

Parareal with a physics-informed neural network as coarse propagator^{*}

Abdul Qadir Ibrahim^{1[0000-0003-3452-8500]}, Sebastian Götschel^{1[0000-0003-0287-2120]}, and Daniel Ruprecht^{1[0000-0003-1904-2473]}

Chair Computational Mathematics, Institute of Mathematics, Hamburg University of Technology, Hamburg, Germany
 {abdul.ibrahim, sebastian.goetschel, daniel.ruprecht}@tuhh.de

Abstract. Parallel-in-time algorithms provide an additional layer of concurrency for the numerical integration of models based on time-dependent differential equations. Methods like Parareal, which parallelize across multiple time steps, rely on a computationally cheap and coarse integrator to propagate information forward in time, while a parallelizable expensive fine propagator provides accuracy. Typically, the coarse method is a numerical integrator using lower resolution, reduced order or a simplified model. Our paper proposes to use a physics-informed neural network (PINN) instead. We demonstrate for the Black-Scholes equation, a partial differential equation from computational finance, that Parareal with a PINN coarse propagator provides better speedup than a numerical coarse propagator. Training and evaluating a neural network are both tasks whose computing patterns are well suited for GPUs. By contrast, mesh-based algorithms with their low computational intensity struggle to perform well. We show that moving the coarse propagator PINN to a GPU while running the numerical fine propagator on the CPU further improves Parareal's single-node performance. This suggests that integrating machine learning techniques into parallel-in-time integration methods and exploiting their differences in computing patterns might offer a way to better utilize heterogeneous architectures.

Keywords: Parareal · parallel-in-time integration · PINN · Machine learning · GPUs · heterogeneous architectures

1 Introduction

Models based on differential equations are ubiquitous in science and engineering. High-resolution requirements, often due to the multiscale nature of many problems, typically require that these models are run on high-performance computers

* This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955701. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Belgium, France, Germany, and Switzerland. This project also received funding from the German Federal Ministry of Education and Research (BMBF) grant 16HPC048. This project has also received funding from the German Federal Ministry of Education and Research (BMBF) under grant 16ME0679K.

to cope with memory demand and computational cost. Spatial parallelization is already a widely used and effective approach to parallelize numerical algorithms for partial differential equations but, on its own, will not deliver enough concurrency for extreme-scale parallel architectures. Parallel-in-time integration algorithms can help to increase the degree of parallelism in numerical models. Combined space-time parallelization can improve speedup over spatial parallelization alone on hundreds of thousands of cores [24].

Parallel-in-time methods like Parareal [14], PFASST [4] or MGRIT [5] rely on serial coarse level integrators to propagate information forward in time. These coarse propagators constitute an unavoidable serial bottleneck which limits achievable speedup. Therefore, the coarse-level integrators must be as fast as possible. However, these methods are iterative and speedup will also decrease as the number of iterations goes up. A coarse propagator that is too inaccurate, even when computationally cheap, will not provide good speedup because the number of required iterations will be too large. Hence, a good coarse propagator needs to be at least somewhat accurate but also needs to run as fast as possible. This trade-off suggests that using neural networks as coarse propagators could be promising: once trained, they are very fast to evaluate while still providing reasonable accuracy. Furthermore, neural networks are well suited for running on GPUs whereas mesh-based discretizations are harder to run efficiently because of their lower computational intensity. Therefore, algorithms featuring a combination of mesh-based components and neural network components would be well suited to run on heterogeneous systems combining CPUs and GPUs or other accelerators.

Our paper makes three novel contributions. It (i) provides the first study of using a PINN as a coarse propagator in Parareal, (ii) shows that a PINN as a coarse propagator can accelerate Parareal convergence and improve speedup and (iii) illustrates that moving the PINN coarse propagator to a GPUs improves speedup further. While we demonstrate our approach for the Black-Scholes equation, a model from computational finance, the idea is transferable to other types of partial differential equations where Parareal was shown to be effective. We only investigate performance on a single node with one GPU. Extending the approach to parallelize in time across multiple nodes and to work in combination with spatial parallelization left for future work. The code used to generate the results shown in this paper is freely available [19].

2 Related Work

Using machine learning (ML) to solve differential equations has become an active field of research. Some papers aim to entirely replace the numerical solver by neural networks [21,25]. Physics-informed neural networks (PINNs) [20], which use the residual of a partial differential equation (PDE) as well as boundary- and initial conditions in the loss function, are used in many applications. This includes a demonstration for the Black Scholes equation (1), showing that a PINN is capable of accurately pricing a range of options with complex payoffs,

and is significantly faster than traditional numerical methods [23]. However, solving differential equations with ML alone generally does not provide the high accuracy that can be achieved by numerical solvers. This has led to a range of ideas where ML is used as an ingredient of classical numerical methods instead and not as a replacement [9].

