|epcc|

Parallel-in-time Methods with Machine Learning - Report

Viktor Csomor

April 17, 2020

Contents

1 Introduction

Differential equations can model a wide array of dynamical systems that play important roles in science, engineering, and finance. Examples of these include population growth, the motion of fluids, and stock market dynamics. While some simple differential equations can be solved analytically, most of them require a numerical approach [1, p. 310]. Due to the importance of these models, their accurate and efficient solution is a well researched area of numerical analysis. A lot of this research is focused on the parallelisation of numerical solvers to achieve higher accuracy or to reduce execution times. Usually, this parallelisation is applied either across the system or across time [2]. The former generally means exploiting parallelism for the evaluation or integration of the right-hand side of a differential equation at a single time point [3, p. 1-2]. While this is a straightforward approach, it only works well for high dimensional differential equations. In contrast, parallel-in-time methods allow for the utilisation of high levels of parallelism regardless of the dimensionality of the problem.

Our project focuses on the Parareal algorithm [4] from the family of parallel-in-time methods. Given the necessary computational resources, this algorithm can solve problems faster, in terms of wall clock time, than traditional parallel solvers which can usually only take advantage of limited levels of parallelism tightly bounded by the problem size. This property of the algorithm makes it ideal for problems requiring real-time solutions which is why it is called "Parareal" [4, p. 3].

The Parareal algorithm solves time-dependent differential equations by splitting them up along the time domain and assigning a sub-problem to each processor. It first computes the initial values for each sub-problem serially using a coarse operator. It then solves each of these problems based on the initial values in parallel using a fine operator and calculates a correction to the initial values. This iterative process goes on until a sufficient level of accuracy is reached. As the coarse operator is executed serially at every iteration, it has to be fast but it also needs to be accurate enough for rapid convergence.

This brings us to the primary hypothesis of our project which is that a machine learning (ML) model can be trained and used as the coarse operator in a Parareal framework and it can achieve better performance, in terms of accuracy and execution time, than conventional operators. As there is a substantial body-of-work on using ML to solve differential equations, we begin this report by briefly introducing the most relevant and most notable pieces of research. We then present some of the results of our preliminary investigations into the dissertation project. Following that, we elaborate on our final project proposal, introduce our work plan, present our risk analysis, and outline the contents of the dissertation report. Lastly, we finish with a review of Nestor Sanchez' dissertation from 2018 "Fitting Large-Scale Gaussian Mixtures With Accelerated Gradient Descent" [5].

2 Background and Literature Review

The Parareal algorithm works by dividing the time domain up into P slices where P is the number of executing processes or threads. Each one of these P time slices then represents an initial value problem (IVP) that can be solved in parallel. To calculate the initial values, the Parareal algorithm relies on a coarse operator, G, that is run serially. Once the initial value estimates are computed, the fine operator, F, is run in parallel across the time slices to solve the individual IVPs and provide correction terms for the subsequent initial values. This is an iterative process where each iteration refines the solution and the total number of iterations necessary is determined by the solution's accuracy requirements.

As G is executed serially, it needs to be as fast as possible to avoid creating a bottleneck that restricts parallel speed-up. However, it also needs to be accurate enough to keep the number of corrective iterations required at the minimum. Conventional coarse operators, such as those relying on numerical integration,

usually trace the approximate solution using a significantly finer temporal discretisation than the extents of the P time slices. The finer this step size is, the more accurate the initial value estimates are [1, p. 319-320]. However, a finer step size also means more evaluations to calculate the initial values. Moreover, there is an additional trade-off between the sophistication of the operator and the step size. Higher order numerical integrators, for example, are more accurate but they are also more computationally expensive [1, p. 325].

The Parareal algorithm has been attracting a lot of interest due to its suitability for long-time simulations and problems requiring fast solutions [6]. A lot of this interest has been directed to the analysis of the algorithm itself [4] [7]. An important property of the algorithm is that it converges to the serial solution as computed by F in a maximum of P iterations [4, p. 6-7]. However, for optimal speedup, it needs to converge in much fewer iterations than P. In fact, if the computational cost of integrating over the entire domain using the operator G matches the cost of integrating over a single time slice using F, the algorithm has to converge in fewer than $P/2$ iterations to achieve any speedup [4, p. 7].

