Parareal: An application of probabilistic methods to time-parallelisation

Kamran Pentland, Yijie Zhou, Haoran Ni, Jimmy McKendrick and Jacques Bara

EPSRC & MRC Centre for Doctoral Training in Mathematics for Real-World Systems, University of Warwick

ARTICLE INFO

Keywords:
Parareal algorithm
Time-parallelisation
Probabilistic methods
Runge-Kutta methods
High performance computing

Abstract

Modern spatial parallelisation techniques are highly efficient and have been applied to a variety of large-scale applied numerical problems. Further efficiency gains and numerical speed-up are, however, limited as algorithms remain constrained by the serial component of time integration. The purpose of this report is to apply probabilistic methods to the parareal algorithm, a robust parallel-in-time numerical integrator, in order to improve upon current convergence rates. A modified, stochastic version of the parareal algorithm is designed, implemented and tested on a number of compute processors in order to rapidly calculate solutions to systems of nonlinear ordinary differential equations. It is demonstrated that, when applied to the Lorenz and Brusselator systems, the modified algorithm converges, with high probability, more rapidly than the original under certain conditions. A large number of sampled solutions (from pre-specified sampling rules built into the algorithm); an optimally sized variance; and a much larger number of compute cores are required for rapid convergence. Two different sampling rules, based on normal distributions, are tested and analysed, with each performing better on a particular system, depending on the size of its variance and the stability of the system. The modified 'stochastic parareal' algorithm is also shown to preserve solution accuracy in both optimal cases.

1. Introduction

In its most basic form, parallel computing is the process by which a numerical problem is partitioned into a number of sub-problems that can be solved simultaneously with or without prior knowledge of each other. More efficient and widespread parallelism is becoming increasingly important for high performance computing within many different fields of study. In applications from molecular chemistry [1], nonlinear ordinary differential equations [2, 3] and turbulent plasma simulations [4, 5], reducing the computational burden on machine hardware is becoming ever more necessary and apparent.

Simply increasing the number of computer cores, or central processing units (CPUs), is no longer a viable option for reducing simulation wallclock times, as saturation points - in terms of financial cost, physical space, power usage and hardware cooling - are being reached [6]. The so called 'power wall' has levelled off increases in CPU and GPU (graphical processing unit) clock speeds, demanding the development of more algorithmic efficiency gains instead. Converting existing sequential algorithms into their equivalent parallel counterparts is challenging, however it is one option for relieving the strain on computer hardware.

Burrage [7] classified parallelisation into three types: parallelism across the system, across the method and across the time. Spatial decomposition methods fit within parallelism across the system and have been well explored for solving initial boundary value problems (IBVPs) sequentially in time and in parallel in space. Often this involves solving smaller IBVPs on small sub-domains (in parallel) and then piecing these together before moving on to the next time step sequentially. These methods are very efficient for high dimensional systems however they too are reaching scale-up limits and integration speeds often bottleneck in the time dimension. For instance, modern algorithms used to simulate Edge Localised Modes (ELMs) in turbulent fusion plasmas can take anywhere between 100-200 days to integrate over a time interval of one second [4].

As mentioned, assigning more computing power will not continually reduce the simulation times for these problems owing to Amdahl's law [8]. Serial time integration will always limit computation speeds and in many cases some solutions, such as the ELMs mentioned previously, become infeasible to attain in good time. Whilst most applications of parallelism involve solving partial differential equations (PDEs), ordinary differential equations (ODEs), which have

no spatial component, cannot utilise existing parallel domain decomposition methods[1]. These ODE problems, as well as time-dependent PDEs, are what motivated the initial research and development of a less obvious but nonetheless fascinating approach: parallel across the time methods.

Numerically speaking, time is perceived as a sequential process over which a system evolves, one (discrete) step after the other, with the solution at each step depending on the previous step(s). Some intuitive thinking is required in order to parallelise an ODE problem in time and hence, integrate it in a non-sequential manner. One approach, similar to spatial parallelisation, involves partitioning the entire time interval into a sequence of smaller sub-intervals on which the problem can be solved using existing numerical techniques in parallel. The obvious difficulty however, arises when trying to prescribe the correct initial condition from which to begin the integration, as initial values for each sub-interval depend upon the solution at the end of the last sub-interval. This approach where one 'shoots' for the solution from an initial condition is usually referred to as a multiple shooting method [9].

Various methods for time parallelisation have been proposed over the last 55 years or so, beginning in 1964, when Nievergelt [10] first proposed partitioning time into sub-intervals and integrating an ODE in parallel, using interpolation to correct for the discontinuities created between the sub-intervals. Though relatively slow, this initial work was paramount to the development and improvement of multiple shooting methods in time [11, 12, 13]. Multigrid approaches were developed in 1984 by Hackbusch [14], solving equations in space-time using different levels of fine and coarse discretisation which could then be solved in parallel [15, 16]. Waveform relaxation methods [17, 18] focus on partitioning space and solving the equation with an initial guess to the solution over the entire time interval, refining the solution iteratively thereafter. Other alternative, but limited approaches, to iterative schemes are direct methods [19]. More detailed reviews of these methods are discussed in both [7] and [20].

Almost 40 years after Nievergelt's initial paper, Lions et al. developed the parareal algorithm [21], an elegantly simple method that combines multigrid and multiple shooting techniques. Many of the methods developed before parareal were often problem-specific, struggling with certain nonlinear equations or being too small/large-scale in terms of parallelisation. Parareal does not suffer from these issues, it is relatively easy to implement and is compatible with many existing numerical integrators [22, 23]. It has, for example, been successfully applied to a range of problems including, but not limited to: molecular dynamics [1]; financial American put options [2]; plasma fusion simulations [4, 5]; and the Navier-Stokes equations [24, 25].

The iterative predictor-corrector update rule employed within parareal has been observed (see above references) to achieve a speed-up of around 5-10 times compared to traditional time sequential integration methods. The aim of this report is to investigate whether applying probabilistic methods to this existing update rule can achieve a faster convergence rate for initial value problems. Convergence is measured by the number of iterations k that the parareal algorithm takes to converge to a continuous solution, not the simulation wallclock time. This is because wallclock timings vary with the quality of compute cores available, how the algorithm has been coded and in what language, whereas k remains fixed regardless. With this is mind, attaining faster simulation times, hence convergence, is crucial in fusion plasma research. In order to simulate highly nonlinear transport, which occurs in dissipative-trapped electron mode turbulence, significant computational resources are required. At time intervals of scientific interest, solutions enter fully-developed turbulent states which will inevitably vary between independent simulations in a stochastic-like manner, hence in order to study the statistical behaviour of a large number of ensemble solutions, alternative methods are required for their rapid numerical computation.

In order to improve upon the existing method, a modified algorithm is built using the same structure of parareal, however instead of selecting a single deterministically updated solution at each time step, a pre-specified 'sampling rule' is used to generate a *distribution* of solutions that are propagated forward in time (in parallel). An optimal solution can then be selected from this ensemble, increasing the probability that the numerical solution 'jumps' closer to the true solution of the equations being solved. The intuition is to exploit the statistical differences that exist between multiple solutions at each time step, thereby increasing the convergence rate. It is the aim that the proposed algorithm will rapidly compute previously unreachable (temporally) solutions to large-scale systems of equations so that statistical analysis, among other research, can be carried out upon them.

The report is organised as follows. Section 2 introduces the original parareal algorithm, demonstrating its convergence and accuracy for two initial value problems: the Brusselator and Lorenz equations. Next, Section 3 details the modified stochastic parareal algorithm and its accompanying sampling rules. In Section 4, the stochastic algorithm

[1] Note that existing 'parallel across the method' algorithms solve ODE problems in parallel. See parallel Runge-Kutta schemes and extrapolation techniques in the literature [7].

is tested and statistically compared to the results found in Section 2. Finally in Section 5, conclusions are drawn and avenues for future research are proposed.

2. The Parareal Algorithm

Much like domain decomposition methods, the main idea behind the parareal algorithm is to partition the time domain into a set of sub-intervals upon which a group of sub-problems can be independently solved in parallel. Initial conditions are required to begin integrating the smaller evolution problems, therefore parareal locates these sequentially, using a fast but low accuracy numerical integrator. Using these low accuracy initial states, a slow but more accurate solver is then deployed in parallel to provide a more accurate approximation to the solution in each sub-interval. The combination of low and high accuracy solutions are then used within a predictor-corrector rule to iteratively update and improve the accuracy of the solution across the time domain.

The following section provides a more detailed mathematical insight into the parareal algorithm in its original form, alongside details regarding its functionality and convergence. Following this, it is applied to two systems of nonlinear ODEs in order to demonstrate convergence and accuracy.

2.1. The Algorithm

Following previously outlined descriptions [21, 23], consider a system of $m \geq 1$ nonlinear ODEs

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}(\mathbf{u}(t)) \quad \text{on} \quad t \in [T_0, T_N], \quad \text{with} \quad \mathbf{u}(T_0) = \mathbf{u}^0, \tag{2.1}$$

where $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^m$ is a multi-valued function, $\mathbf{u} : \mathbb{R} \to \mathbb{R}^m$ the time-dependent vector solution and $\mathbf{u}^0$ the initial condition. Partition the time domain into N intervals such that $[T_0, T_N] = [T_0, T_1] \cup \cdots \cup [T_{N-1}, T_N]$, where each sub-interval has fixed length $\Delta T := T_n - T_{n-1}$ for all $n = 1, \ldots, N$. Figure 1 displays a schematic of the time partitioning with a spatial component that becomes relevant when considering PDE problems[2].