Specific to parallel-in-time integration methods, there are two research directions aiming to connect them with machine learning. On the one hand, there are attempts to use ML techniques to improve parallel-in-time algorithms. Our paper falls into this category. Using a neural network as coarse propagator for Parareal has been studied in two previous papers. Yalla and Enquist [26] were the first to explore this approach. They use a neural network with one hidden layer of size 1000 and demonstrate for a high dimensional oscillator that it helps Parareal converge faster compared to a numerical coarse propagator. However, no runtimes or speedups are reported. Agbogh et al. [1] use a feed-forward deep neural network as a coarse propagator to integrate an ordinary differential equation modeling responses to a robot arm pushing multiple objects. They also observe that the trained coarse propagator improves Parareal convergence compared to a simplified analytical coarse model. Nguyen and Tsai [17] do not fully replace the numerical coarse propagator but use supervised learning to enhance its accuracy for wave propagation modeling. They observe that this enhances stability and accuracy of Parareal, provided the training data contains sufficiently representative examples. Gorynina et al. [6] study the use of a machine-learned spectral neighbor analysis potential in molecular dynamics simulations with Parareal.

A few papers go the opposite way and adopt ideas from parallel-in-time integration methods to parallelize and accelerate the process of training deep neural networks. Günther et al. [7] use a nonlinear multi-grid method to improve the training process of a deep residual network. They use MGRIT, a multi-level generalization of Parareal, to obtain layer-parallel training on CPUs, reporting a speedup of up to 8.5 on 128 cores. Kirby et al. [11] extend their approach to multiple GPUs, obtaining further performance gains. In a similar way, Meng et al. [16] use Parareal to generate starting values for a series of PINNs to help with the training process. Motivated by the observation that it becomes expensive to train PINNs that integrate over long time intervals, they concatenate multiple short-time PINNs instead. They use a cheap numerical coarse propagator and a Parareal iteration to connect these PINNs with each PINN inheriting the parameters from its predecessor. While they mention the possibility of using a PINN as coarse propagator, they do not pursue this idea further in their paper. Lorin [15] derives a parallel-in-time variant of neural ODEs to improve training of deep Residual Neural Networks. Finally, Lee et al. [13] use a Parareal-like procedure to train deep neural networks across multiple GPUs.

3 Algorithms and Benchmark Problem

The Black-Scholes equation is a widely used model to price options in financial markets [3]. It is based on the assumption that the price of an asset follows a

geometric Brownian motion, so that the log-returns of the asset are normally distributed. Closed form solutions exist for the price of a European call or put option [12], but not for more complex options such as American options or options with multiple underlying assets. To be able to compute numerical errors, we thus focus on the European call option, a financial derivative that gives the buyer the right, but not the obligation, to buy an underlying asset at a predetermined price (the strike price) on or before the expiration date. The price V of the option can be modeled by

$$f(V) = \frac{\partial V}{\partial t}(S, t) + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2}(S, t) + rS \frac{\partial V}{\partial S}(S, t) - rV(S, t) = 0, \quad (1)$$

where S denotes the current value of the underlying asset, t is time, r denotes the no-risk interest rate (for example saving rates in a bank) and σ denotes the volatility of the underlying asset. To fully determine the solution to (1), we impose a final state at expiry time $t = T$ and two boundary conditions with respect to S , motivated by the behaviour of the option at $S = 0$ and as $S \rightarrow \infty$. For the call option, the expiry time condition is

$$V(T, S) = \max(S - K, 0) \text{ for all } S. \quad (2)$$

If the underlying asset becomes worthless, then it will remain worthless, so the option will also be worthless. Thus,

$$V(t, 0) = 0 \text{ for all } t. \quad (3)$$

On the other hand, if S becomes very large, then the option will almost certainly be exercised, and the exercise price is negligible compared to S . Thus, the option will have essentially the same value as the underlying asset itself and

$$V(t, S) \sim 0 \text{ as } S \rightarrow \infty, \text{ for fixed } t. \quad (4)$$

For the European call option, we select an interval of $t = 0$ and $T = 1$ and an artificial bound for the asset of $S = 5000\text{€}$.