This trade-off between the number of iterations to convergence and the computational cost of G is a crucial element of the Parareal algorithm's performance characteristics. Operators that are faster at the same level of accuracy result in better parallel speedup due to a reduced serial bottleneck. Likewise, operators that are more accurate at the same speed yield better parallel speedup due to faster convergence. This leads us to another important property of the algorithm which is that as long as both G and F are convergent and stable, they can be of completely different classes of solvers and they may even solve slightly different versions of the differential equation [4, p. 8]. Therefore, the Parareal algorithm is indifferent to how its operators work as long as they meet the above requirements and conform to the same interface.

This property of the algorithm has been the focal point of most research into maximizing its speedup. Some of this research has tried to address the speed-accuracy dilemma of the coarse operator by relaxing its speed requirements through more efficient implementations of the Parareal algorithm using improved scheduling. In these event based frameworks, instead of waiting for the entire previous iteration to complete, the tasks associated with the time slices are run as soon as the necessary data is available [8] [9].

In contrast, others have attempted to tackle the issue directly by experimenting with different coarse operators. Generally, a simple Runge-Kutta method such as the forward Euler method is used as G and a higher order method, potentially with finer temporal discretisation, such as the Runge-Kutta 4th order (RK4) method is used as F [6, p. 1822] [10, p. 2]. However, it has been shown that there is more to consider than just the integration method and its step size. In some cases, it is possible to improve the speed of the coarse operator without a significant detriment to accuracy by simplifying the differential equation (only for G) and thus reducing the complexity of the model it operates on [10]. Consequently, selecting and fine tuning the coarse operator is a challenging task that boils down to finding or building a differential equation solver with optimal speed and accuracy characteristics for the problem at hand.

Given the success of ML in a wide range of computational fields, it is unsurprising that their application to the solution of differential equations is a popular field of research. The most attractive prospect of using ML to this end is that a trained model can directly represent the solution without the need for any discretisation (which can be intractable for high dimensional problems). Due to their prowess as universal function approximators [11], most efforts are targeted towards using artificial neural networks (ANN) to approximate the solutions of differential equations [12] [13] [14]. This is also partly attributable to the fact that ANNs are differentiable which allows for more efficient training approaches. While the majority of recent works take advantage of the differentiability of neural networks and use training processes tailored explicitly to them, it has been shown that it is possible to use ML models to approximate the solutions of differential equations in a strictly supervised, black-box setting as well [15].

Perhaps the most successful approach so far to solve complex differential equations with ML is based on

physics informed neural networks (PINN) [16] [17] [18]. PINNs approximate the function that solves the differential equation, therefore their inputs are the values of the differential equation's independent variables (t and x) and their output is the value of the solution at the point defined by the inputs. In terms of architecture, PINNs are flexible and can be implemented by simple ANNs, convolutional neural networks, and even recurrent neural networks (RNN). The most important feature of PINNs is their specialised loss function that penalises violations of the initial conditions, the boundary conditions, and the differential equation itself in the form of mean squared error terms. By minimising this loss function, for example through gradient descent, the network can be trained in a largely unsupervised fashion on collocation points sampled from the differential equation's spatiotemporal domain according to a multivariate distribution (with an additional set of points sampled from the boundaries of the domain according to a different distribution). Although training a PINN can be very costly, especially on differential equations with a large domain, a trained model may perform more than an order of magnitude faster than conventional solvers at comparable levels of accuracy [19]. However, while PINNs are fully mesh-free and very efficient for high dimensional partial differential equations (PDE), for some problems they are still slower than conventional solvers such as the finite element method [20].

One of the main issues with PINNs is that their training can easily become prohibitively expensive for long-time simulations as the amount of data required to train them is proportional to the extent of the temporal domain. This is what parallel-in-time PINNs (PPINN) [21] address by essentially replacing the fine operator F in a Parareal framework with a PINN. We note that while PPINNs involve both the Parareal algorithm and an ML operator, they are merely a time-parallel implementation of PINNs rather than a solution to the speed-accuracy dilemma of coarse operators. Therefore, PPINNs are vulnerable to the same speedup limitations imposed by potentially poor convergence or serial bottlenecks.

3 Preliminary Investigations

In order to complement our literature review and crystallise our understanding of the dissertation project, we have taken a practical approach to our initial investigations. As part of these investigations, we have accomplished three milestones which we briefly present below.

The first one of these milestones has been the implementation of a basic one dimensional diffusion simulation in C defined by the model

$$\frac{\partial u}{\partial t} = d\frac{\partial^2 u}{\partial x^2}$$

where u is the unknown function, x is the spatial coordinate, t is time, and d is the diffusion coefficient. This simulation uses simple Dirichlet boundary conditions and Gaussian initial values. The spatial discretisation is done via the finite difference method and the integrator used is the forward Euler method. The implementation can be found at `code/c/src/main.c` in the project repository.