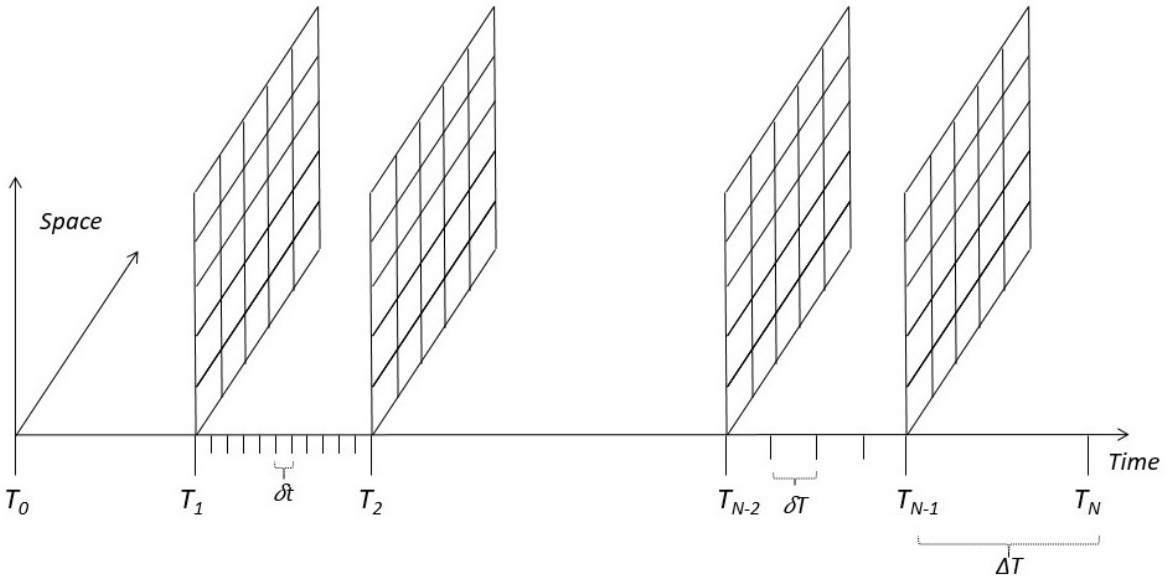


Figure 1: Schematic of the time interval decomposition with a spatially discretised grid for PDE problems if required. Three levels of temporal discretisation are shown: sub-intervals (size ΔT), coarse intervals (size δT) and fine intervals (size δt).

[2]Whilst parareal has been applied to various PDE problems with promising results, the spatial discretisation and integration schemes only add to the complexity of the algorithm. Hence for the purposes of this report, only time-dependent problems need be considered.

Following this decomposition, N smaller sub-problems (2.2) now need to be solved.

$$\frac{d\mathbf{u}_n}{dt} = \mathbf{f}(\mathbf{u}_n(t)) \quad \text{on} \quad t \in [T_n, T_{n+1}], \quad \text{with} \quad \mathbf{u}_n(T_n) = \mathbf{U}_n \quad \text{for} \quad n = 0, \ldots, N-1. \tag{2.2}$$

The initial conditions at the start of each sub-interval are given by $\mathbf{U}_n = \mathbf{u}_n(T_n)$, noting that $\mathbf{u}_0(T_0) = \mathbf{u}^0 = \mathbf{U}_0$ is already known. Next define a *coarse integrator* $\mathcal{G}$ that can integrate $\mathbf{U}_n$ forward in time, from T_n to T_{n+1}, using coarse time steps $\delta T < \Delta T$. Similarly, define a *fine integrator* $\mathcal{F}$ with much higher numerical accuracy than $\mathcal{G}$, that integrates $\mathbf{U}_n$ with finer time step $\delta t < \delta T$ over the same interval. The idea is that, if used to integrate (2.1) over $[T_0, T_N]$ serially, the high accuracy integrator $\mathcal{F}$ will take an infeasible amount of computational time. Whereas $\mathcal{G}$ must be chosen so that it integrates significantly faster than $\mathcal{F}$ in order for parareal to achieve speed up. The fast integrator $\mathcal{G}$ will always be run serially whereas $\mathcal{F}$ will be run in parallel because it is assumed to be the integrator that is too computationally heavy to run serially on $[T_0, T_N]$, hence the need for time parallelisation.

The initial values, $\mathbf{U}_n$, must therefore satisfy the system of equations (2.3) when subject to a (theoretical) *exact integrator* ϕ. This integrator works the same way as $\mathcal{F}$ and $\mathcal{G}$, propagating the solution forward in time from $\mathbf{U}_{n-1}$ to $\mathbf{U}_n$.

$$\mathbf{U}_0 = \mathbf{u}^0 \quad \text{and} \quad \mathbf{U}_n = \phi(\mathbf{U}_{n-1}) \quad \text{for} \quad n = 1, \ldots, N-1. \tag{2.3}$$

If the system was linear it could be solved explicitly, however assuming some nonlinearity exists, the equations can be resolved using Newton's method, to form the iterative system

$$\mathbf{U}_0^k = \mathbf{u}^0, \quad \mathbf{U}_n^k = \phi(\mathbf{U}_{n-1}^{k-1}) + \phi'(\mathbf{U}_{n-1}^k)\left[\mathbf{U}_{n-1}^{k-1} - \mathbf{U}_{n-1}^{k-1}\right] \quad \text{for} \quad n = 1, \ldots, N-1 \quad \text{and} \quad k \geq 0. \tag{2.4}$$

This result however, involves calculating the Jacobian, ϕ', of the exact integrator, which, if it was known, would still be computationally expensive to calculate. Instead of trying to find ϕ' explicitly, Lions et al. make a multigrid approximation using the fine and coarse solvers in place of the Jacobian. Recall that multigrid means taking two different levels of temporal discretisation to solve the problem (see Figure 1 again). The result is that the initial values $\mathbf{U}_n$ are iteratively improved at successive k using the predictor-corrector update rule given by

$$\mathbf{U}_n^k = \mathcal{F}(\mathbf{U}_{n-1}^{k-1}) + \left(\mathcal{G}(\mathbf{U}_{n-1}^k) - \mathcal{G}(\mathbf{U}_{n-1}^{k-1})\right). \tag{2.5}$$

The pseudo code for parareal (henceforth abbreviated as $\mathcal{P}$), in its original form, is detailed in Algorithm 1. With the aid of Figure 2, the first iteration of $\mathcal{P}$ can be visualised converging toward the fine solution. Following initialisation, $\mathcal{G}$ is used serially, in Step 2, over the entire time interval to calculate an initial guess to the solution. In Step 3(i), iteration $k = 1$ begins and the previously located solutions are propagated forward in time (in parallel) on each sub-interval by $\mathcal{F}$. The coarse solver is then used iteratively in Step 3(ii) to 'predict' the value of a solution at iteration $k = 1$, after which it is immediately 'corrected' by update rule (2.5). This process is carried out sequentially over all sub-intervals. Finally in Step 3(iii), if the difference between successive iterations is smaller than a given tolerance,

$$\|\mathbf{U}_n^k - \mathbf{U}_n^{k-1}\|_\infty < \epsilon, \tag{2.6}$$

at *all* time steps, the solution is considered converged and the algorithm stops. Otherwise it moves to the next k. Condition (2.6) signifies that further iterations of $\mathcal{P}$ will not improve the solution any more than it currently has and therefore avoids wasting computational resources.

Algorithm 1: Parareal ($\mathcal{P}$)

Step 1: Set counter $k = 0$ and define $\mathbf{U}_n^k$ as the solution at n^{th} time step and k^{th} iteration (recall that $\mathbf{U}_0^k$ is known $\forall k$).

Step 2: Calculate the initial guesses at the start of each sub-interval T_n using $\mathcal{G}$ serially.
$\hat{\mathbf{U}}_0^0 = \mathbf{U}_0^0$
for $n = 1$ **to** N **do**
$\hat{\mathbf{U}}_n^0 = \mathcal{G}(\hat{\mathbf{U}}_{n-1}^0)$
$\mathbf{U}_n^0 = \hat{\mathbf{U}}_n^0$
end for

Step 3: **for** $k = 1$ **to** N **do**

(i) Running $\mathcal{F}$ in parallel, propagate the solution on each sub-interval using previously calculated initial values.
for $n = 1$ **to** N **do**
$\tilde{\mathbf{U}}_n^{k-1} = \mathcal{F}(\mathbf{U}_{n-1}^{k-1})$
end for

(ii) Iteratively propagate the most up to date solution using $\mathcal{G}$. Then predict and correct the solution.
for $n = 1$ **to** N **do**
$\hat{\mathbf{U}}_n^k = \mathcal{G}(\mathbf{U}_{n-1}^k)$
$\mathbf{U}_n^k = \tilde{\mathbf{U}}_n^{k-1} + \hat{\mathbf{U}}_n^k - \hat{\mathbf{U}}_n^{k-1}$
end for

(iii) Check for solution convergence.
if $\|\mathbf{U}_n^k - \mathbf{U}_n^{k-1}\|_\infty < \epsilon \ \forall n$, **return** $\mathbf{U}^k$.
else, **next** k.

end for

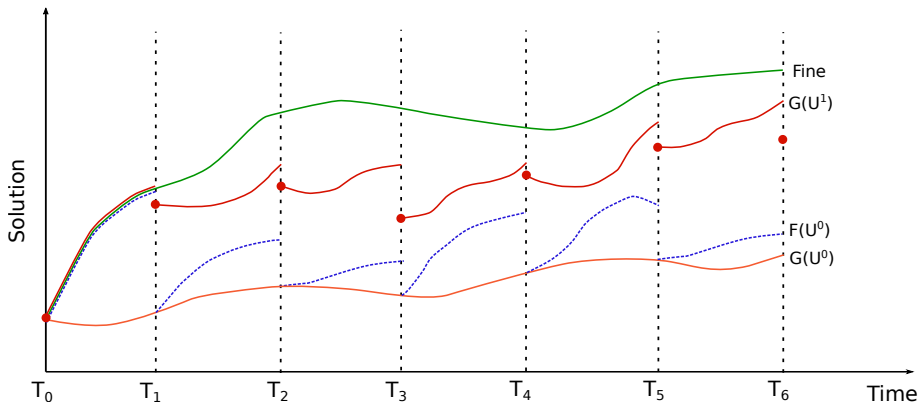


Figure 2: An artificial depiction of the first iteration of the parareal algorithm. The fine (true) solution is given in green; the first simulations of $\mathcal{G}$ and $\mathcal{F}$ in orange and blue respectively; and the second simulation of G in red. The red dots indicate the updated solution after applying (2.5).

It should be clear that the purpose of this algorithm is to locate the solution that would traditionally be integrated entirely sequentially using $\mathcal{F}$. This would take a computationally infeasible amount of time to simulate and therefore parareal suggests that by deploying $\mathcal{F}$ in parallel, using initial values calculated by a much faster and less accurate solver $\mathcal{G}$, a solution can be iteratively improved to converge toward the unknown fine solution in much more computationally feasible time.

At first glance it is not obvious that $\mathcal{P}$ should converge to a solution faster than the fine solver, however $\mathcal{P}$ requires N processors, one for each sub-interval $[T_n, Tn+1]$, in order to solve the problem. This ensures that $\mathcal{F}$ can be run in parallel over all sub-intervals simultaneously during each parareal cycle, therefore taking approximately $1/N$th of the usual time to run $\mathcal{F}$ over the entire time domain. This implies that, in the absence of communication time between processors (as well as any time consuming algorithm-specific processes), $\mathcal{P}$ could theoretically achieve perfect parallel speed-up if it was to converge in $k = 1$ iteration. If, however, $\mathcal{P}$ converges only on the final iteration, $k = N$, then this is equivalent to running $\mathcal{F}$ serially in all sub-intervals, and therefore over the entire time domain, yielding zero-parallel speed-up. In light of this, wall clock timings, which can be affected by the quality of processors one has, as well as algorithm specific run times, will not be the focus of this work. Instead the overarching aim is to minimise the number of iterations k that $\mathcal{P}$ converges to a solution in. More detailed analysis on the numerical convergence of parareal, as well as some wall clock timings for certain problem, can be found in [22, 23, 26].