3.1 Parareal

Parareal is an iterative algorithm to solve an initial value problem of the form

$$V'(t) = \phi(V(t)), \quad t \in [0, T], \quad V(0) = V_0, \quad (5)$$

where in our case the right hand side function ϕ stems from the discretization of the spatial derivatives in (1). Note that the coefficients in (1) do not depend on time, so we can restrict our exposition to the autonomous case. Decompose the time domain $[0, T]$ into N time-slices $[T^n, T^{n+1}]$, $n = 0, \dots, N - 1$. Denote as \mathcal{F} a numerical time stepping algorithm with constant step size δt and high accuracy and as

$$V_{n+1} = \mathcal{F}(V_n) \quad (6)$$

the result of integrating from some initial value V_n at the start time T^n of a time slice until the end time T^{n+1} . Classical time stepping corresponds to evaluating (6) for $n = 0, \dots, N - 1$ in serial. Parareal replaces this serial procedure with the iteration

$$V_{n+1}^{k+1} = \mathcal{G}(V_n^{k+1}) + \mathcal{F}(V_n^k) - \mathcal{G}(V_n^k) \quad (7)$$

where $k = 1, \dots, K$ counts the iterations. The key in (7) is that the computationally expensive evaluation of \mathcal{F} can be parallelized across all N time slices. Here, we always assume that $P = N$ many processes are used and each process holds a single time slice. A visualization of the Parareal workflow as well as pseudocode can be found in the literature [22]. As $k \rightarrow N$, V_n^k converges to the same solution generated by serial evaluation of (6). However, to achieve speedup, we require convergence in $K \ll N$ iterations. An upper bound for speedup achievable with Parareal using P processors to integrate over $N = P$ time slices is given by

$$s_{\text{bound}}(P) = \frac{1}{\left(1 + \frac{K}{P}\right) \frac{c_c}{c_f} + \frac{K}{P}} \quad (8)$$

where K is the number of iterations, c_c the runtime of \mathcal{G} and c_f the runtime of \mathcal{F} [22]. Since (8) neglects overhead and communication, it is an upper bound on achievable speedups and measured speedups will be lower.

3.2 Numerical solution of the Black-Scholes equation

We approximate the spatial derivatives in (1) by second order centered finite differences on an equidistant mesh

$$0 = S_0 < S_1 < \dots < S_N = L \quad (9)$$

with $S_{i+1} - S_i = \Delta S$ for $i = 0, \dots, N - 1$. For the inner nodes, we obtain the semi-discrete initial value problem

$$V'_j(t) = -\frac{1}{2}\sigma^2 S_j^2 \frac{V_{j+1} - 2V_j + V_{j-1}}{\Delta S^2} - rS_j \frac{V_{j+1} - V_{j-1}}{2\Delta S} + rV_j \quad (10)$$

with $j = 1, \dots$. This is complemented by the boundary condition $V_0 = 0$ for a zero asset value. We also impose the asymptotic boundary condition (4) at finite distance L so that $V_N = 0$. In time, we use a second order Crank-Nicolson method for \mathcal{F} and a first order implicit Euler method as numerical \mathcal{G} . Since we have a final condition instead of an initial condition, we start at time $T = 1$ and solve the problem backwards. We use 200 steps for the fine method and 100 steps for the coarse.

3.3 Physics Informed Neural Network (PINN)

The PINN we use as coarse propagator gets a time slice $[t_{\text{start}}, t_{\text{end}}] \subset [0, T]$, the asset price V at t_{start} and stock values S , and outputs the predicted state

of the asset price \tilde{V} at t_{end} . To train it, we define three sets of collocation points in time and stock price: $(S_i, t_i), i = 1, \dots, N_f$ in the interior of the space-time domain for evaluating the residual $f(V)$ of the Black-Scholes equation (1), $(S_i, t_i), i = 1, \dots, N_b$ collocation points on the boundary to evaluate (2), and $S_i, i = 1, \dots, N_{\text{exp}}$ for the final state conditions (3), (4). The loss function to be minimized is given by

$$\text{MSE}_{\text{total}} = \text{MSE}_f + \text{MSE}_{\text{exp}} + \text{MSE}_b, \quad (11)$$

consisting of a term to minimize the PDE residual $f(V)$

$$\text{MSE}_f = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(\tilde{V}(t_i, S_i))|^2, \quad (12)$$

the boundary loss term

$$\text{MSE}_b = \frac{1}{N_b} \sum_{i=1}^{N_b} |\tilde{V}(t_i, S_i) - V(t_i, S_i)|^2, \quad (13)$$

and the loss at expiration

$$\text{MSE}_{\text{exp}} = \frac{1}{N_{\text{exp}}} \sum_{i=1}^{N_{\text{exp}}} |\tilde{V}(T, S_i) - \max(S_i - K, 0)|^2, \quad (14)$$

For our setup, we randomly generate $N_f = 100,000$ collocation points within the domain $[0, 5000] \times [0, 1]$, $N_b = 10,000$ collocation points at the boundary $[0, 1]$ and $N_{\text{exp}} = 10,000$ collocation points to sample the expiration condition over $[0, 5000]$. The derivatives that are required to compute the PDE loss are calculated by automatic differentiation [2]. We compute the PDE residual (12) over the points inside the domain, the boundary condition loss (13) over the spatial boundary and the expiration loss (14) over the end points. The sum of the three forms the total loss function (11). Figure 1 shows a subset of the generated collocation points to illustrate the approach.