The second, somewhat more ambitious milestone has been the implementation of a simple shared-memory Parareal framework in C using POSIX threads to solve an ordinary differential equation (ODE) describing the population growth of rabbits. This ODE is expressed by

$$\frac{\mathrm{d}R}{\mathrm{d}t} = \alpha R \tag{1}$$

where R is the number of rabbits and α is the population growth rate. The framework employs the forward Euler method as G and RK4 as F but it can only handle matching coarse and fine operator step sizes. Although the algorithm runs for a predefined number of corrective iterations, the implementation demonstrates a significant speedup at reasonable levels of accuracy even when running on only four threads. Moreover, it showcases how the parallel solution converges to the serial fine solution as the

number of iterations approaches P. This implementation can be found in the same file as the diffusion simulation.

Finally, our third and most substantial milestone has been the development of a distributed Parareal framework in Python using mpi4py [22]. This framework is fairly flexible in that it can handle any operator and any ODE that conform to the respective interface definitions. The algorithm also supports early stopping based on the absolute value of the largest update to the solution at any time step. In addition, the framework provides a number of Runge-Kutta method implementations and a conventional operator that takes an integrator instance and a step size value as its parameters. Furthermore, the framework also includes an ML accelerated operator implementation that is parameterised by a regression model, a target operator, and a step size. The ML operator trains the model on every new differential equation by generating a set of input-output pairs through a stochastic process using the target operator and fitting the regression model to the generated data. The data generation involves repeatedly integrating over the differential equation by the target operator and adding a zero-mean Gaussian noise to the estimate at every time step to use as the initial value for the next step and thus perturbate the trajectory of the target operator's solution. Using this approach, we have successfully trained and utilised an ML operator backed by a linear regression model as the coarse operator G in our Parareal framework to solve a simple ODE. The implementation of this framework and our experiment can be found in the `code/python` directory of the project repository.

Table 1 shows the average execution times, based on ten runs, of two different coarse operators as they integrate over the entire time domain of the rabbit population problem defined by Equation 1 using the values $R_0 = 1000$, $\alpha = 0.0001$, and $t \in [0, 50000]$. The first operator is a conventional one using the explicit midpoint integration method and a step size of 0.25 while the second operator is an ML accelerated one with a step size of 100 backed by an sklearn [23] linear regression model trained on data labelled by the first operator. As seen in the table, the ML accelerated operator solves the differential equation almost fourteen times faster with an average deviation of less than 10^{-6} from the conventional operator's solution. This performance difference is due to the fact that the ML operator's step size is 400 times greater and therefore the solution is evaluated 400 times fewer while still maintaining the same level of accuracy.

	Operator	
	Explicit midpoint	Linear regression
Run time / s	0.233	0.017

Table 1: The average execution times of different coarse operators on the rabbit population problem

Table 2 shows the average execution times, again across ten runs, of different solvers on the same rabbit population problem. The first, serial solver is a conventional operator using the RK4 integration method and a step size of 0.01. The others are all Parareal solvers running on four processes using the first solver as their fine operator and 0.001 as their update threshold (which means that the algorithm finishes once the absolute value of the largest update to the solution at any point of the fine operator's temporal discretisation is below this threshold). The first Parareal solver uses the explicit midpoint operator from Table 1 as its coarse operator while the second and third Parareal solvers use the linear regression operator. Both the conventional and the ML accelerated Parareal solvers converge after just one corrective iteration and thus are able to achieve a significant speedup compared to the serial solver while producing a solution identical to the serial one to six and five significant figures respectively. While the ML accelerated operator is slightly less accurate, it still converges within the same number of iterations given our accuracy threshold and it actually solves the differential equation a whole second faster. In contrast, when taking the model training into account, the ML accelerated Parareal solver is barely faster than the serial solver. However, the training time could potentially be significantly reduced for the given accuracy requirements

by decreasing the volume of training data generated.

	Solver			
	Serial	Parareal	ML Parareal	ML Parareal with training
Run time / s	8.387	6.082	5.080	8.020

Table 2: The average execution times of different solvers on the rabbit population problem

Through these investigations and experiments presented above, we have improved our understanding of the Parareal algorithm and gained a little experience in applied numerical analysis. Moreover, we have also managed to confirm the project's feasibility on a highly simplified problem by showing that an ML accelerated coarse operator can outperform a conventional one. Finally, we have created a strong foundation and starting point for our dissertation project.