One challenge in attempting to minimise the convergence rate of $\mathcal{P}$, is to identify an optimal coarse solver $\mathcal{G}$. It must be coarse enough, much more so than $\mathcal{F}$, to achieve significant speed-up however not too coarse as to allow the solution to lose numerical accuracy. Henceforth, a combination of single step, explicit second and fourth-order Runge Kutta methods[3] (RK2 and RK4 respectively) will be used within $\mathcal{P}$. Note that $\mathcal{G}$ and $\mathcal{F}$ can be chosen as the same solver, however a much coarser time step must be chosen for $\mathcal{G}$ than $\mathcal{F}$ in order to preserve algorithmic speed-up.

2.2. An application: The Brusselator

In order to test the convergence and accuracy of $\mathcal{P}$, the Brusselator system (2.7), a pair of coupled nonlinear ODEs that model an auto-catalytic chemical reaction, is chosen. With parameters $A = 1$ and $B = 3$, the system exhibits oscillatory behaviour that approaches a limit cycle as $t \to \infty$ and will therfore demonstrate that $\mathcal{P}$ can deal with stiff nonlinear ODEs.

$$\frac{dx}{dt} = A + x^2y - (B+1)x, \quad \frac{dy}{dt} = Bx - x^2y, \quad \text{with} \quad \mathbf{x}(0) = (0,1). \tag{2.7}$$

In the following simulations of $\mathcal{P}$, $t \in [0, 12]$ is divided into $N = 28$ sub-intervals with 280 coarse and 14,000 fine time steps respectively[4]. The convergence tolerance is set as $\epsilon = 10^{-6}$ and both coarse and fine solvers are always chosen to be RK4 methods, unless otherwise stated. A set of numerical results[5] are plotted in Figure 3.

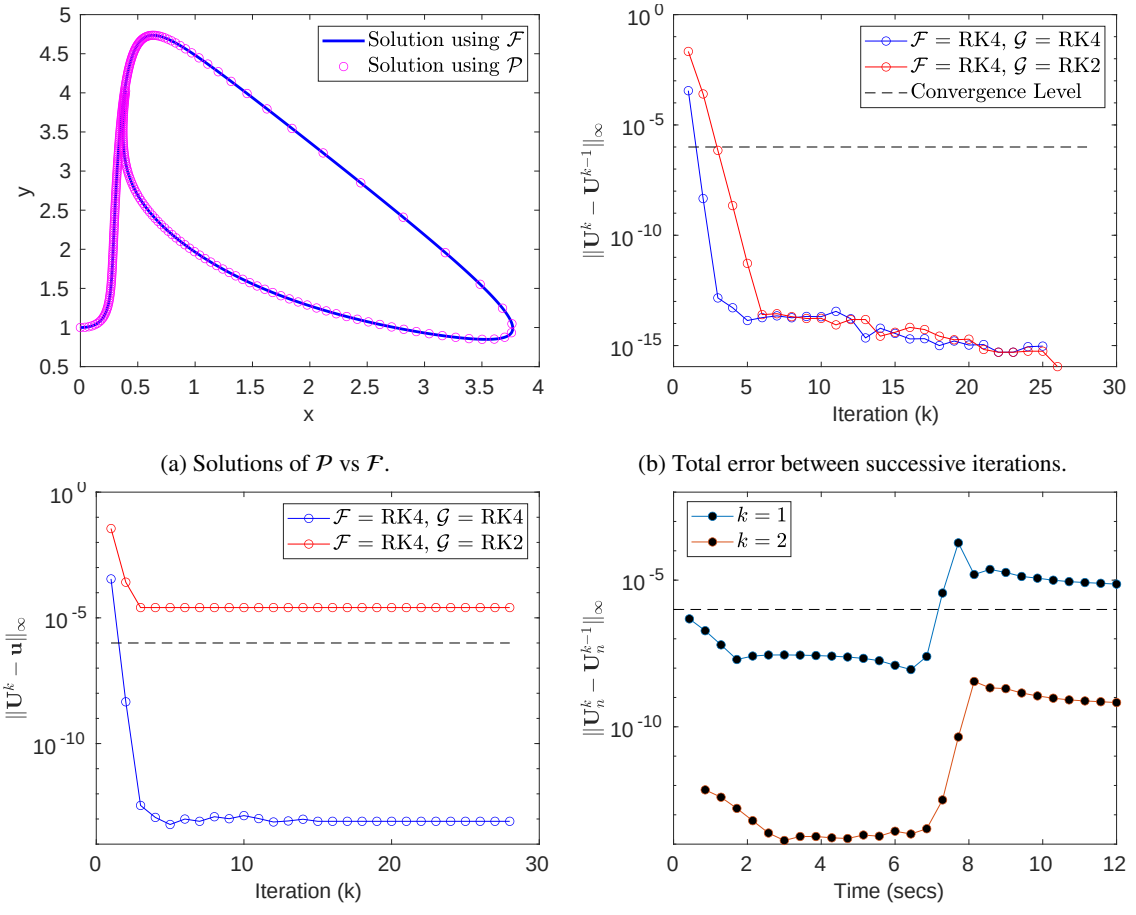
The solutions to (2.7), solved using $\mathcal{F}$ and $\mathcal{P}$ separately, are plotted together in Figure 3a and show excellent agreement, with both converging to the limit cycle known to exist in the phase space. It takes $\mathcal{P}$ just 2 (out of 28) iterations to converge to a sufficient accuracy with the coarse RK4 solver and 3 iterations with the coarse RK2 solver (see Figure 3b). This demonstrates that using the more accurate RK4 coarse solver helps the solution converge more quickly than the less accurate RK2 coarse solver. The accuracy of $\mathcal{P}$ compared to $\mathcal{F}$ is plotted in Figure 3c and shows that choosing RK4 as the coarse solver also yields a much more numerically accurate solution (as the RK2 solver cannot get as close to the fine solution). The final plot, Figure 3d, displays a more detailed version of Figure 3b where the errors between successive iterations (2.6) are plotted at each time step (ΔT) for successive iterations k. This again shows $\mathcal{P}$ successfully converging across all time steps with each further iteration k.

These simulations demonstrate that $\mathcal{P}$ computes a highly accurate solution (with coarse RK4 solver) and converges systematically after each iteration. The Brusselator is, however, a two-dimensional system with relatively stable solutions that do not diverge drastically if accuracy is not met. Next, $\mathcal{P}$ is run on a three-dimensional system where chaotic solutions are possible. Solutions that are initially close together, can diverge drastically in short time intervals, hence the accuracy of the solution found at each iteration becomes increasingly important to maintain.

[3] Alternative, more numerically stable, solvers such as multi-step and implicit methods can be used however tend to be relatively slow in this setting and introduce unnecessary complications upon implementation.

[4] Note how the number of coarse and fine time intervals must be multiples of N in order for interval boundaries to align correctly.

[5] Note that numerical simulations for all test problems henceforth have been run in parallel using up to a maximum of $N = 28$ compute cores on Warwick's high performance computing facilities (specifically the HPC designated Orac).



(a) Solutions of $\mathcal{P}$ vs $\mathcal{F}$.

(b) Total error between successive iterations.

(c) Error between $\mathcal{P}$ and $\mathcal{F}$.

(d) Errors between successive iterations across time.

Figure 3: Parareal applied to the Brusselator system. (a) Two dimensional trajectories determined using both the fine solver and parareal. Note that not all points of the solution of $\mathcal{P}$ have been plotted for comparison purposes. (b) The total error between successive iterations, k, of $\mathcal{P}$. (c) The numerical error between $\mathcal{P}$ and $\mathcal{F}$ using two different coarse solvers (RK2 and RK4) and RK4 fine solver. (d) The same errors as (b), when using a coarse RK4 solver, plotted across each time step for each iteration k. Convergence is met when the errors fall below the dashed line at 10^{-6}.

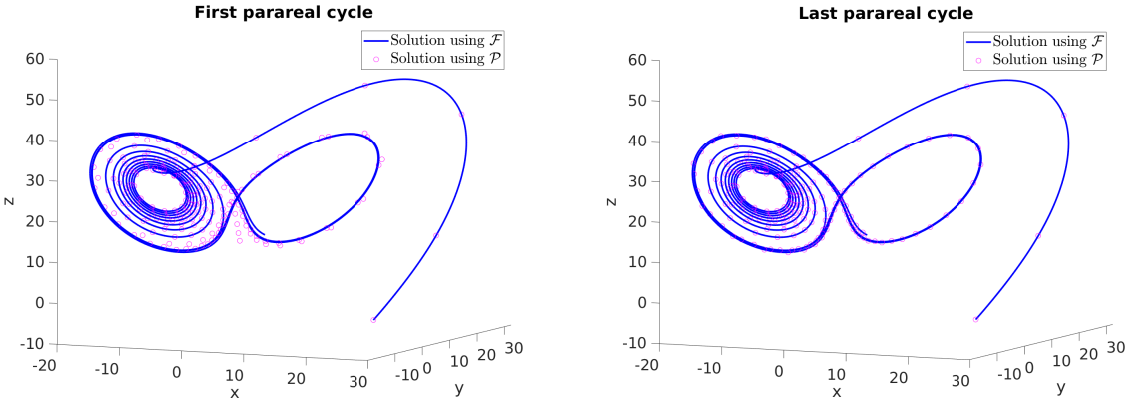
2.3. An application: The Lorenz Equations

Parareal is now applied to the Lorenz system (2.8), a set of nonlinear ODEs that demonstrate how small perturbations to initial values can cause solutions to evolve in remarkably different ways [27]. The parameters $\sigma = 10$, $\rho = 28$ and $\beta = 8/3$ are chosen in order to investigate how $\mathcal{P}$ performs as solutions approach a chaotic attractor. Subjecting the system to the same initial conditions as in [23], the time interval $t \in [0, 10]$ is divided into $N = 25$ sub-intervals with 2,500 coarse and 15,000 fine time steps respectively. This system was chosen because it evolves in a highly chaotic manner where small numerical errors at a given time step can propagate through the solution, resulting in vastly different final states.