The neural network consists of 10 fully connected layers with 50 neurons in each and was implemented using Pytorch [18]. Figure 2 shows the principle of a PINN but for a smaller network for the sake of readability. Every linear layer, excluding the output layer, is followed by the ReLU activation function. The weights for the neural network are initialized using Kaiming [8]. We focus here on a proof-of-concept and have not undertaken a systematic effort to optimize the network architecture but this would be an interesting avenue for future research.

We used the Adam optimizer [10] with a learning rate of 10^{-2} for the initial round of training for 5000 epochs, followed by a second round of training with a learning rate of 10^{-3} for 800 epochs. The training data (collocation points) was shuffled during every epoch to prevent the model from improving predictions based on data order rather than the underlying patterns in the data. Table 1 shows the behavior of the three loss function terms. The total training time for this model was around 30 minutes.

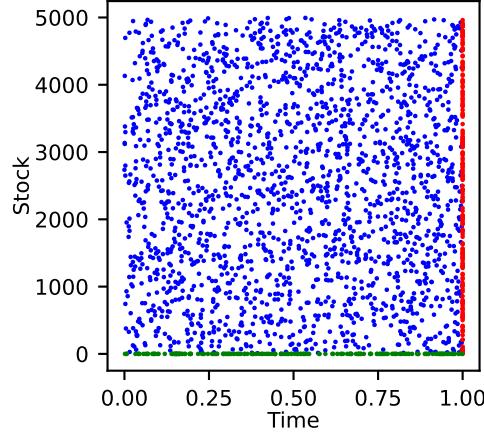


Fig. 1: Subset of the randomly generated collocation nodes. The solution is forced to satisfy the PDE at the inner nodes by minimizing the PDE residual, to satisfy the boundary condition at the green nodes via the boundary loss and the expiration condition at the red nodes via the expiration loss.

4 Results

The numerical experiments were conducted on OpenSUSE Leap 15.4 running an Intel Core 24 x 12th Gen Intel i9-12900K with a base clock speed of 3.2 GHz and a maximum turbo frequency of 5.2 GHz, with 62.6 GiB of RAM and an NVIDIA GeForce RTX 3060/PCIe/SSE2 GPU. Implementations were done using Python 3.10, pytorch1.13.1+cu117, mpi4py3.1.4, as well as numba0.55.1 for the GPU runs.

Parareal convergence. Figure 3 shows the normalized ℓ_2 error for the serial fine, numerical coarse and PINN-coarse propagator over time (left). As expected, the fine propagator is the most accurate with an ℓ_2 error of around 10^{-3} at the end of the simulation. The numerical coarse propagator is noticeably less accurate. The PINN coarse propagator is more accurate than the numerical coarse propagator but also does not reach the accuracy of the fine. To illustrate the importance of encoding the differential equation in the loss function, we also show a neural network (NN) trained only on data produced with the fine propagator but without the terms encoding the differential equation. The neural network without PDE residual is somewhat more accurate than the numerical coarse method but not as good as the PINN. Note that the PINN used here does not need numerically generated trajectories as training data, as the loss function (11) only consists of PDE residual, boundary and expiration conditions and does not include a data mismatch term.

Figure 3 (right) shows the normalized ℓ_2 error of Parareal against the number of iterations. For all three coarse propagators, numerical, NN and PINN, Parareal

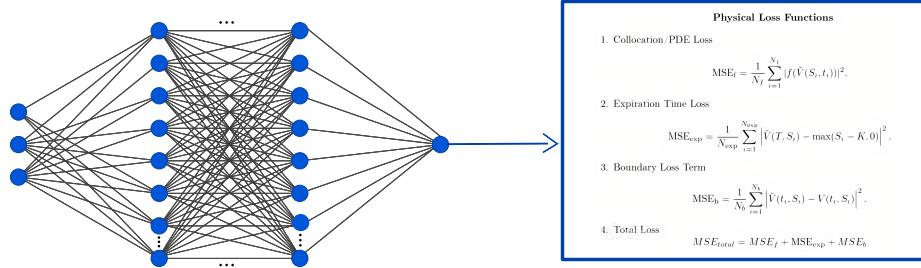


Fig. 2: Structure of the PINN. The network takes the time $t_{\text{start}}, t_{\text{end}}$, asset values V and stock values S as input and returns the predicted asset values \tilde{V} at t_{end} . The loss function encodes the PDE, the expiration condition and the boundary conditions. Figure produced using <https://alexlenail.me/NN-SVG/index.html>.

converges very quickly. Although PINN and NN are slightly more accurate than the numerical coarse propagator, the impact on convergence is small. After one iteration, the iteration error of Parareal is smaller than the discretization error of the fine method. After $K = 3$ iterations, Parareal has reproduced the fine solution up to round-off error. Below, we report runtimes and speedup for $K = 3$. With only a single iteration, the K/P term in (8) is less important and reducing the runtime of the coarse propagator increases overall speedup even more. Therefore, the case with $K = 3$ is the case where switching to the coarse propagator will yield less improvement.