4 Final Proposal

The main focus of our project is implementing a Parareal framework using an ML model as the coarse operator and solving differential equations of increasing complexity. The primary metrics we are interested in are the wall time, computational cost, and accuracy of the coarse operator and the framework as a whole. Consequently, we formulate our research questions as

1. How do ML accelerated coarse operators compare to traditional coarse operators in terms of accuracy, execution time, and training time?
2. Is an ML accelerated Parareal framework faster, both excluding and including model training time, than a conventional Parareal framework given the same accuracy requirements?
3. Is an ML accelerated Parareal framework faster, both excluding and including model training time, than other conventional parallel solvers given the same accuracy requirements and using the same number of parallel processes?

To answer our first research question, we propose training and evaluating a number of different regression models to replace the coarse operator. The models we are interested in trying include linear regression, decision tree ensembles such as boosted trees and random forests, and ANNs. All of these models can be treated as black boxes in terms of both training and evaluation. Their output is the estimate of the solution of the differential equation at the next time step (as determined by the step size of the operator) while their inputs are some combination or subset of the values of the independent variables, the derivatives of the solution, and the estimate of the solution at the provided values of the independent variables. These target outputs are supplied by a conventional operator which may be the intended fine operator of the framework or an entirely unrelated one.

In addition, we also intend to try sequence models such as RNNs to capture temporal patterns in the differential equations' solutions. We hypothesize that this increases the robustness of the learned models and aids the approximation of periodic solutions. However, this requires the capturing of inputs from possibly multiple previous time steps. Therefore, we propose a more general training and evaluation scheme where the input is a sequence. The training of the non-recurrent models is, thus, merely a special case of this scheme where the input sequence contains only one element. This allows for the implementation of a single ML accelerated operator that is agnostic of the model.

Finally, we also intend to train and evaluate a separate class of ML accelerated operators backed by PINNs. As the training process of PINNs is significantly different, they are not interchangeable with other models. However, due to the fact that this training process is largely unsupervised and mesh-free, PINNs are likely the best option for higher dimensional PDEs. Therefore, we propose implementing an additional, PINN based ML accelerated operator to be used as the coarse operator of our Parareal framework. This differs from [21] in that employing a PINN as the coarse operator, instead of the fine one, allows us to increase the step size and thus minimise the number of inferences without compromising the fine trajectory of our solution (as the coarse operator only needs to provide the estimates of the initial conditions of the time slices and not the fine solution). Consequently, this enables us to experiment with more complex network architectures in favour of higher accuracy at a lower cost to wall time.

While we are optimistic about machine learning accelerated operators with a greater step size outperforming conventional operators in terms of execution time at similar levels of accuracy, we are conscious of the overhead introduced by training data generation and model fitting. Therefore, we propose measuring the average training time for all the models tested. Moreover, due to the inherently stochastic nature of the training processes, we propose basing all our results on the means and standard deviations of measurements across a high number of runs and using fixed random seeds to facilitate reproducibility. Even though the combined training time and execution time of a machine learning operator is more than likely to significantly exceed the execution time of a conventional operator, this may be mitigated in a Parareal framework where a model is trained only once but used for inference potentially several times. Therefore, by keeping track of the training times of the models, we can establish a more complete picture of their performance potential as coarse operators in a Parareal setting.

We propose implementing our operators in Python using sklearn for linear regression and the decision tree based models and either Tensorflow [24] and Keras [25] or PyTorch [26] for the neural network models. As for our benchmarks, we plan to compare the execution time and accuracy of the ML accelerated operators to conventional solvers provided by scipy [27], FEniCS [28], and FiPy [29] (depending on the type of the differential equation).

To answer our second question, we propose extending the Python Parareal framework implemented as part of our preliminary investigations. As most of the computationally expensive operations are delegated to third party libraries with C bindings, we expect the performance cost of using Python to be minimal. In case the overhead proves to be more significant than expected, we can further optimize our Parareal implementation using static typing with Cython [30].

For the comparison between the ML accelerated and the normal versions of our Parareal framework, we are interested in the wall time of the solution using the same number of processes and the same accuracy requirements. To ensure our analysis is comprehensive enough, we propose performing comparisons across different combinations of process counts and accuracy levels. We intend to time the ML accelerated version both including and excluding the training of the models. Furthermore, as a stretch goal, we also aim to consider the amortized cost of model training through transfer learning. We plan to do so by quantifying how well a model trained on one IVP performs on a different IVP expressed by the same differential equation. We propose using the same approach to answer the third research question as well where we compare our ML accelerated Parareal framework to conventional parallel ODE and PDE solvers provided by FEniCS. For differential equations without an analytic solution, we plan to base our accuracy comparisons on the solutions of higher-order, smaller-step size solvers.