$$\frac{dx}{dt} = \sigma(y - x), \quad \frac{dy}{dt} = \rho x - xz - y, \quad \frac{dz}{dt} = xy - \beta z \quad \text{with} \quad \mathbf{x}(0) = (20, 5, -5). \tag{2.8}$$

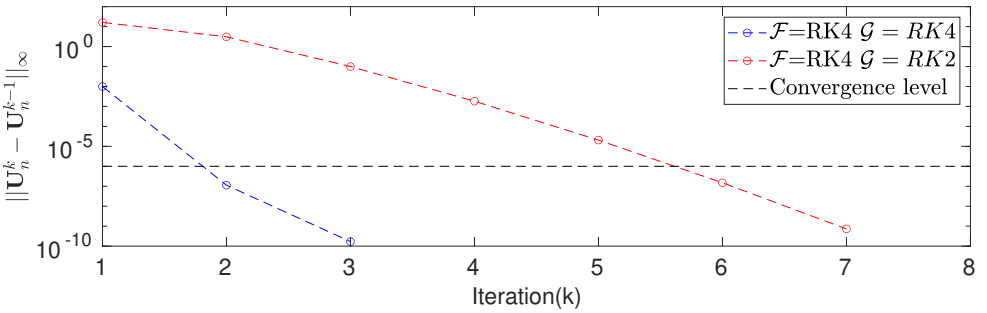
Figure 4 demonstrates how $\mathcal{P}$ solves the system. Starting from the initial guess, the solution after the first parareal iteration, $k = 1$, is compared to fine solution in Figure 4a. It is observed that, close to the initial condition and after

a short time interval, the solution matches the 'true' solution, given by $\mathcal{F}$, very well. As time increases however, the parareal solution begins to deviate from $\mathcal{F}$ as the initial conditions calculated at the start of each sub-interval are not yet accurate enough. As the solution diverges, this error increases with increasing time due to the chaotic nature of the system. Nevertheless, $\mathcal{P}$ continues to improve the solution at these later (unconverged) time intervals with further iterations up to $k = 2$, upon which it stops. Figure 4b shows the converged solution found by $\mathcal{P}$ matches the fine solver almost perfectly.



(a) Solutions of $\mathcal{P}$ vs $\mathcal{F}$ at $k = 1$.

(b) Solutions of $\mathcal{P}$ vs $\mathcal{F}$ at $k = 2$



(c) Total error between successive iterations.

Figure 4: (a) Solution from $\mathcal{P}$ after $k = 1$ iterations (pink circles) vs the solution obtained using $\mathcal{F}$. (b) The same as (a) but after $k = 2$. (c) The error between successive iterations, k, of $\mathcal{P}$ for the two different coarse solvers. Note the x-axis range has been adjusted, for the algorithm stops when convergence criteria are met.

For test purposes again, the RK4 coarse solver is replaced with an RK2 method, which has local truncation error $\mathcal{O}(\delta T^3)$, in order to see if convergence is improved. In Figure 4c observe that by choosing RK2 as the coarse solver, the convergence rate increases drastically from $k = 2$ to $k = 6$. This occurs because the initial guesses calculated by $\mathcal{G}$ are now much less accurate, which is known to be incredibly detrimental in a chaotic system, hence it takes more time for $\mathcal{P}$ to attain the required accuracy. Both methods do however demonstrate rapid convergence at a near-exponential rate, as shown in Figure 4c.

This analysis demonstrates that $\mathcal{P}$ can converge quickly and accurately for highly nonlinear systems where errors on sub-interval boundaries need to be very small. This becomes very important when probabilistic methods are used within $\mathcal{P}$ as multiple sampled solutions are taken at these sub-interval boundaries that will generate very different solution trajectories when propagated by one of the numerical solvers. Therefore if solutions can converge for such a chaotic system, they should therefore perform well for other more stable nonlinear systems. Next, the modified stochastic parareal algorithm is explained in detail.

3. The Stochastic Parareal Algorithm

At first, the idea of replacing a deterministic numerical integrator with one that relies on generating solutions from a given probability distribution seems rather counter-intuitive. The aim is to improve the convergence rate of $\mathcal{P}$ by incorporating the statistical differences between solutions generated by $\mathcal{F}$ and $\mathcal{G}$ at each sub-interval. Therefore, instead of calculating a single solution at each sub-interval using update rule (2.5), the stochastic parareal algorithm ($\mathcal{P}_s$) samples multiple solutions from a specified probability distribution. This distribution will be based on known coarse and fine solutions at the time. Following this, $\mathcal{P}_s$ can then piece together a continuous solution across the intervals, from an ensemble of trajectories, thereby increasing the likelihood of locating the 'true' continuous solution. Think of this as a multiple shooting method with multiple 'shots' being taken at each time sub-interval instead of just one. If this method can be demonstrated, with high probability, to converge in fewer iterations than the deterministic parareal, it can be assured that speed-up has been achieved.

3.1. The Algorithm

The new method largely follows the original structure of $\mathcal{P}$. Solutions are propagated forward in time, alternating between the coarse and fine solvers, however the way in which the solutions are used to update the final solution at the end of each iteration has changed. Algorithm 2 contains detailed pseudocode of $\mathcal{P}_s$. At iteration $k = 0$, $\mathcal{G}$ is used as before to determine an initial guess for the solution and $\mathcal{F}$ immediately follows, propagating these guesses forward in parallel. Previously, $\mathcal{G}$ would be used again to predict the updated solution before correction with rule (2.5), however this method is no longer used. In order to sample solutions at $k = 1$, another run of $\mathcal{G}$ is required using initial values from the most accurate current solution (at this iteration, the solutions from $\mathcal{F}$).

Following this, enough information is now known to begin sampling solutions. In order to get close to the 'true' solution at time step n, sample $m = 1, \dots, M$ solutions, named $\bar{\mathbf{U}}^1_{n_m}$, from a particular probability distribution, say (3.1), in parallel. The distributions being used are referred to henceforth as sampling rules and are discussed in Section 3.2. For each sampled solution, propagate it using the fine solver, still in parallel, to obtain M candidate trajectories, $\mathcal{F}(\bar{\mathbf{U}}^1_{n_m})$ for $m = 1, \dots, M$ (see Figure 5). The chosen trajectory is the one that minimises the error between its starting condition and the most accurate solution from the endpoint of a trajectory in the previous sub-interval. This process of selection is carried out sequentially (see Step 4(iii) in Algorithm 2) and in doing so, continuity of the solution across sub-intervals is assured. See Figure 6 for a depiction of how the continuous solution is pieced together from the sets of candidate trajectories in each sub-interval (note that sampling is only carried out for unconverged time intervals). Given that the first interval $[T_0, T_1]$ is converged during $k = 0$, there is no need to sample at the boundary T_1 during iteration $k = 1$, hence only $N - 1$ intervals are initially unconverged at the start of $k = 1$. Therefore $\mathcal{F}$ can be run immediately from the converged solution at T_1. It follows that the maximum number of samples generated by $\mathcal{P}_s$ is $M(N - 2)$, occurring at the first iteration $k = 1$.

This ensures that within $\mathcal{P}_s$, only unconverged sub-intervals are improved at any particular k, with the entire process stopping once all N sub-intervals are converged. This differs slightly from $\mathcal{P}$ where the solution is improved across all sub-intervals regardless of whether some had converged or not, recall Figure 3d demonstrating this property. This improved functionality enables the algorithm to consider multiple steps converged, up to time T_I say, in a single iteration, before moving on to the unconverged intervals beyond time T_I in the following iteration. This saves computational resources and will be demonstrated more clearly in Section 4.1.

It should be noted that the sampling and subsequent propagation of all candidate trajectories, at any iteration k, are carried out in parallel to ensure rapid computation of the fine solver. Therefore in order to preserve and achieve speed-up, $\mathcal{P}_s$ requires at least $M(N - 2)$ processors when solving, significantly more than $\mathcal{P}$ requires. It is assumed however that a large enough number of compute cores are available to solve these sorts of problems. The choice of both $\mathcal{G}$ and $\mathcal{F}$ remain an important factor in the level of speed-up one wishes to attain. The less accurate a solver $\mathcal{G}$ is, the more iterations $\mathcal{P}_s$ will require in order to achieve the level of accuracy specified by ϵ.

Due to the stochastic nature of $\mathcal{P}_s$, each simulation of the algorithm will converge in a different number of iterations $k_s \in \{1, \dots, N\}$. This makes analytical convergence analysis difficult and therefore statistical methods are employed to determine the rate of convergence. Such methods include running multiple simulations of $\mathcal{P}_s$ on the same problem (with a fixed number of samples M) in order to determine a convergence distribution for k_s. This will demonstrate, with some probability, whether $\mathcal{P}_s$ performs better or worse than the fixed convergence rate k_d of $\mathcal{P}$.

The number of sampled solutions taken at each step also needs to be optimised. If M is chosen too small, sampling will be ineffective and solutions are unlikely to 'jump' close to the true solution. On the other hand, a saturation point

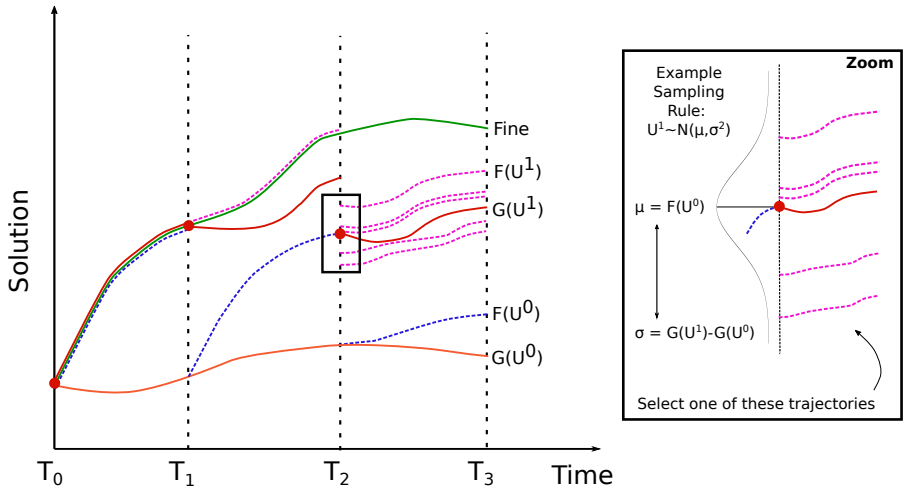


Figure 5: An artificial depiction of the first iteration of $\mathcal{P}_s$. The initial guess by $\mathcal{G}$ and subsequent $\mathcal{F}$ runs at $k = 0$ are given in orange and dashed blue respectively. At $k = 1$, $\mathcal{G}$ is run again using these $\mathcal{F}$ values, see solid red trajectories. Next, M solutions are sampled around the fine solution using (3.1) at time step T_2 and then propagated forward, see purple dashed trajectories. Note that no samples are taken at T_1 as the preceding trajectory is fully converged.