Generalization. Figure 4 shows how Parareal with a PINN coarse propagator converges if applied to (1) with parameters different from those for which the PINN was trained. As parameters become increasingly different from the training values, the coarse propagator will become less accurate. However, if Parareal converges, it will produce the correct solution since the numerical fine propagator always uses the correct parameters. The combination of Parareal + PINN generalizes fairly well. Even for parameters more than ten times larger than the training values it only requires one additional iteration to converge. While the additional iteration will somewhat reduce achievable speedup as given by (8), the performance results presented below should not be overly sensitive to changes in the model parameters.

Parareal runtimes and speedup. Reported runtimes are measured using the `time` command in Linux and include the time required for setup, computation and data movement. Table 2 shows the runtime in milliseconds of Parareal using $P = 16$ cores for four different coarse propagator configurations. Shown are averages over five runs as well as the standard deviation. Replacing the numerical coarse propagator with a PINN on a CPU reduces Parareal execution time by a factor of 2.4, increasing to 2.9 if the PINN is run on a GPU. For the numerical coarse propagator, using the GPU offers no performance gain because the resolution

Epoch	Expiration	Boundary	Residual
0	9.21×10^2	9.21×10^2	7.33×10^3
2000	5.58×10^{-1}	3.45×10^{-2}	2.50×10^{-2}
4000	4.11×10^{-2}	2.34×10^{-2}	5.00×10^{-3}
5000	5.92×10^{-1}	1.34×10^{-2}	4.22×10^{-3}
5300	4.19×10^{-2}	3.22×10^{-3}	1.94×10^{-4}
5500	6.46×10^{-4}	1.96×10^{-4}	5.73×10^{-5}
5800	2.92×10^{-5}	1.14×10^{-5}	3.19×10^{-4}

Table 1: Evolution of the loss function during network training. The three columns show the MSE for the three terms of the loss function related to the end condition (2), boundary conditions (3) and (4) and residual (1). After 5000 epochs with training rate 10^{-2} , another 800 epochs of training with a reduced training rate of 10^{-3} were performed.

and thus computational intensity is not high enough. The much faster coarse propagator provided by the PINN significantly reduces the serial bottleneck in Parareal and will, as demonstrated below, yield a marked improvement in speedup.

	Numerical	PINN	Speedup over CPU-Numerical
CPU	3.48 ± 0.056	1.47 ± 0.073	2.4
GPU	3.99 ± 0.651	1.21 ± 0.041	2.9
Speedup	–	1.21	

Table 2: Runtime c_c in milliseconds of the coarse propagator \mathcal{C} averaged over five runs plus/minus standard deviation.

Table 3 shows runtimes for the full Parareal iteration averaged over five runs. The fastest configuration is the one that runs the numerical fine propagator on the CPU and the PINN coarse propagator on the GPU. Executing both fine and coarse propagator on the CPU takes about a factor of three longer. Importantly, moving both to the GPU, while somewhat faster than running all on the CPU, is slower than the mixed version by a factor of about two. The full GPU variant will eventually be faster if the resolution of the fine and coarse

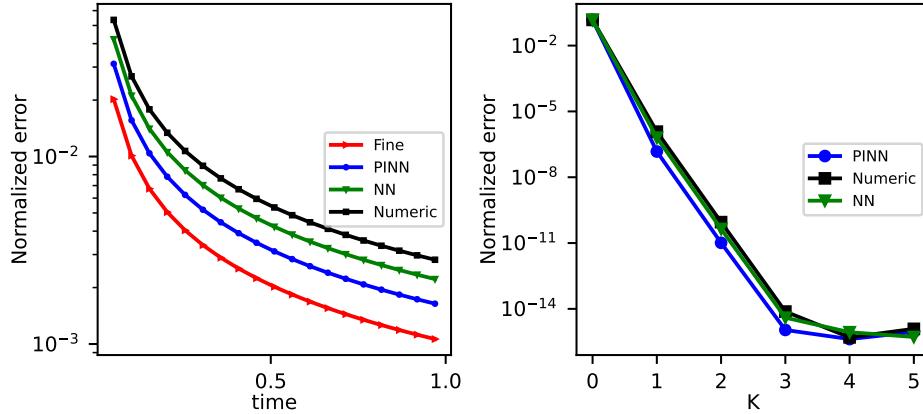


Fig. 3: Normalized ℓ_2 -error over time of coarse and fine propagator against the analytical solution (left). Normalized ℓ_2 -error against the serial fine solution versus number of iterations for three different variants of Parareal (right). The black line (squares) is Parareal with a numerical coarse propagator, the green line (diamonds) is Parareal with a neural network as coarse propagator that is trained only on data while the blue line (circles) is Parareal with a PINN as coarse propagator that also uses the terms of the differential equation in the loss function. Parareal uses $P = 16$ time slices in all cases.

propagator are both extremely high. However, the current resolution already produces an error of around 10^{-3} which will be sufficient in most situations. This illustrates how a combination of numerical method and PINN within Parareal can not only improve performance due to the lower cost of the PINN but also help to better utilize a node that features both CPUs and GPUs or even neural network accelerators. Thus, the different computing patters in finite difference numerical methods and neural networks can be turned into an advantage.