Finally, as for the differential equations to benchmark our operators and framework on, we set forth three different problems of increasing complexity. The first one is the Lotka-Volterra model [31] which is a system of two ODEs expressing the dynamics of the populations of predators and their preys. This requires the extension of our initial framework to handle systems of differential equations which is the same as handling differential equations with vector valued solutions. The second differential equation we propose solving is the two dimensional heat equation. This requires the extension of the framework to

handle PDEs, that is differential equations of multiple independent variables. Lastly, the third problem we consider is the Black-Scholes equation [32] which is a high dimensional PDE that can be computationally challenging to solve using traditional mesh-based approaches.

5 Work Plan

In this section, we outline our work plan for the dissertation period beginning on 25th May and ending on 21st August. The three main tasks of the dissertation we identify, based on our final project proposal, are the implementation of the software framework, the benchmarking of its performance, and the write up of the dissertation report. Each one of these high level tasks are divided up into smaller sub-tasks.

1. **Implementation**:
    1.1. **Testing**: This includes general software testing in the form of regression tests and unit tests of the components of our framework where possible.
    1.2. **Systems of ODEs**: The extension of our Parareal framework to handle systems of ODEs. This is synonymous with enabling differential equations to have vector valued unknown functions. As the first problem we intend to test our framework on, the Lotka-Volterra model, is a system of two ODEs, it is imperative that we complete this sub-task as early as possible.
    1.3. **PDEs**: The extension of our existing framework to handle PDEs. This requires differential equations to be able to specify boundary conditions and have multiple independent variables. The remaining two problems we plan to test our framework on are PDEs, therefore this is a high priority sub-task as well.
    1.4. **ML operators**: The implementation of the ML accelerated operators for the different model types and training processes (one for PINNs and one for all the other models).
    1.5. **Performance optimisation**: This includes hyperparameter tuning and any code optimisations to improve the performance of the operators and the Parareal framework.
2. **Benchmarking**:
    2.1. **Only ML operators**: The benchmarking of the different ML accelerated operators against conventional solvers on the three problems outlined in our project proposal. This includes the measurement of accuracy and training time as well.
    2.2. **Framework with and without ML**: The benchmarking of the framework using different ML accelerated coarse operators against the same framework using different conventional operators. As we intend to use fixed accuracy requirements, the primary metrics for this task are the wall time to solution and the model training time.
    2.3. **Against another parallel solver**: The comparison of the framework using different ML accelerated coarse operators against conventional parallel solvers using the same number of processes. The metrics for this task are the same as above with the addition of accuracy (as we most probably cannot specify accuracy requirements the same way between our Parareal implementation and other solvers).
3. **Writing**:
    3.1. **Engineering log**: The maintenance of an engineering log during the implementation phase to keep track of design decisions and technical details.
    3.2. **Phase 1**: The writing up of the problem formulation and the design and implementation sections of the dissertation.

3.3. **Phase 2**: The completion of the rest of the dissertation such as the introduction, background, results, and conclusions sections.

3.4. **Proof reading**: The quality control of the dissertation report to make sure it is free of typos, grammar mistakes, and other errors.

Figure 1 contains a Gantt chart displaying the timeline of the dissertation project and the different tasks it encompasses. Although the times are just rough and somewhat optimistic estimates, the chart sufficiently visualises the overlaps between certain tasks. For example, it clearly captures the non-linear nature of performance optimisation and benchmarking as they feed back into each other.