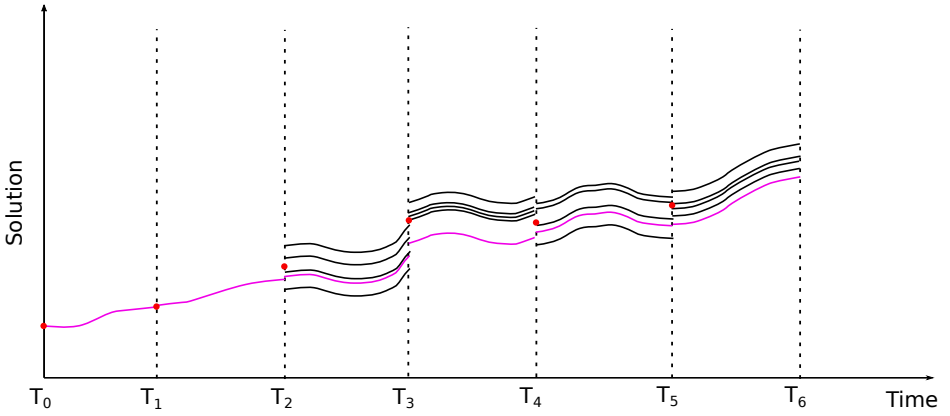


Figure 6: An artificial depiction of the sampled trajectories ($M = 5$ in this case) generated by $\mathcal{P}_s$ at each unconverged sub-interval (T_2 and beyond) at iteration $k = 1$. Samples are taken near the red dots (i.e. the mean of the sampling rule (3.1)) and a continuous solution is determined serially by minimising the difference between trajectories on the boundaries at increasing time steps.

will inevitably be reached for large M, beyond which sampled solutions are so close together they go to waste. By analysing the distributions of k_s for increasing M, an optimal number (or range) for M and therefore the number of processors required can be determined.

After locating an optimal range for M, an ensemble of stochastic solutions, generated by multiple independent runs of $\mathcal{P}_s$, can be used to test accuracy against the fine solution. This becomes very important when simulating solutions from chaotic systems such as the Lorenz equations. The stochastic nature of $\mathcal{P}_s$ will generate slightly different solution trajectories in phase space with each independent run, hence by observing the spread of the solutions at different time slices across the domain, the accuracy and statistical distribution of the ensemble trajectories can be understood more clearly.

3.2. Sampling Rules

The main reason for using sampling rules to generate solutions is to exploit the statistical differences that are inherently generated by the coarse and fine solvers for numerical gain. Therefore the known solutions (at iteration k)

identified by $\mathcal{G}$ and $\mathcal{F}$ provide good starting positions from which to begin sampling. There are many distributions that could be chosen for sampling, however the normal distribution is naturally suited to sampling solutions near to the mean as long as a suitable standard deviation can be chosen. It a good choice for an initial sampling rule, however for problems that exhibit rapidly growing or decaying solutions, a more skewed or heavy-tailed distribution may be more appropriate. Depending on whether the solutions at $\mathcal{G}$ or $\mathcal{F}$ are more accurate at iteration k, either of the following rules may be more suitable for a particular problem than the other.

The sampling rules are given below at a generic time step n and iteration k. Recall notation from Algorithm 2 where $\hat{\mathbf{U}}$ and $\tilde{\mathbf{U}}$ refer to solutions calculated by the coarse and fine solvers respectively.

$$\text{Rule 1:} \quad \left[\bar{\mathbf{U}}^k_{n_1}, \dots, \bar{\mathbf{U}}^k_{n_M}\right] \sim \mathcal{N}\left[\tilde{\mathbf{U}}^{k-1}_n, \left(\hat{\mathbf{U}}^k_n - \hat{\mathbf{U}}^{k-1}_n\right)^2\right] \tag{3.1}$$

$$\text{Rule 2:} \quad \left[\bar{\mathbf{U}}^k_{n_1}, \dots, \bar{\mathbf{U}}^k_{n_M}\right] \sim \mathcal{N}\left[\hat{\mathbf{U}}^k_n, \left(\tilde{\mathbf{U}}^{k-1}_n - \hat{\mathbf{U}}^{k-1}_n\right)^2\right] \tag{3.2}$$

Solutions generated by sampling rule (3.1) are centred around a mean of the most accurate fine solution available at $k-1$, with a standard deviation of the absolute difference of the two coarse solutions at k and $k-1$. This works well for stable problems that require a relatively small standard deviation, with samples clustered close to the mean value.

Sampling rule (3.2) instead generates solutions close to the most accurate coarse solution available at iteration k. Whilst this may seem counter-intuitive, the standard deviation associated with this rule is much larger, set as the absolute difference between the fine and coarse solutions at $k-1$, enabling the algorithm a chance to select an optimal solution as close to or as far from $\mathcal{G}$ as it needs to be to ensure continuity. This may be more suitable for unstable systems of equations that vary rapidly on short timescales. Both rules will however require an increasingly larger M in order for solutions far from the mean to be well sampled (if required).

Depending on the nature of the problem being solved, alternative probability distributions may be suitable candidates for sampling rules, however for the purposes of this report, rules (3.1) and (3.2) are general enough to analyse the performance of $\mathcal{P}_s$ on the Brusselator and Lorenz test problems.

Algorithm 2: Stochastic Parareal ($\mathcal{P}_s$)

Step 1: Set counters $k = 0$, $I = 0$ and define $\mathbf{U}_n^k$ as solution at n^{th} time step and k^{th} iteration (recall that $\mathbf{U}_0^k$ is known for all k).

Step 2: Calculate the initial guesses at the start of each sub-interval T_n using $\mathcal{G}$ serially.
$\hat{\mathbf{U}}_0^0 = \mathbf{U}_0^0$
for $n = 1$ **to** N **do**
$\hat{\mathbf{U}}_n^0 = \mathcal{G}(\hat{\mathbf{U}}_{n-1}^0)$
end for

Step 3: Running $\mathcal{F}$ in parallel, propagate the solution on each sub-interval using previously calculated initial values.
for $n = 1$ **to** N **do**
$\tilde{\mathbf{U}}_n^0 = \mathcal{F}(\hat{\mathbf{U}}_{n-1}^0)$
$\mathbf{U}_n^0 = \tilde{\mathbf{U}}_n^0$
end for
$I = 1$ (as solution up to T_I is now converged)

Step 4: **for** $k = 1$ **to** N **do**

(i) (*Coarse solve - serial*)
for $n = I + 1$ **to** N **do**
$\hat{\mathbf{U}}_n^k = G(\mathbf{U}_{n-1}^{k-1})$
end for

(ii) (*Fine step - parallel*)
for $n = I + 1$ **to** N **do**
if $n = I + 1$, $\mathbf{U}_n^k = \mathcal{F}(\mathbf{U}_{n-1}^{k-1})$ (as $\mathbf{U}_{n-1}^{k-1}$ has converged)
else, generate M solution samples from one of the sampling rules (see Section 3.2). Propagate each one to obtain M trajectories $\mathcal{F}(\bar{\mathbf{U}}_{n_m}^k)$.
end for

(iii) (*Find continuous solution*)
for $n = I + 2$ **to** N **do**
$\bar{m} = \underset{m}{\text{argmin}} \, \|\bar{\mathbf{U}}_{n_m}^k - \mathcal{F}(\mathbf{U}_{n-1}^k)\|_\infty$
$\mathbf{U}_n^k = \bar{\mathbf{U}}_{n_{\bar{m}}}^k$
end for

(iv) (*Error check*)
$I = I + 1$ (as solution up to T_I is now converged)
for $j = I + 1$ **to** N **do**
if $\|\mathbf{U}_j^k - \mathbf{U}_j^{k-1}\|_\infty < \epsilon$,
$I = I + 1$
save $\mathbf{U}_j^k$ $\forall k$
else, **break**
end for
if $I = N$, **return** $\mathbf{U}^k$ (all intervals converged up to T_N)
else, **next** k

end for

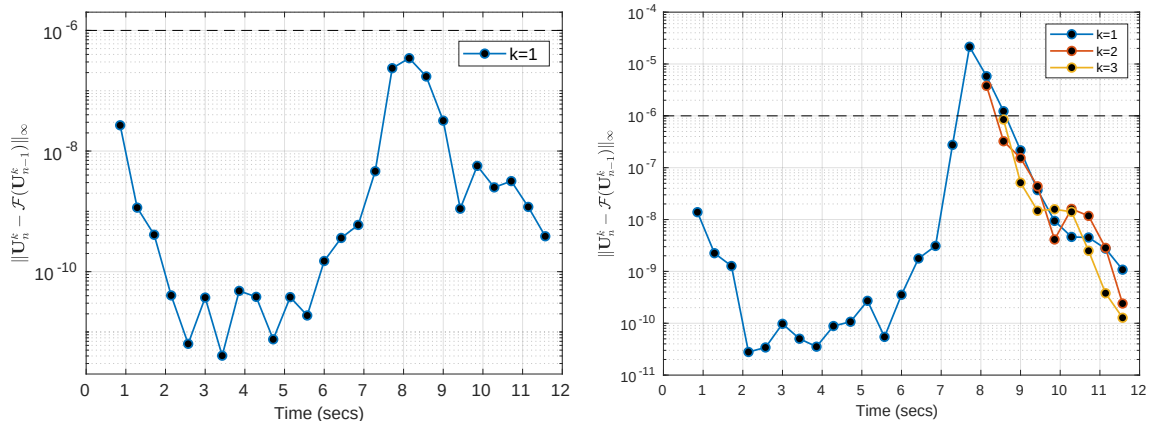
4. Analysis of Stochastic Parareal

In the following sections, results are obtained by applying $\mathcal{P}_s$ to the previously described test problems outlined in Sections 2.2 and 2.3. Although $\mathcal{P}_s$ requires at least $M(N-2)$ processors to achieve wall clock speed-up against the fine solver, the number of processors has no effect on the convergence rate or the solution accuracy. Hence the following simulations are still run on 28 compute cores. To ensure consistency between comparisons of $\mathcal{P}$ and $\mathcal{P}_s$, parameters values and initial conditions for both problems are held fixed. The coarse and fine solvers are also selected both as RK4 methods, with $\mathcal{G}$ having a much larger time step than $\mathcal{F}$. Recall that $\mathcal{P}$ converged in $k_d = 2$ iterations in both the Brusselator and Lorenz systems, making this the target for $\mathcal{P}_s$ to beat. Also recall that a new parameter M has been introduced which will be analysed using both of the selected sampling rules independently. For ease of discussion, equations (3.1) and (3.2) will henceforth be referred to as rule 1 and rule 2 respectively.