	CPU-Coarse	GPU-Coarse
CPU-Fine	128.48 ± 0.715	41.241970 ± 0.334
GPU-Fine	83.2545 ± 0.356	87.45234 ± 0.253

Table 3: Runtimes in milliseconds for Parareal averaged over five runs plus/minus standard deviation.

Figure 5 shows runtimes for Parareal with both a PINN and numerical coarse propagator on a CPU (left) and GPU (right) against the number of cores/time slices P . The numerical fine propagator is always run on the CPU. In both

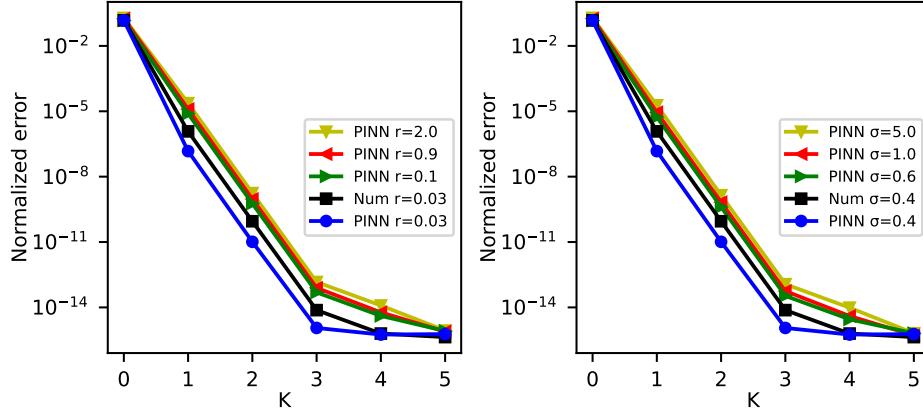


Fig. 4: Convergence of Parareal for different interest rates r (left) and volatilities σ (right). In all cases, the coarse propagator is the PINN trained for values of $r = 0.03$ and $\sigma = 0.4$. Even for parameter values more than ten times larger than the ones for which the PINN was trained, Parareal requires only one additional iteration to converge to within machine precision of the fine integrator.

cases, runtimes decrease at a similar rate as the number of time slices/cores P increases. The numerical coarse propagator is consistently slower than the PINN and the gap is similar on the CPU and GPU. Finally, Figure 6 shows the speedup (left) and parallel efficiency (right) for Parareal with a numerical, PINN-CPU and PINN-GPU coarse propagator. The speedup bounds (8) are shown as lines. Moving from a numerical coarse propagator to a PINN and moving the PINN from the CPU to a GPU each improves speedup significantly. For the numerical coarse propagator, Parareal achieves a speedup of around $S(16) \approx 2$. Replacing the numerical integrator with a PINN improves speedup to $S(16) \approx 3$. Running this PINN on a GPU again improves speedup to $S(16) \approx 4.5$, more than double what we achieved with the numerical coarse propagator on a CPU. The improvements in speedup translate into increased parallel efficiency, which improves from around 30% for the numerical coarse propagator to around 60% for the PINN-GPU coarse method. For smaller numbers of processors, the gains in speedup are less pronounced, because the K/P term in (8) is more dominant. But gains in parallel efficiency are fairly consistent from $P = 2$ cores to $P = 16$ cores. In summary, this demonstrates that replacing a CPU-run numerical coarse propagator with a GPU-run PINN can greatly improve the performance of Parareal by minimizing the serial bottleneck from the coarse propagator.