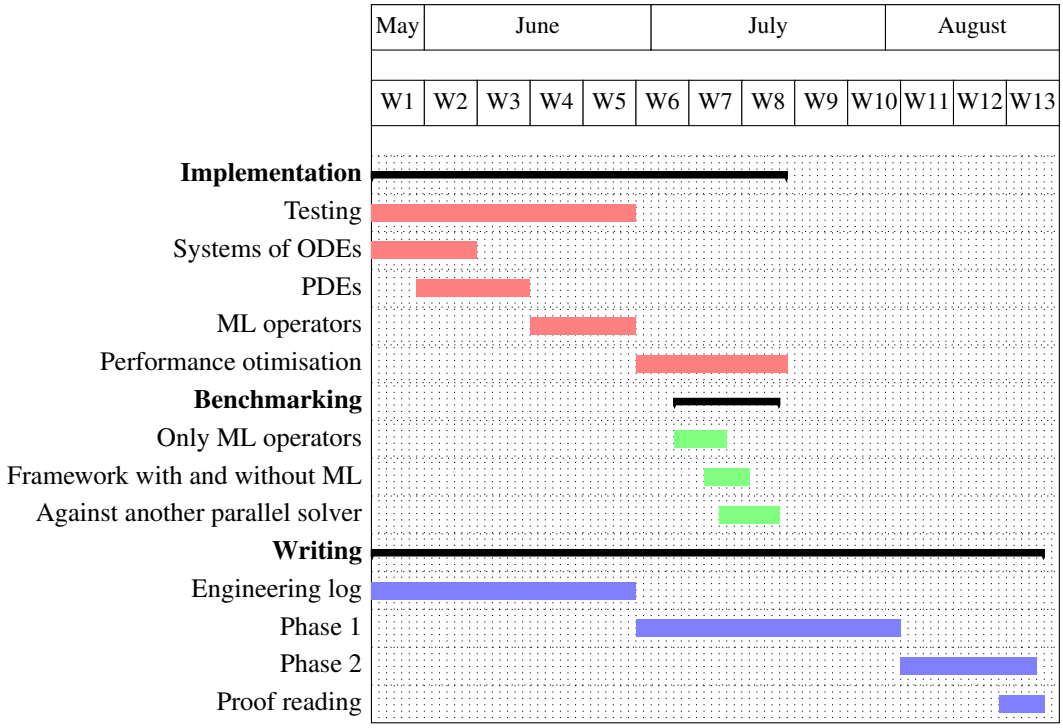

Figure 1: Gantt chart of the dissertation work plan

As for hardware requirements, our plan is to use the author's personal computer for most of the implementation phase. This computer has six physical cores which is sufficient for development and testing purposes. However for optimisation and benchmarking, we plan to use high performance computing services. Due to the generally low number of processors needed for our benchmarks, we estimate our required budget to be about 2000 to 3000 CPU core hours on Cirrus. Additionally, if the new Cirrus GPUs are installed by late June, we also plan to take advantage of them for our neural network based models. For this, our required budget estimate is 500 GPU hours.

6 Risk Analysis

The first risk of the dissertation project we identify is the author's failure to implement a Parareal framework that can handle ODEs, systems of ODEs, and PDEs in time for our benchmarks that need to be performed early enough to leave sufficient time for writing up the results. Failure to do this would likely

restrict our benchmarks to ODEs only which are not representative of more complex real life problems. However, this risk is mitigated by the fact that we already have a strong base in the form of our Parareal framework implementation that can handle any ODE. Moreover, we have a good idea of how to extend this framework to enable systems of ODEs and PDEs as well.

The second major risk is that some of the models or training processes we intend to try may simply not work. While this is a high probability risk, it is rather low impact due to the fact that we plan to try multiple approaches, some of which have already been shown to work on several problems (such as PINNs). Additionally, we have already succeeded in training an ML accelerated operator that can outperform some conventional solvers on a very simple ODE.

The third risk is erratic model behaviour due to the stochastic nature of training. As both our proposed supervised training approach and the unsupervised training method of PINNs have elements of randomness (and the weights of neural networks are initialised randomly as well), the models may behave fairly differently between runs. To address this, we plan to base our results on high numbers of runs. For the sake of reproducibility, we also intend to use fixed random seeds in our benchmarks.

The next main risk we identify is that transfer learning may not work. This means that a model trained on a smaller subset of a differential equation's domain may not be able to infer the solution well enough outside of that subset. Alternatively, a model trained on a differential equation using certain initial value and boundary conditions may not do well on the same equation with different conditions. This is another high probability risk, but utilising transfer learning is only a secondary goal of our project and we believe that our results would be valuable even without it.

The fifth risk is the unavailability of computing resources. This includes multiple possibilities. Cirrus may be less available towards the end of the dissertation period when most students run their experiments. It may not get the new GPUs in time for our benchmarks. Finally, it may become completely unavailable due to unforeseen circumstances. Therefore, it is important that we have a fallback plan in the form of alternative computing resources. In case Cirrus becomes unavailable for any reason, we can fall back to using Archer. If Archer becomes unavailable as well, we can use the author's computer which should be sufficient for smaller scale benchmarks. As for the GPUs, if Cirrus does not get the upgrade in time, we may try to find an alternative GPU computing service or just use CPUs for our neural networks instead.