4.1. Performance on the Brusselator

Before analysing the results, it is helpful to observe how $\mathcal{P}_s$ converges toward a continuous solution. Similarly to Figure 3b, the continuity errors produced at each iteration (and each ΔT) following a single simulation of $\mathcal{P}_s$ are plotted in Figure 7. Recall that for $\mathcal{P}_s$ to consider a time interval converged, this error must be below $\epsilon = 10^{-6}$. These plots demonstrate (for rule 1 at least) that $\mathcal{P}_s$ can beat the convergence rate, $k_d = 2$, achieved by $\mathcal{P}$. It is not however always guaranteed to perform better. Recall that its stochastic nature will cause the the convergence rate k_s and the final solution itself to change with each independent simulation. Therefore it becomes necessary to investigate the distribution of k_s produced by multiple realisations of P_s with varying M.

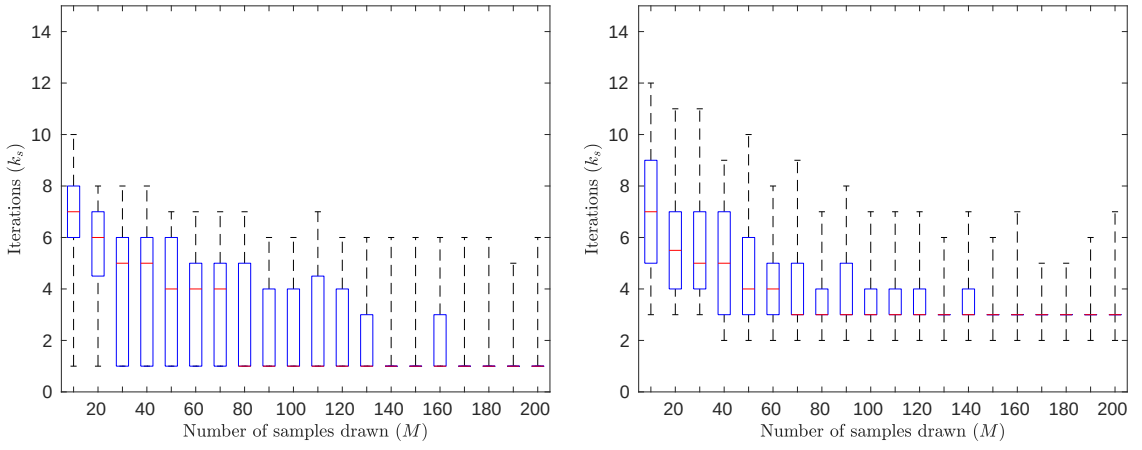
With this in mind, using rule 2, $\mathcal{P}_s$ converges only until the first 16 time intervals in the first cycle, such that subsequent iterations are required for full convergence. Notice that peak error occurs at a time (approximately) close to where the ODE solution becomes stiff. Additional time intervals N could potentially reduce this error spike. Two more subsequent iterations show rule 2 reaching the required tolerance in $k_s = 3$ iterations for this single run of $\mathcal{P}_s$.



(a) Rule 1 (b) Rule 2

Figure 7: Continuity errors of the Brusselator solution, during a single run of $\mathcal{P}_s$, at sub-interval boundaries. Successive iterations of $\mathcal{P}_s$ are plotted using (a) rule 1 and (b) rule 2 with $M = 70$ samples in both. The algorithm converges once all errors are below $\epsilon = 10^{-6}$.

Having demonstrated convergence, the algorithm's stochastic nature must now be investigated. Box plot distributions of the convergence rate k_s, for 100 independent simulations of $\mathcal{P}_s$, are plotted for increasing sample number in Figure 8. The plots demonstrate the effect that increasing M has, not only on the median value of k_s, but also on the range and interquartile ranges (IQRs). For both rules, a much larger M increases the probability of sampling from the tails of the normal distributions, hence increasing the probability that $\mathcal{P}_s$ hits a continuous solution. The decreasing medians with increasing M demonstrate this effect. Whilst rule 2 exhibits much more consistent convergence, with smaller IQRs, the minima show that it is unable to reach $k_s = 1$ and therefore cannot beat $\mathcal{P}$. Rule 1 on the other hand much more consistently beats $\mathcal{P}$, with with median values and ranges encompassing $k_s = 1$ for any M, much more so for larger values however.



(a) Rule 1 (b) Rule 2

Figure 8: Box plot distributions of the number of iterations k_s obtained by running $\mathcal{P}_s$ independently 100 times with (a) rule 1 and (b) rule 2. The number of samples M are increased in increments of 10. Whiskers indicate minimum and maximum values, blue boxes give the interquartile ranges and solid red lines indicate the median number of iterations.

In particular the fraction of simulations of $\mathcal{P}_s$ that converge at the same rate, or faster, than $\mathcal{P}$ is a useful metric. For high numbers of independent simulations, these fractions can be thought of as the probability that $\mathcal{P}_s$ beats or equals the convergence rate of $\mathcal{P}$. As can be seen in Figure 9, the probabilities increase almost monotonically with increasing M for both sampling rules. Rule 1 outperforms rule 2 at every value of M, tending to probability 1 as M increases to $M = 370$.

Rule 1 outperforms rule 2 because it has a much smaller variance, suited to stable systems like the Brusselator where solutions initially close together do not diverge from one another after short evolution periods. With much higher variance, rule 2 has a much larger space from which to sample solutions, hence requiring many more samples to achieve a higher probability of beating or equalling $\mathcal{P}$. This shows that rule 1 requires fewer samples (and therefore processors) than rule 2 in order to perform as well as $\mathcal{P}$.

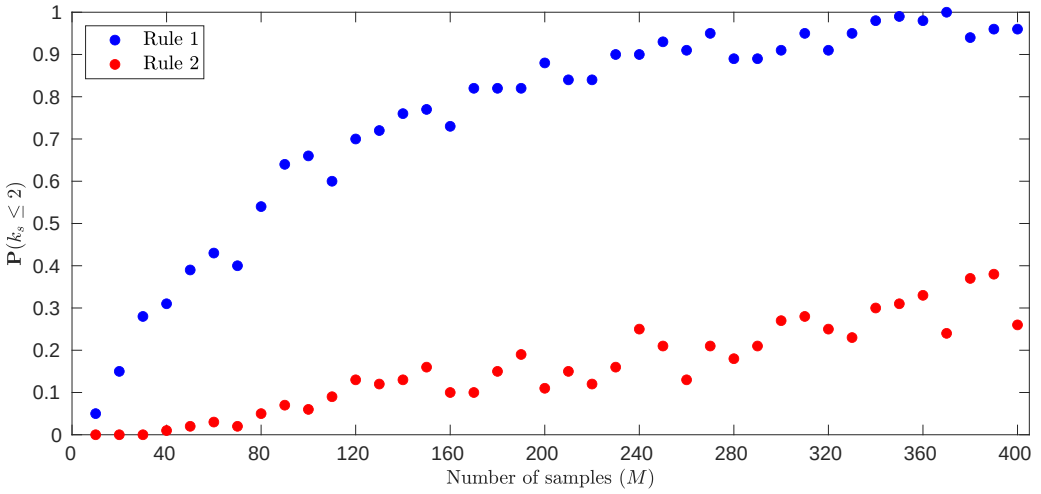


Figure 9: The discrete probabilities that $\mathcal{P}_s$ beats, or equals, the convergence rate of $\mathcal{P}$ as the sample number M is increased. Probabilities for rules 1 and 2 are given in blue and red respectively.

The set of solutions produced by 250 independent simulations of $\mathcal{P}_s$ (using rule 1 with $M = 100$) are now analysed. Plotting all 250 independent realisations of the Brusselator trajectories in phase space is not very informative as the

standard deviation between the trajectories at any time point is always below 10^{-5}. Therefore to measure accuracy, the average (mean) trajectory is identified and compared to the 'true' solution given by the fine solver. The error between these trajectories as well as the error between the trajectory found using $\mathcal{P}$ and the fine solution are plotted in Figure 10. Notice how the two plots match incredibly well, with the errors bounded above by approximately 10^{-4}. To improve accuracy further, one would need to increase the number of coarse time steps within $\mathcal{P}_s$, however this will come at the cost of a higher wall clock time upon simulation.

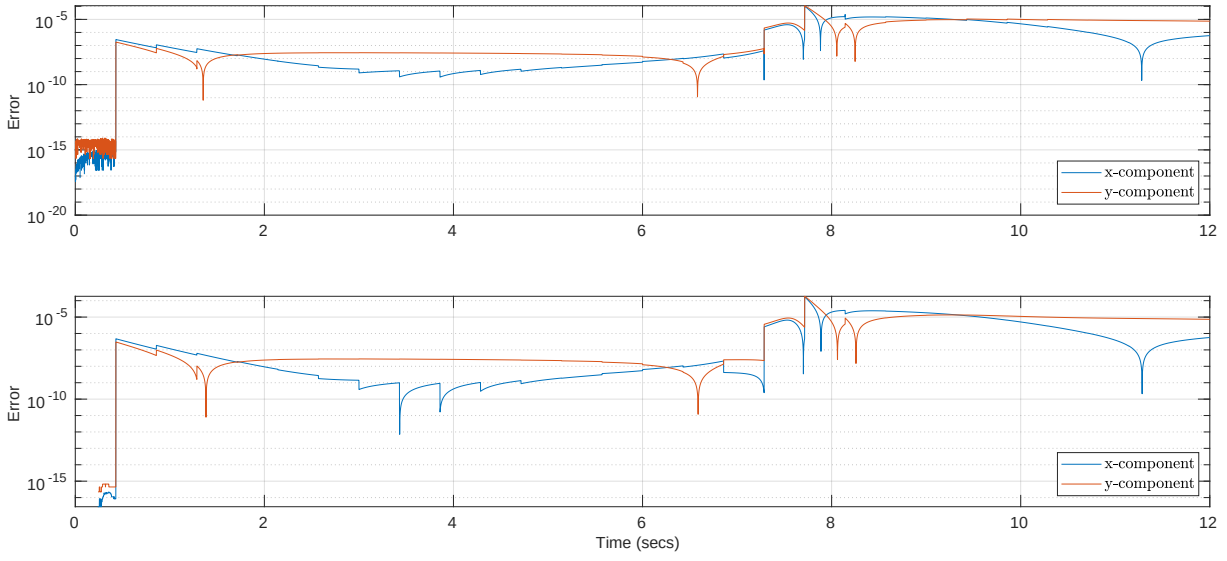


Figure 10: (Top panel) The error between the average stochastically generated trajectory from $\mathcal{P}_s$ and the fine solution of the Brusselator. (Bottom panel) The same, except the trajectory is generated deterministically by $\mathcal{P}$.