5 Discussion

Parareal is a parallel-in-time method that iterates between a cheap coarse and a parallel expensive fine integrator. To maintain causality, the coarse propagator

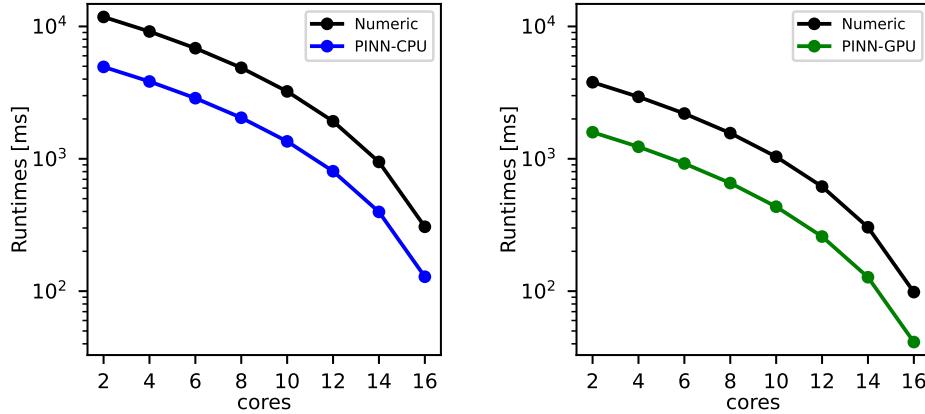


Fig. 5: Runtimes in milliseconds for Parareal (dots) and the serial numerical fine propagator (horizontal lines) on a CPU (left) and GPU (right)

needs to run in serial and therefore reflects a bottleneck that limits achievable speedup. Mostly, coarse propagators are similar to fine propagators and build using numerical methods but with lower order, lower resolution or, in some cases, models of reduced complexity. We investigate the use of a physics-informed neural network (PINN) instead. The PINN is shown to be slightly more accurate than a numerical coarse propagator but a factor of three faster. Using it does not affect convergence speed of Parareal but greatly reduces the serial bottleneck from the coarse propagator.

We show that, on a single node with one GPU, a combination of a numerical fine propagator run on a CPU with a PINN coarse propagator run on a GPU provides more than twice the speedup than vanilla Parareal using a numerical coarse propagator run on the CPU. Also, we demonstrate that moving both fine and coarse propagator to the GPU is slower than moving just the PINN coarse method to the GPU and keeping the numerical fine method on the CPU. The reason is that unless the resolution of the fine propagator is extremely high, its low computational intensity means there is little gain from computing on a GPU and so overheads from data movement are dominant. By contrast, evaluating PINNs is well suited for GPU computation. Our results demonstrate that using PINNs to build coarse level models for parallel-in-time methods is a promising approach to reduce the serial bottleneck imposed by causality. They also suggest that parallel-in-time methods featuring a combination of numerical algorithms and neural networks might be useful to better utilize heterogeneous systems.

References

1. Agboh, W., Grainger, O., Ruprecht, D., Dogar, M.: Parareal with a learned coarse model for robotic manipulation. Computing and Visualization in Science **23**(8) (2020)

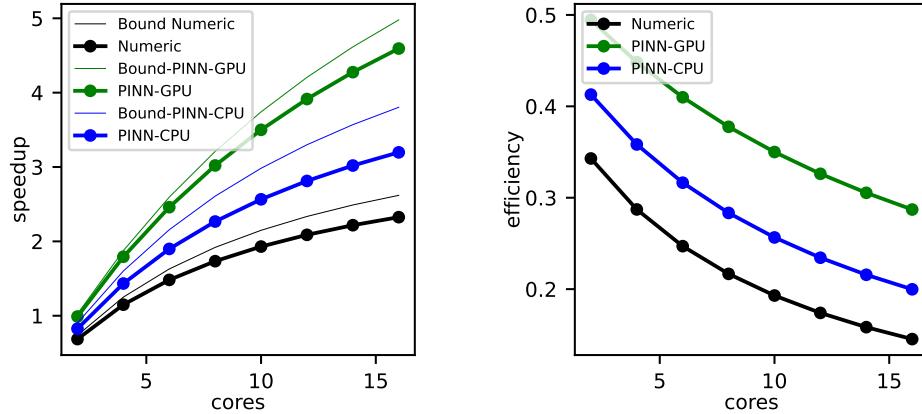


Fig. 6: Speedup (left) and parallel efficiency (right) of Parareal over the serial numerical fine propagator on a CPU. Because the PINN-GPU coarse propagator is faster, it reduces the serial bottleneck of Parareal and allows for better speedup and parallel efficiency.