Finally, the last risk we identify is potential disruptions due to Covid-19. A prolonged pandemic and lockdown may have serious economical implications in the United Kingdom that may indirectly affect the dissertation project. While these effects are hard to predict or prevent, we intend to maintain our focus on the project as our main priority and carry on with our regular remote meetings with the project supervisors.

7 Outline of the Dissertation Report

Based on our final project proposal and work plan, we set forth the following structure of contents for the dissertation report.

1. Introduction
2. Background
    2.1. Parareal algorithm
    2.2. Machine learning for differential equations
    2.3. Parareal with neural network based fine operator

3. Problem statement
    3.1. Operator-agnostic Parareal framework
    3.2. Advantages of machine learning for coarse operator
    3.3. Transfer learning
4. Design and implementation
    4.1. Language and tools
    4.2. Parareal framework
        i. Differential equations
        ii. Integrators
        iii. Operators
    4.3. Machine learning accelerated operators
    4.4. Usage
5. Results
    5.1. Machine learning operator performance
    5.2. Parareal performance with and without machine learning
    5.3. Framework performance against conventional parallel solvers
6. Conclusions

8 Previous Dissertation Review

In this section, we briefly review the 2018 dissertation "Fitting Large-Scale Gaussian Mixtures With Accelerated Gradient Descent" [5] by Nestor Sanchez Guadarrama.

The dissertation presents a new approach to optimise a class of clustering algorithms called Gaussian mixture models (GMM). GMMs are made up of a fixed number of Gaussian distributions that best describe clusters in a data set. To fit the parameters (mean and covariance) of these distributions to the data, usually the expectation maximisation (EM) algorithm is used. However, as EM requires the entire data set to be available all at once, it is unsuitable for high volume or high velocity data. To enable the usage of GMMs for stream processing and massive data sets, Sanchez describes fitting the models using a stochastic gradient descent algorithm (SGD) on a reformulation of the GMM optimisation problem. This reformulation enforces the symmetry and positive-definiteness constraints on the covariance parameters of the Gaussian distributions and thus enables mini-batch optimisation through gradient descent. In addition, he also presents a logarithmic barrier regularisation term to avoid covariance singularities (which are formed by the optimiser minimising the covariance to a null-matrix).

After explaining the theoretical foundations of the dissertation in the second and third sections, Sanchez goes on to present the design and details of his distributed implementation of the SGD optimisable GMM using Scala and Spark. He briefly introduces the main components of his library which include, among others, two regularisation schemes and five variations of the SGD algorithm. He then demonstrates the usage of the library through short examples and code snippets.

In the fifth section, Sanchez begins by introducing his automated test setup and describes how the synthetic test data is generated using the parameters read from a configuration file. He then presents the

results of some qualitative behaviour tests according to which (Nesterov) momentum accelerated SGD with logarithmic barrier regularisation performs the best. This is followed by the results of some hyper-parameter tuning of the optimisers and regularisers which confirm that the effect of logarithmic barrier regularisation is much less detrimental to model quality than that of conjugate prior regularisation. Finally, the dissertation is concluded with the general performance and data streaming test results. The serial performance test results again confirm (Nesterov) momentum accelerated SGD as the best optimiser substantially outperforming EM on large data sets. However, due to an incompatibility between Spark's programming paradigm and Sanchez' distributed SGD implementation, the performance of his program when using mini-batch optimisation in a distributed setting fails to meet expectations.

Sanchez explains the context in which the dissertation is set very well in an iterative fashion. He first gives a high level overview of the dissertation and its topic in the introduction section. This includes a short description of the current state of affairs concerning mainstream GMM implementations, the problem statement, and the motivation of the dissertation project. He then goes on to elaborate on the technical background of the project in a dedicated section which constitutes a wider overview of related works. Finally, he presents the results of the previous work that his dissertation directly builds upon in yet another dedicated section. His approach is a good example for us to follow and is indeed what we base the outline of our dissertation report on.

Sanchez' dissertation is fairly clear about the scope of the problem. It is explicitly stated in the introduction section and also explained in the results section. The approaches taken for the implementation of the program and its testing are also carefully described in the program design and results sections. The testing methods, featuring an automated framework and parameterised synthetic data generation, are remarkably sophisticated and comprehensive. The program is compared against a good standard in the form of the GMM implementation of Spark's ML library. Due to the fixed random seeds, the results are reproducible as well. These are all good practices that we intend to utilise for our dissertation, too. However, Sanchez is not entirely clear about the metrics of success. Although the serial benchmarks show promising results, the distributed tests can probably be considered unsuccessful. Moreover, the data streaming performance test results are hard to interpret without additional context or a reference to compare against.

The overall structure of the dissertation is well organised and intuitive. Sanchez does an excellent job explaining the technicalities in an easily understandable, straightforward manner. He manages this by using simple and concise sentences that build on top of each other and thus allow him to progressively convey complex information. Generally, the language he uses is objective and appropriate for a scientific report. With the exception of a few typos and grammar mistakes, his writing, punctuation, and overall quality of presentation are excellent.

Sanchez' use of algorithms, tables, and figures is clean and fairly consistent. His figures are visually pleasing, well labeled, big enough to read the details, and most importantly, they aid understanding. Some of them could perhaps benefit from a grid and more visible axes, but the general quality of the figures is high. His algorithms and tables are all neatly formatted as well and all his captions are informative. Except for equation 25, which is numbered but never referenced, his use of equations throughout the report is clean and helpful. Finally, while Sanchez' references are well formatted and organised, some of his in-text citations are rather unconventional due to missing white spaces between the words in the text and the citations.

References

[1] E. Süli and D. F. Mayers. *An introduction to numerical analysis*. Cambridge university press, 2003.

[2] C. W. Gear. Parallel methods for ordinary differential equations. *CALCOLO*, 25(1):1–20, Mar 1988.

[3] S. I. Solodushkin and I. F. Iumanova. Parallel numerical methods for ordinary differential equations: a survey, 2016.

[4] G. A. Staff. The parareal algorithm. *Science And Technology*, 60(2):173–184, 2003.

[5] N. Sanchez. Fitting large-scale Gaussian mixtures with accelerated gradient descent, 2018.

[6] G. Gurrala et al. Parareal in time for fast power system dynamic simulations. *IEEE Transactions on Power Systems*, 31(3):1820–1830, 2015.

[7] M. J. Gander and S. Vandewalle. Analysis of the parareal time-parallel time-integration method. *SIAM Journal on Scientific Computing*, 29(2):556–578, 2007.

[8] E. Aubanel. Scheduling of tasks in the parareal algorithm. *Parallel Computing*, 37(3):172–182, 2011.

[9] L. A. Berry et al. Event-based parareal: A data-flow based implementation of parareal. *Journal of Computational Physics*, 231(17):5945–5954, 2012.

[10] N. Duan et al. Applying reduced generator models in the coarse solver of parareal in time parallel power system simulation. In *2016 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, pages 1–5. IEEE, 2016.

[11] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[12] I. E. Lagaris, A. Likas, and D. I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

[13] I. E. Lagaris, A. C. Likas, and D. G. Papageorgiou. Neural-network methods for boundary value problems with irregular boundaries. *IEEE Transactions on Neural Networks*, 11(5):1041–1049, 2000.

[14] J. Han, A. Jentzen, and E. Weinan. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018.

[15] F. Regazzoni, L. Dede, and A. Quarteroni. Machine learning for fast and reliable solution of time-dependent differential equations. *Journal of Computational Physics*, Jul 2019.

[16] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017.

[17] A. M. Tartakovsky et al. Learning parameters and constitutive relationships with physics informed deep neural networks. *arXiv preprint arXiv:1808.03398*, 2018.

[18] J. Sirignano and K. Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, Dec 2018.

[19] G. S. Misyris, A. Venzke, and S. Chatzivasileiadis. Physics-informed neural networks for power systems. *arXiv preprint arXiv:1911.03737*, 2019.

[20] L. Lu et al. DeepXDE: A deep learning library for solving differential equations, 2019.

[21] X. Meng et al. PPINN: Parareal physics-informed neural network for time-dependent PDEs, 2019.

[22] L. Dalcín, R. Paz, and M. Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108 – 1115, 2005.

[23] F. Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] M. Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[25] François Chollet et al. Keras. `https://keras.io`, 2015.

[26] A. Paszke et al. PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dÁlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[27] P. Virtanen et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020.

[28] M. S. Alnæs et al. The FEniCS project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[29] J. E. Guyer, E. Wheeler, and J. A. Warren. FiPy: Partial differential equations with Python. *Computing in Science & Engineering*, 11(3):6–15, 2009.

[30] S. Behnel et al. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, Mar-Apr 2011.

[31] A. J. Lotka. Elements of physical biology. *Science Progress in the Twentieth Century (1919-1933)*, 21(82):341–343, 1926.

[32] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.