4.2. Performance on the Lorenz system

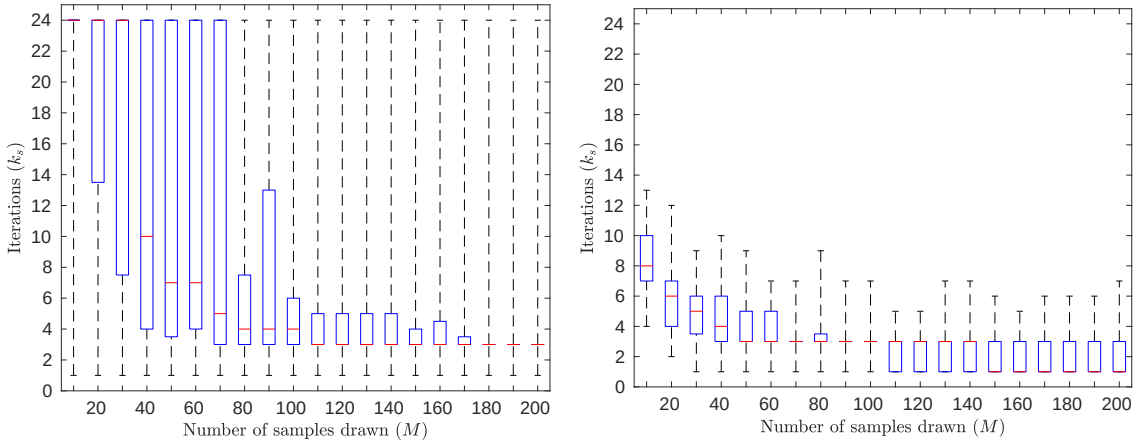
Similarly to Section 4.1, $\mathcal{P}_s$ is demonstrated to perform at least as well, if not better than $\mathcal{P}$, requiring at best, $k_s = 1$ iteration to converge. The box plot distributions for k_s in Figure 11a however, show that using rule 1, some simulations of $\mathcal{P}_s$ often take the full $k = 24$ iterations to converge to a solution. These simulations of $\mathcal{P}_s$ are as slow as running $\mathcal{F}$ serially and clearly are not a viable solution to the problem. Such a large range demonstrates that the variance of rule 1 is clearly too small to sample effectively in this chaotic system. This decreases the probability that a continuous solution will be located, unless M is increased drastically. Even for $M = 200$, the full range $k_s \in \{1, \ldots, 24\}$ still exists.

Rule 2 on the other hand (see Figure 11b) performs much better, converging within a much smaller range of iterations for all tested values of M. This exemplifies the idea that a higher variance for a more chaotic system increases the probability that a solution is found. The chaotic nature of the Lorenz system inherently demands a larger variance as trajectories diverge drastically from one another over short intervals.

Following this, the fraction of simulations that perform as well or better than $\mathcal{P}$ are displayed in Figure 12. As expected, rule 2 is clearly the obvious choice for solving this system rapidly, with the probability approaching almost certainty as sample numbers reach approximately $M = 400$. Whilst the probability does plateau at around 0.9 after $M = 270$ samples, the range given by the corresponding box plots show that additional iterations are kept to a minimum. Much like the Brusselator, increasing M further can increase the probability to almost certainty if the required processors are available. Using rule 2, remarkable agreement between the average trajectory, out of 250 independent simulations, of $\mathcal{P}_s$ and the fine solution is demonstrated once again in Figure 13. The x, y and z errors correspond very well to the ones deterministically generated by $\mathcal{P}$ as well.

To observe the stochastic nature of the $\mathcal{P}_s$ trajectories more clearly, all 250 solutions generated are plotted in phase space in Figure 14a. The standard deviation is known to be minimal, on scales of 10^{-4} to 10^{-10}, hence trajectories overlap and do not clearly deviate from one another in this plot. However if magnified at specific times, 'clouds' of scattered discrete points of each of the 250 trajectories can be viewed in 3D space (see Figures 14b-14d). Whilst

individual trajectories from P_s do indeed deviate on tiny scales, they preserve accuracy when compared to the solution from deterministic parareal (see blue points in the previous Figures). What this demonstrates is that $\mathcal{P}_s$ has the ability to rapidly compute large numbers of trajectories for highly nonlinear and chaotic systems whilst preserving solution accuracy.



(a) Rule 1 (b) Rule 2

Figure 11: Box plot distributions of the number of iterations k_s obtained by running $\mathcal{P}_s$ independently 100 times with (a) rule 1 and (b) rule 2 on the Lorenz system. The number of samples M are increased in increments of 10. Whiskers indicate minimum and maximum values, blue boxes give the interquartile ranges and solid red lines indicate the median number of iterations.

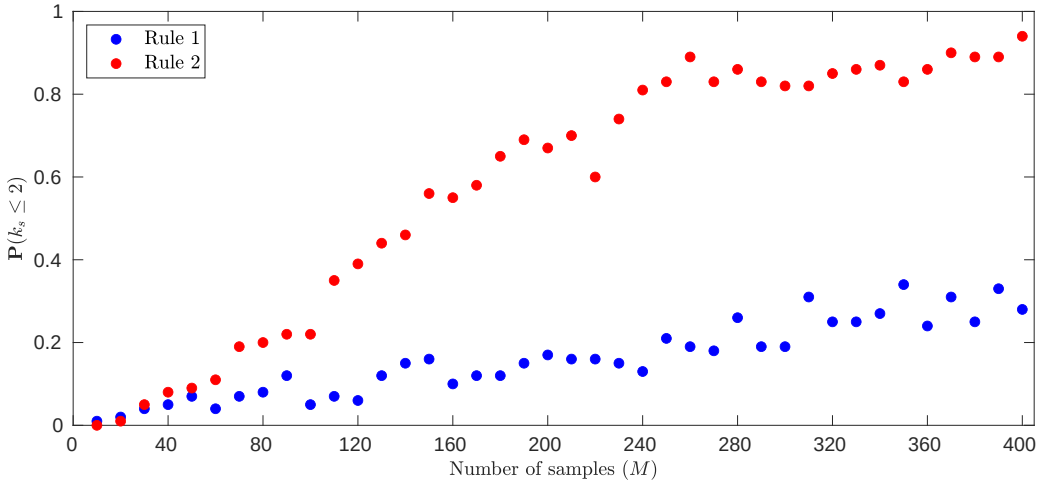


Figure 12: The discrete probabilities that $\mathcal{P}_s$ beats, or equals, the convergence rate of $\mathcal{P}$ as the sample number M is increased. Probabilities for rules 1 and 2 are given in blue and red respectively.

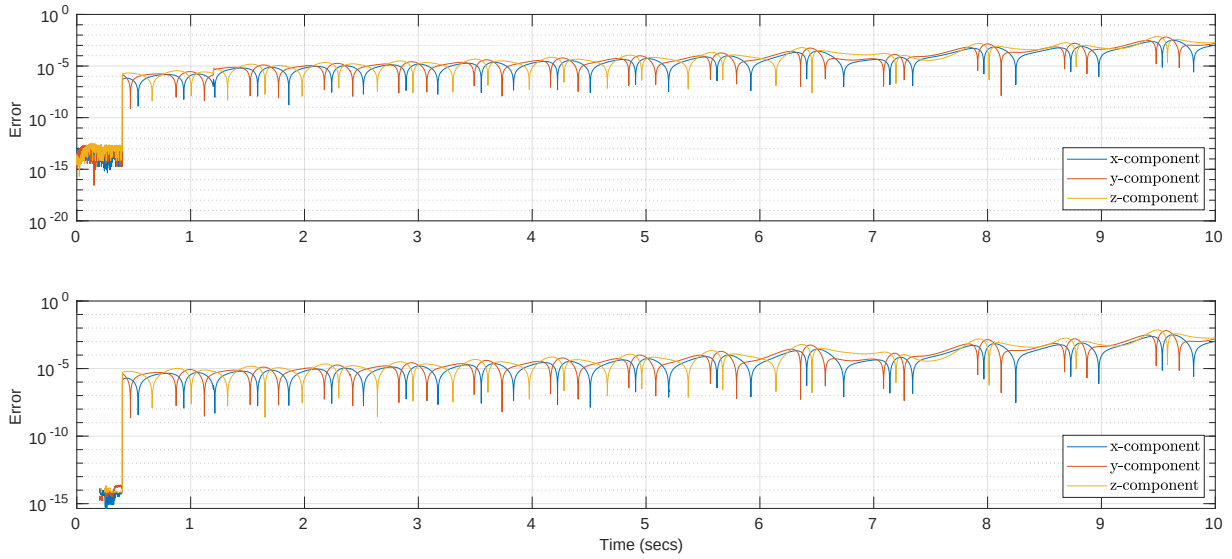


Figure 13: (Top panel) The error between the average stochastically generated trajectory from $\mathcal{P}_s$ and the fine solution of the Lorenz system. (Bottom panel) The same, except the trajectory is generated deterministically by $\mathcal{P}$.

5. Conclusions

It has been demonstrated that the stochastic parareal algorithm $\mathcal{P}_s$ has the potential, under specific conditions, to outperform its deterministic counterpart $\mathcal{P}$ with a varying degree of probability. This probability is based on a number of factors, namely the number of time intervals N, the number of samples taken at each sub-interval M, the type of problem being solved and the type of sampling rule in use.

In the case of both the Brusselator and Lorenz systems, it was shown that $\mathcal{P}_s$ could converge in just a single iteration $k_s = 1$. As mentioned in Section 2, this indicates that, in the absence of algorithm-specific inefficiencies, the stochastic parareal algorithm has the potential to offer perfect parallel speed-up (when compared to the fine solver $\mathcal{F}$). It must be noted that for much larger problems in the literature, $\mathcal{P}$ does not usually converge so quickly and hence by testing on such small systems of ODEs, beating $\mathcal{P}$ was to be challenging. The fact that it could however be beaten in such few iterations shows that the prospects this new method provide for larger-scale applications are nonetheless promising. More interestingly was that for a large enough number of samples, the probability to converge in less than or equal the same number of iterations as deterministic $\mathcal{P}$ approached almost certainty. The stochastic trajectories were also shown clustering in clouds around the solution found using $\mathcal{P}$, showing that accuracy of the stochastic solutions was preserved.

It can be concluded that the performance of $\mathcal{P}_s$ was highly dependent upon the chosen sampling rule, which itself needed to be chosen dependent on the type of problem being solved. The preliminary analysis suggests that more unstable and chaotic systems like the Lorenz equations, require a larger variance to converge quickly toward a solution whereas those that are more stable, like the Brusselator, required a smaller variance. Both sampling rules 1 and 2 however required a large number of generated samples M to ensure that $\mathcal{P}_s$ converged sooner or at the same time as $\mathcal{P}$. Clearly there is an optimal balance that needs to be struck between the variance of the sampling rule and the number of samples taken, depending on the problem in question. It must also be recalled that in order to beat the wall clock times of $\mathcal{P}$, the number of processors required to solve a particular problem using $\mathcal{P}_s$ must increase to $M(N-2)$. Thus this may be a constraining factor if the number of available compute cores is limited.

There are many different avenues for further work in testing and applying the stochastic parareal algorithm. One obvious area to explore is to identify and test the performance of $\mathcal{P}_s$ using an assortment of different sampling rules on different types of differential problems with various mathematical properties. Investigating the impact on convergence and accuracy of probability distributions with heavier tails or some form of skew would provide a clearer picture of how $\mathcal{P}_s$ can converge optimally. A sampling rule consisting of the original predictor-corrector rule plus some variable white noise would be valuable in determining a specific variance for which the algorithm converges optimally. Alternatively,

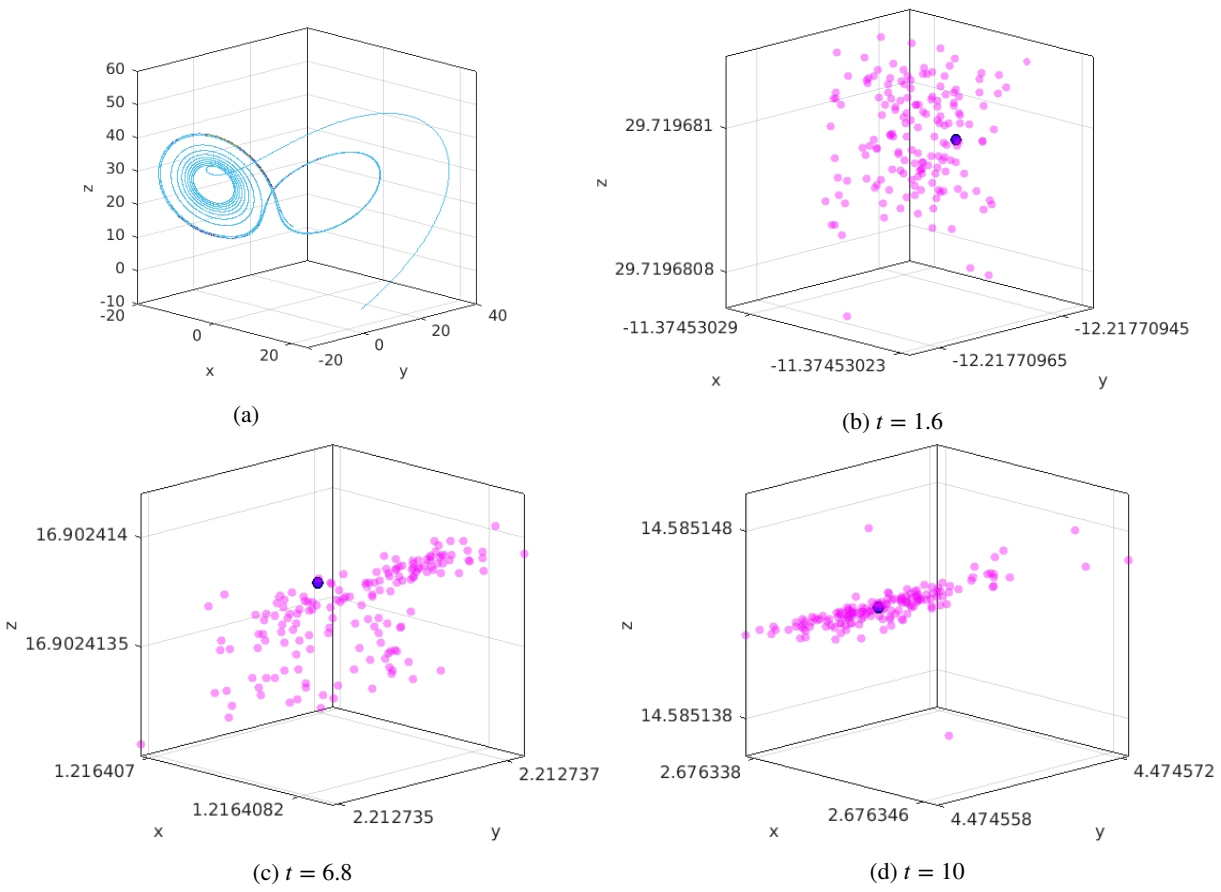


Figure 14: (a) All 250 stochastic trajectories of the Lorenz system in phase space generated by $\mathcal{P}_s$ (with rule 2) and the single trajectory generated by $\mathcal{P}$. Magnifications of these trajectories are plotted as scattered discrete points at times (b) $t = 1.6$, (c) $t = 6.8$ and (d) $t = 10$. Purple dots are from $\mathcal{P}_s$ whilst the single blue point is from $\mathcal{P}$.

one could investigate incorporating Bayesian methods into the sampling rule.

On the algorithmic level, assuming a sufficiently large computing memory, one could also look to store all previously calculated trajectories at each iteration k, thus having many more trajectories from which to construct a continuous solution from at each successive iteration, potentially offering faster convergence. Whilst $\mathcal{F}$ and $\mathcal{G}$ were both fixed as fourth-order Runge Kutta methods, there is also potential for other combinations of solvers to be used in $\mathcal{P}_s$ to exploit speed up. The final avenue to explore would be to apply $\mathcal{P}_s$ to much larger time-dependent evolution problems and determine whether the algorithm can still beat $\mathcal{P}$. The importance of developing and investigating time-parallel methods for high performance computing cannot be understated and remains an active and exciting area of research.

Bibliography

[1] L. Baffico, S. Bernard, Y. Maday, G. Turinici, and G. Zérah. Parallel-in-time molecular-dynamics simulations. *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, 66(5):4, Nov 2002.

[2] G. Bal and Y. Maday. A Parareal Time Discretization for Non-Linear PDE's with Application to the Pricing of an American Put. In *Recent Developments in Domain Decomposition Methods*, pages 189–202. Springer, Berlin, Heidelberg, 2002.

[3] G A. Staff and E M. Rønquist. Stability of the Parareal algorithm. *Lecture Notes in Computational Science and Engineering*, 40:449–456, 2005.

[4] D. Samaddar, D.P. Coster, X. Bonnin, L.A. Berry, and D.B. Elwasif, W.R.and Batchelor. Application of the parareal algorithm to simulations of ELMs in ITER plasma. *Computer Physics Communications*, 235:246–257, Feb 2019.

[5] D. Samaddar, D. E. Newman, and R. Sánchez. Parallelization in time of numerical simulations of fully-developed plasma turbulence using the parareal algorithm. *Journal of Computational Physics*, 229:6558–6573, Sep 2010.

[6] I. L. Markov. Limits on fundamental limits to computation, Aug 2014.

[7] K. Burrage. *Parallel and sequential methods for ordinary differential equations*. 1995.
[8] J. L. Gustafson. *Amdahl's Law*, pages 53–60. Springer US, Boston, MA, 2011.
[9] D. Morrison, J. D. Riley, and J. F. Zancanaro. Multiple Shooting Method for Two-Point Boundary Value Problems. *Communications of the ACM*, 5(12):613–614, 1962.
[10] J. Nievergelt. Parallel methods for integrating ordinary differential equations. *Communications of the ACM*, 7(12):731–733, Dec 1964.
[11] A. Bellen and M. Zennaro. Parallel algorithms for initial-value problems for difference and differential equations. Technical Report 3, 1989.
[12] P. Chartier and B. Philippe. A parallel shooting technique for solving dissipative ODE's. *Computing*, 51(3-4):209–236, Sep 1993.
[13] B. M.S. Khalaf and D. Hutchinson. Parallel algorithms for initial value problems: Parallel shooting. *Parallel Computing*, 18(6):661–673, Jun 1992.
[14] W. Hackbusch. Parabolic multi-grid methods. *Computing Methods in Applied Sciences and Engineering*, VI:189–197, 1984.
[15] G. Horton and S. Vandewalle. A Space-Time Multigrid Method for Parabolic Partial Differential Equations. *SIAM Journal on Scientific Computing*, 16(4):848–864, Jul 1995.
[16] C. Lubich and A. Ostermann. Multi-grid dynamic iteration for parabolic equations. *Bit*, 27(2):216–234, Jun 1987.
[17] M. J. Gander and A. M. Stuart. Influence of Overlap on the Convergence Rate of Waveform Relaxation. 1996.
[18] A. E. Ruehli and A. L. Sangiovanni-Vincentelli. The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1(3):131–145, 1982.
[19] C. W. Gear. The Numerical Integration of Ordinary Differential Equations. *Mathematics of Computation*, 21(98):146, 1967.
[20] M J. Gander. 50 Years of Time Parallel Time Integration. In Thomas Carraro, Michael Geiger, Stefan Körkel, and Rolf Rannacher, editors, *Multiple Shooting and Time Domain Decomposition Methods*, pages 69–113, Cham, 2015. Springer International Publishing.
[21] J. L. Lions, Y. Maday, and G. Turinici. Résolution d'EDP par un schéma en temps ńpararéel ż. *Comptes Rendus de l'Academie des Sciences - Series I: Mathematics*, 332(7):661–668, Apr 2001.
[22] M J. Gander and E. Hairer. In *Lecture Notes in Computational Science and Engineering*.
[23] M J. Gander and S. Vandewalle. On the Superlinear and Linear Convergence of the Parareal Algorithm. *Lecture Notes in Computational Science and Engineering*, 55:291–298, 2007.
[24] P F. Fischer, F. Hecht, and Y. Maday. A parareal in time semi-implicit approximation of the navier-stokes equations. *Lecture Notes in Computational Science and Engineering*, 40:433–440, 2005.
[25] J. M. F. Trindade and J. C. F. Pereira. Parallel-in-Time Simulation of Two-Dimensional, Unsteady, Incompressible Laminar Flows. *Numerical Heat Transfer, Part B: Fundamentals*, 50(1):25–40, Mar 2006.
[26] A. S. Nielsen. *Feasibility study of the parareal algorithm*. PhD thesis, 2012.
[27] E. Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.