2. Baydin, A.G., Pearlmutter, B.A., Radul, A.A., Siskind, J.M.: Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research* **18**, 1–43 (2018)
3. Black, F., Scholes, M.: The pricing of options and corporate liabilities. *The Journal of Political Economy* **81**(3), 637–654 (1973)
4. Emmett, M., Minion, M.L.: Toward an Efficient Parallel in Time Method for Partial Differential Equations. *Communications in Applied Mathematics and Computational Science* **7**, 105–132 (2012). <https://doi.org/10.2140/camcos.2012.7.105>
5. Falgout, R.D., Friedhoff, S., Kolev, T.V., MacLachlan, S.P., Schroder, J.B.: Parallel time integration with multigrid. *SIAM Journal on Scientific Computing* **36**, C635–C661 (2014). <https://doi.org/10.1137/130944230>
6. Gorynina, O., Legoll, F., Lelievre, T., Perez, D.: Combining machine-learned and empirical force fields with the parareal algorithm: application to the diffusion of atomistic defects (2022)
7. Günther, S., Ruthotto, L., Schroder, J.B., Cyr, E.C., Gauger, N.R.: Layer-parallel training of deep residual neural networks. *SIAM Journal on Mathematics of Data Science* **2**(1), 1–23 (2020). <https://doi.org/10.1137/19M1247620>
8. He, K., Zhang, X., Ren, S., Sun, J.: Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: Proceedings of the IEEE international conference on computer vision. pp. 1026–1034 (2015)
9. Huang, R., Li, R., Xi, Y.: Learning optimal multigrid smoothers via neural networks. *SIAM Journal on Scientific Computing* pp. S199–S225 (2022). <https://doi.org/10.1137/21M1430030>
10. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization (2014). <https://doi.org/10.48550/arxiv.1412.6980>
11. Kirby, A., Samsi, S., Jones, M., Reuther, A., Kepner, J., Gadepally, V.: Layer-parallel training with GPU concurrency of deep residual neural networks via nonlinear multigrid. In: 2020 IEEE High Performance Extreme Computing Conference (HPEC). IEEE (2020). <https://doi.org/10.1109/hpec43674.2020.9286180>

12. Kumar, S., Yildirim, A., Khan, Y., Jafari, H., Sayevand, K., Wei, L.: Analytical solution of fractional Black-Scholes European option pricing equation by using Laplace transform. *Journal of fractional calculus and Applications* **2**(8), 1–9 (2012)
13. Lee, Y., Park, J., Lee, C.O.: Parareal neural networks emulating a parallel-in-time algorithm. *IEEE Transactions on Neural Networks and Learning Systems* pp. 1–12 (2022). <https://doi.org/10.1109/tnnls.2022.3206797>
14. Lions, J.L., Maday, Y., Turinici, G.: A "parareal" in time discretization of PDE's. *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics* **332**, 661–668 (2001). [https://doi.org/10.1016/S0764-4442\(00\)01793-6](https://doi.org/10.1016/S0764-4442(00)01793-6)
15. Lorin, E.: Derivation and analysis of parallel-in-time neural ordinary differential equations. *Annals of Mathematics and Artificial Intelligence* (2020). <https://doi.org/10.1007/s10472-020-09702-6>
16. Meng, X., Li, Z., Zhang, D., Karniadakis, G.E.: PPINN: Parareal physics-informed neural network for time-dependent PDEs. *Computer Methods in Applied Mechanics and Engineering* **370**, 113250 (2020). <https://doi.org/10.1016/j.cma.2020.113250>
17. Nguyen, H., Tsai, R.: Numerical wave propagation aided by deep learning. *Journal of Computational Physics* **475**, 111828 (2023). <https://doi.org/10.1016/j.jcp.2022.111828>
18. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library (2019). <https://doi.org/10.48550/arXiv.1912.01703>
19. Qadir, I.A., Götschel, S., Ruprecht, D.: Parareal with a physics-informed neural network as coarse propagator (2023). <https://doi.org/10.5281/ZENODO.7695133>
20. Raissi, M., Perdikaris, P., Karniadakis, G.E.: Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations (2017). <https://doi.org/10.48550/arXiv.1711.10561>
21. Ranade, R., Hill, C., Pathak, J.: Discretizationnet: A machine-learning based solver for Navier-Stokes equations using finite volume discretization. *Computer Methods in Applied Mechanics and Engineering* **378**, 113722 (2021). <https://doi.org/10.1016/j.cma.2021.113722>
22. Ruprecht, D.: Shared Memory Pipelined Parareal, pp. 669–681. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-64203-1_48
23. Sirignano, J., Spiliopoulos, K.: DGM: A deep learning algorithm for solving partial differential equations. *Journal of computational physics* **375**, 1339–1364 (2018)
24. Speck, R., Ruprecht, D., Krause, R., Emmett, M., Minion, M.L., Winkel, M., Gibbon, P.: A massively space-time parallel N-body solver. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. pp. 92:1–92:11. SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA (2012). <https://doi.org/10.1109/SC.2012.6>
25. Stender, M., Ohlsen, J., Geisler, H., Chabchoub, A., Hoffmann, N., Schlaefer, A.: Up-Net: A generic deep learning-based time stepper for parameterized spatio-temporal dynamics. Available at SSRN 4053304 (2022)
26. Yalla, G.R., Engquist, B.: Parallel in time algorithms for multiscale dynamical systems using interpolation and neural networks. In: *Proceedings of the High Performance Computing Symposium*. pp. 9:1–9:12. HPC '18, Society for Computer Simulation International (2018)