: https://pytorch.org/docs/stable/generated/torch.reshape.html. (accessed: Jul 7, 2022) (cit. on p. 20).

[27] D. P. Kingma, J. Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 22).

[28] *PyTorch Adam*. URL: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html. (accessed: Jul 7, 2022) (cit. on p. 22).

[29] *CUDA*. URL: https://developer.nvidia.com/cuda-zone. (accessed: Jul 7, 2022) (cit. on p. 22).

[30] *MPI for Python*. URL: https://mpi4py.readthedocs.io/en/stable/. (accessed: Jul 7, 2022) (cit. on pp. 25, 31).

[31] *torch.set_num_threads*. URL: https://pytorch.org/docs/stable/generated/torch.set_num_threads.html. (accessed: Jul 7, 2022) (cit. on p. 31).

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentlich Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Datum und Unterschrift:

Declaration

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

Date and Signature: