Twelve Ways To Fool The Masses When Giving Parallel-In-Time Results

Sebastian Götschel[1], Michael Minion[2], Daniel Ruprecht[1], and Robert Speck[3]

[1] Hamburg University of Technology
Institute of Mathematics
Chair Computational Mathematics
Am Schwarzenberg-Campus 3
D-21073 Hamburg, Germany
{ruprecht, sebastian.goetschel}@tuhh.de
[2] Lawrence Berkeley National Laboratory
1 Cyclotron Road
Berkeley, CA 94720, USA
mlminion@lbl.gov
[3] Forschungszentrum Jülich GmbH
Jülich Supercomputing Centre
52425 Jülich, Germany
r.speck@fz-juelich.de

Abstract. Getting good speedup—let alone high parallel efficiency—for parallel-in-time (PinT) integration examples can be frustratingly difficult. The high complexity and large number of parameters in PinT methods can easily (and unintentionally) lead to numerical experiments that overestimate the algorithm's performance. In the tradition of Bailey's article "Twelve ways to fool the masses when giving performance results on parallel computers", we discuss and demonstrate pitfalls to avoid when evaluating performance of PinT methods. Despite being written in a light-hearted tone, this paper is intended to raise awareness that there are many ways to unintentionally fool yourself and others and that by avoiding these fallacies more meaningful PinT performance results can be obtained.

1 Introduction

The trend towards extreme parallelism in high-performance computing requires novel numerical algorithms to translate the raw computing power of hardware into application performance [5]. Methods for the approximation of time-dependent partial differential equations, which are used in models in a very wide range of disciplines from engineering to physics, biology or even sociology, pose a particular challenge in this respect. Parallelization of algorithms discretizing the spatial dimension via a form of domain decomposition is quite natural and has been an active research topic for decades. Exploiting parallelization in the time direction is less intuitive as time has a clear direction of information transport. Traditional algorithms for temporal integration employ a step-by-step procedure

that is difficult to parallelize. In many applications, this sequential treatment of temporal integration has become a bottleneck in massively parallel simulations.

Parallel-in-time (PinT) methods, i.e., methods that offer at least some degree of concurrency, are advertised as a possible solution to this temporal bottleneck. The concept was pioneered by Nievergelt in 1964 [15], but has only really gained traction in the last two decades [7]. By now, the effectiveness of PinT has been well established for examples ranging from the linear heat equation in one-dimension to more complex highly diffusive problems in more than one dimension. More importantly, there is now ample evidence that different PinT methods can deliver measureable reduction in solution times on real-life HPC systems for a wide variety of problems. Ong and Schroder [16] and Gander [7] provide overviews of the literature, and a good resource for further reading is also given by the community website `https://parallel-in-time.org/`.

PinT methods differ from space-parallel algorithms or parallel methods for operations like the FFT in that they do not simply parallelize a serial algorithm to reduce its run time.[1] Instead, serial time-stepping is usually replaced with a computationally more costly and typically iterative procedure that is amenable to parallelization. Such a procedure will run much slower in serial, but can overtake serial time-stepping in speed if sufficiently many processors are employed. This makes a fair assessment of performance much harder since there is no clear baseline to compare against. Together with the large number of parameters and inherent complexities in PinT methods and PDEs themselves, there are thus many sometimes subtle ways to fool oneself (and the masses) when assessing performance. We will demonstrate various ways to produce results that seem to demonstrate speedup but are essentially meaningless. The paper is written in a similar spirit as other "ways to fool the masses" papers first introduced in [3] who inspired a series of similarly helpful papers in related areas [10,11,14,9,18,4,17]. One departure from the canon here is that we provide actual examples to demonstrate the *Ways* as we present them. Despite the light-hearted, sometimes even sarcastic tone of the discussion, the numerical examples are similar to experiments one could do for evaluating the performance of PinT methods.

Some of the ways we present are specific to PinT while others, although formulated in "PinT language" correspond to broader concepts from parallel computing. This illustrates another important fact about PinT: while the algorithms often dig deeply into the applied mathematics toolkit, their relevance is exclusively due to the architectural specifics of modern high-performance computing systems. This inherent cross-disciplinarity is another complicating factor when trying to do fair performance assessments. Lastly we note that this paper was first presented in a shorter form as a conference talk at the 9th Parallel in Time Workshop held (virtually) in June, 2020. Hence some of the *Ways* are more relevant to live presentations, although all should be considered in both written and live scenarios. In the next section we present the 12 *Ways* with a series

---

[1] One exception are so-called "parallel-across-the-method" PinT methods in the terminology by Gear [8] that can deliver smaller-scale parallelism.

of numerical examples before concluding with some more serious comments in Section 3.

2   Fool the masses (and yourself)

2.1   Choose Your Problem Wisely!

If you really want to impress the masses with your PinT results, you will want to show as big a parallel speedup as possible, hence you will want to use a lot of processors in the time direction. If you are using, for example, Parareal [13], a theoretical model for speedup is given by the expression

$$S_{\text{theory}} = \frac{N_P}{N_P \alpha + K(1 + \alpha)}, \tag{1}$$

where N_P is the number of processors, α is the ratio of the cost of the coarse propagator $\mathcal{G}$ compared to the fine propagator $\mathcal{F}$, and K is the total number of iterations needed for convergence. Hence to get a large speedup that will impress the masses, we need to choose N_P to be large, α to be small, and hope K is small as well. A common choice for parareal is to have $\mathcal{G}$ be one step of some method and $\mathcal{F}$ be $N_{\mathcal{F}}$ steps of the same method so that $\alpha = 1/N_{\mathcal{F}}$ is small. But note that this already means that the total number of time steps corresponding to the serial method is now $N_P N_{\mathcal{F}}$. Hence we want to choose an equation and problem parameters for which very many time steps can be employed, while still showing good speedup without raising any suspicions that the problem is too "easy". The first example suggests some *Ways* to pull off this perilous balancing act.

In this example, we use the following nonlinear advection-diffusion-reaction equation

$$u_t = vu_x + \gamma u u_x + \nu u_{xx} + \beta u(a - u)(b - u),$$

where the constants v, γ, ν, β, a, and b determine the strength of each term. In order to squeeze in the massive number of time steps we need for good speedup, we choose a long time interval over which to integrate, $t \in [0, T_F]$, with $T_F = 30$. The initial condition is given on the interval $[0, 2\pi]$ by

$$u(x, 0) = 1 - d(1 - e^{-(x-\pi)^4/\sigma}).$$

If you are presenting this example in front of the an audience, try to get all the equations with parameters on one slide and then move on before defining them.

> **Way 1.** *Choose a seemingly complicated problem with lots of parameters which you define later (or not at all).*

For the first numerical test, we choose $N_P = 200$ processors, and use a fourth-order IMEX Runge-Kutta method and a pseudo-spectral discretization in space

using 128 grid points, where the linear advection and diffusion terms are treated implicitly. We use one time step for $\mathcal{G}$ and $N_\mathcal{F} = 64$ steps for $\mathcal{F}$. Since the method is spectrally accurate in space, it gives us cover to use a lot of time steps (more on that later). We set the stopping criterion for Parareal to be when the increment in the iteration is below 10^{-9}, and $v = -0.5$, $\gamma = 0.25$, $\nu = 0.01$, $\beta = -5$, $a = 1$, $b = 0$, and $d = 0.55$ (see also Appendix 1). For these values, Parareal converges on the entire time interval in 3 iterations. The theoretical speedup given by Eq. 1 is 32.4. Not bad!

If we explore no further, we might have fooled the masses. How did we manage? Consider a plot of the initial condition and solution at the final time for this problem shown in Fig. 1, with the blue and orange lines respectively. The lesson here is

Way 2. *Quietly use an initial condition and/or problem parameters for which the solution tends to a steady state. But do not show the actual solution.*
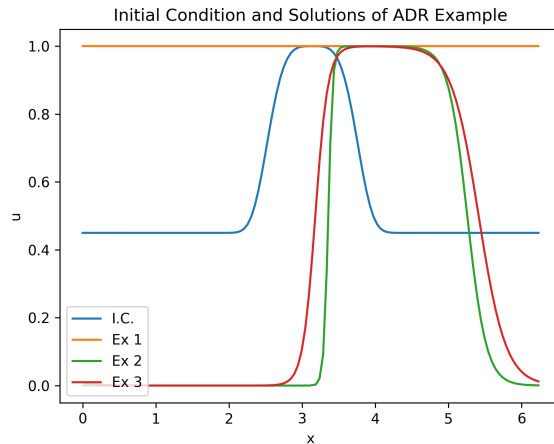

Fig. 1: Initial solution and solution at $t = 30$ for the advection-diffusion-reaction problem demonstrating the significant effect that parameter selection can have on the dynamics and subsequent PinT speedup discussed in *Ways* 1-4.

If we repeat this experiment changing only one parameter, $b = 0.5$, the number of Parareal iterations needed for convergence jumps to $K = 10$ for a less impressive theoretical speedup of 15.05. In this case the solution quickly evolves not to constant state, but a steady bump moving at constant speed (the green line in Fig. 1). This raises another important point to fool the masses:

Way 3. *Do not show the sensitivity of your results to problem parameter changes. Find the best one and let the audience think the behavior is generic.*

Sometimes you might be faced with a situation like the second case above and not know how to get a better speedup. One suggestion is to add more diffusion. Using the same parameters except increasing the diffusive coefficient to $\nu = 0.04$ reduces the number of iterations for convergence to $K = 5$ with a theoretical speedup of 24.38. The solution of this third example is shown by the red line Fig. 1. If you can't add diffusion directly, using a diffusive discretization for advection like first-order upwind finite differences can sometimes do the trick while avoiding the need to explicitly admit to the audience that you needed to increase the amount of "diffusion".

Way 4. *If you are not completely thrilled about the speedup because the number of iterations K is too high, try adding more diffusion. You might have to experiment a little to find just the right amount.*

2.2 Over-resolve the solution! Then over-resolve some more.

After carefully choosing your test problem, there are ample additional opportunities to boost the parallel performance of your numerical results. The next set of *Ways* consider the effect of spatial and temporal resolution. We consider the 1D nonlinear Schrödinger equation

$$u_t = \mathrm{i}\Delta u + 2\mathrm{i}\,|u|^2\,u \tag{2}$$

with periodic boundary conditions in $[0, 2\pi]$ and the exact solution as given by Aktosun et al. [1], which we also use for the initial condition at time $t_0 = 0$.

This is a notoriously hard problem for out-of-the-box PinT methods, but we are optimistic and give Parareal a try. We use a second-order IMEX Runge-Kutta method by Ascher et al. [2] with $N_{\mathcal{F}} = 1\,024$ steps and $N_{\mathcal{G}} = 32$ coarse steps for each of the 32 processors. In space, we again use a pseudo-spectral method with the linear part treated implicitly and $N_x = 32$ degrees-of-freedom. The estimated speedup can be found in Fig. 2a. Using $K = 5$ iterations, we obtain a solution about 6.24 times faster when running with 32 instead of 1 processor. All runs achieve the same accuracy of 5.8×10^{-5} and it looks like speedup in time can be easily achieved after all.

Yet, although the accuracy compared to the exact solution is the same for all runs, the temporal resolution is way too high for this problem, masking the effect of coarsening in time. The spatial error is dominating and instead of $32 \times 1\,024 = 32\,768$ overall time steps, only $32 \times 32 = 1\,024$ are actually needed to balance spatial and temporal error. Therefore, the coarse level already solves the problem quite well: speedup only comes from over-resolving in time.

If we instead choose $N_{\mathcal{F}} = 32$ time steps on the fine level instead with the same coarsening factor of 32 ($N_{\mathcal{G}} = 1$), we get no speedup at all – see the

red curve/diamond markers in Fig. 2b. Using a less drastic coarsening factor of 4 leads to a maximum speedup of 1.78 with 32 processors (blue curve/square markers), which is underwhelming and frustrating and not what we would prefer to present in public. Lesson learned:

> **Way 5.** *Make Δt so small that the coarse integrator is already accurate. Never check if a larger Δt might give you the same solution.*

The astute readers may have noticed we also used this trick to a lesser extent in the advection-diffusion-reaction example above.
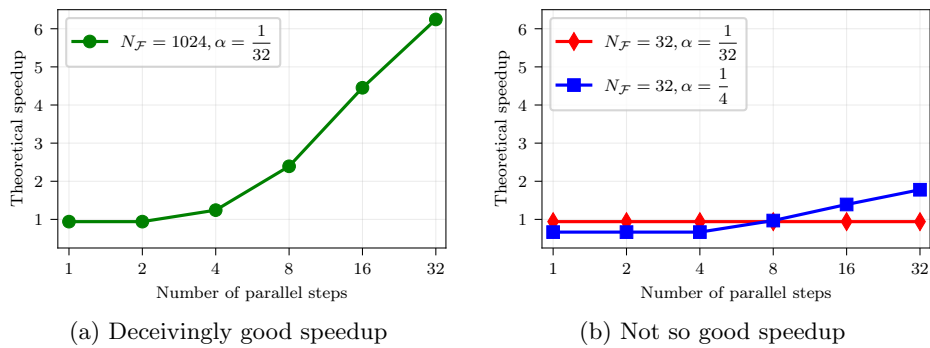

(a) Deceivingly good speedup   (b) Not so good speedup

Fig. 2: Estimated speedup for Parareal runs of the nonlinear Schrödinger example (2) demonstrating the effect of over-resolution in time (*Way* 5).

A similar effect can be achieved when considering coarsening in space. Since we have learned that more parameters are always good to fool the masses we now use PFASST [6] instead of Parareal. We choose 5 Gauss-Lobatto quadrature nodes per time step, leading to an 8th order IMEX method, which requires only 8 time steps to achieve an accuracy of about 5.8×10^{-5}. We do not coarsen in time, but – impressing everybody with how resilient PFASST is to spatial coarsening – go from 512 degrees-of-freedom on the fine to 32 on the coarse level. We are rewarded with the impressive speedup shown in Fig. 3a: using 8 processors, we are 5.7 times faster than the sequential SDC run. To really drive home the point how amazing this is, we point out to the reader that this corresponds to 71% parallel efficiency. Even the space-parallel linear solver crowd would have to grudgingly accept such an efficiency as respectable.

However, since the spatial method did not change from the example before, we already know that 32 degrees-of-freedom would have been sufficient to achieve a PDE error of about 10^{-5}. So using 512 degrees-of-freedom on the fine level heavily over-resolves the problem in space. Using only the required 32 degrees-of-freedom on the fine level with a similar coarsening factor of 4 only gives a speedup of 2.7, see Fig. 2b (red curve/diamond markers). While we could probably sneak

this into an article, the parallel efficiency of 34% will hardly impress anybody outside of the PinT community.

It is worth noting that better resolution in space on the coarse level does not help (blue curve/square markers). This is because the coarse level does not contribute anything to the convergence of the method anymore. Turning it off completely would even increase the theoretical speedup to about 3.5. Hence, for maximum effect:

Way 6. *When coarsening in space, make Δx on the fine level so small that even after coarsening, the coarse integrator is accurate. Avoid the temptation to explore a more reasonable resolution.*
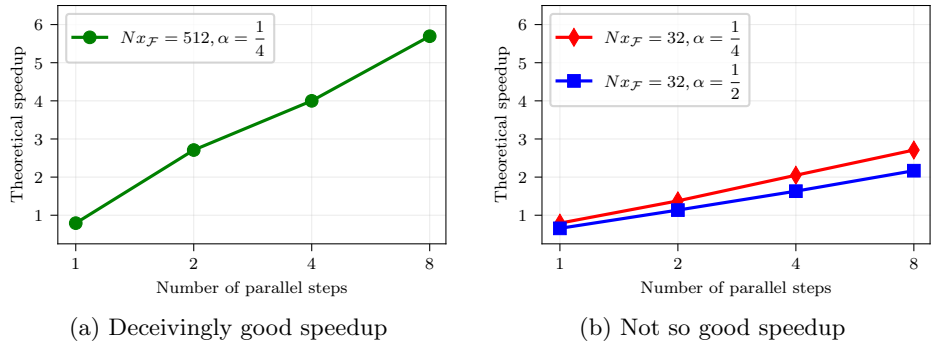

(a) Deceivingly good speedup   (b) Not so good speedup

Fig. 3: Estimated speedup for PFASST runs of the nonlinear Schrödinger example (2) demonstrating the effect of over-resolution in space (*Way* 6).

2.3   Be smart when setting your iteration tolerance!

If the audience catches on about your $\Delta t/\Delta x$ over-resolution issues, there is a more subtle way to over-resolve and fool the masses. Since methods like Parareal and PFASST are iterative methods, one must decide when to stop iterating - use this to your advantage! The standard approach is to check the increment between two iterations or some sort of residual (if you can, use the latter: it sounds fancier and people will ask fewer questions). In the runs shown above, Parareal is stopped when the difference between two iterates is below 10^{-10} and PFASST is stopped when the residual of the local collocation problems is below 10^{-10}.

These are good choices, as they give you good speedup: for the PFASST example, a threshold of 10^{-5} would have been sufficient to reach the accuracy of the serial method. While this leads to fewer PFASST iterations (good!), unfortunately it also makes the serial SDC baseline much faster (bad!). Therefore,

with the higher tolerance, speedup looks much less attractive, even in the over-resolved case, see Fig. 4a.

Similarly, when using more reasonable tolerances, the speedup of the well-resolved examples decreases as shown in Fig. 4b. This leads to our next *Way*, which has a long and proud tradition, and for which we can therefore quote Pakin [17] directly,

Way 7. *"Hence, to demonstrate good [...] performance, always run far more iterations than are typical, necessary, practical, or even meaningful for real-world usage, numerics be damned!"*

(a) Now also not so good speedup   (b) Still not so good speedup
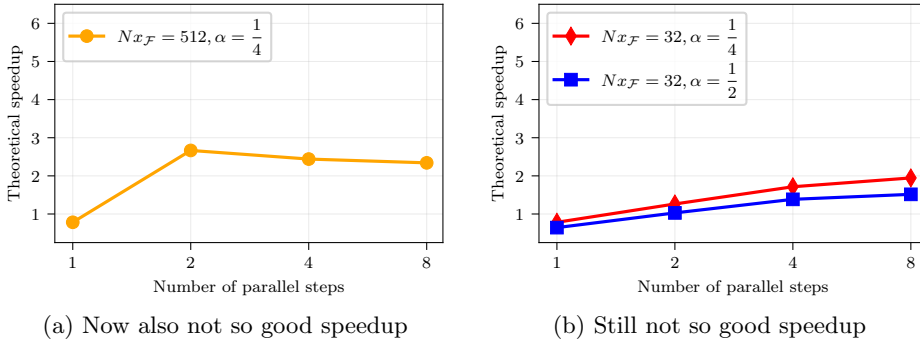
Fig. 4: Estimated speedup for PFASST runs of the nonlinear Schrödinger example (2) with different resolutions in space demonstrating how using a sensible iteration tolerance of 10^{-5} can reduce speedup (*Way* 7).

Yet another smart way to over-resolve is to choose a ridiculously small tolerance for an inner iterative solver. Using something cool like GMRES to solve the linear systems for an implicit integrator far too accurately is an excellent avenue for making your serial baseline method much slower than it needs to be. This is further desirable because it reduces the impact of tedious overheads like communication or waiting times. We all know that an exceptional way to good parallel performance is using a really slow serial baseline to compare against.

Way 8. *Not only use too many outer iterations, but try to maximize the amount of work done by iterative spatial solvers (if you have one, and you always should).*

Note that for all the examples presented so far, we did not report any actual speedups measured on parallel computers. Parallel programming is tedious and difficult, as everybody understands, and what do we have performance models for, anyway? It is easier to just plug your parameters into a theoretical model. Realizing this on an actual system can be rightfully considered Somebody Else's

Problem (SEP) or a task for your dear future self. But for completeness, the next example will address this directly.

2.4   Don't report runtimes!

Because solving PDEs only once can bore an audience, we will now talk about optimal control of the heat equation, the "hello world" example in optimization with time-dependent PDEs. This problems has the additional advantage that even more parameters are available to tune. Our specific problem is as follows. Given some desired state u_d on a space-time domain $\Omega \times (0, T)$, $\Omega \subset \mathbb{R}^d$, we aim to find a control c to minimize the objective functional

$$J(u, c) = \frac{1}{2} \int_0^T \|u - u_d\|_{L^2(\Omega)}^2 \, \mathrm{d}t + \frac{\lambda}{2} \int_0^T \|c\|_{L^2(\Omega)}^2 \, \mathrm{d}t$$

subject to

$$u_t - \nabla^2 u = c + f(u)$$

with periodic boundary conditions (allowing us to use FFT to evaluate the Laplacian and perform the implicit linear solves (*Way* 8 be damned). For the linear heat equation considered in the following, the source term is $f(u) \equiv 0$ (see *Way* 1). Optimization is performed using steepest descent; for computation of the required reduced gradient we need to solve a forward-backward system of equations for state u and adjoint p,

$$u_t - \nabla^2 u = c + f(u) \qquad -p_t - \nabla^2 p - f'(u)p = u - u_d$$
$$u(\cdot, 0) = 0 \qquad\qquad\qquad p(\cdot, T) = 0.$$

To parallelize in time, we use, for illustration, the most simple approach: given a control c, the state equation is solved parallel-in-time for u, followed by solving the adjoint equation parallel-in-time for p with PFASST using $N_P = 20$ processors. For discretization, we use 20 time steps and three levels with 2/3/5 Lobatto IIIA nodes in time as well as 16/32/64 degrees of freedom in space. As a sequential reference we use MLSDC on the same discretization. We let PFASST/MLSDC iterate until the residual is below 10^{-4} instead of iterating to high precision, so we can openly boast how we avoid *Way* 7.

In the numerical experiments we perform one iteration of an iterative, gradient-based optimization method to evaluate the method (i.e., solve state, adjoint, evaluate objective, compute gradient). As initial guess for the control we do not use the usual choice $c_0 \equiv 0$ as this would lead to $u \equiv 0$ but a nonzero initial guess—again, we make sure everybody knows that we avoid *Way* 2 in doing so. To estimate speedup we count PFASST/MLSDC iterations and compute

$$S = \frac{\text{total MLSDC iterations state} + \text{adjoint}}{\text{total iterations state on } CPU_{20} + \text{total iterations adjoint on } CPU_1}.$$

We get $S = \frac{40+60}{7+7} = 7.1$, for a nice parallel efficiency of 35.5%.

Before we publish this, we might consider actual timings from a real computer. Unfortunately, using wall clock times instead of iterations gives

$$S = \frac{\text{serial wall clock time}}{\text{parallel wall clock time}} = \frac{44.3s}{18.3s} = 2.4,$$

and thus only roughly a third of the theoretical speedup. To avoid this embarrassment:

Way 9. *Only show theoretical/projected speedup numbers (but mention this only in passing or not at all). If you include the cost of communication in the theoretical model, assume it is small enough not to affect your speedup.*

Why is the theoretical model poor here? One cause is the overhead for the optimization—after all, there is the evaluation of the objective functional, and the construction of gradient. Ignoring parts of your code to report better results is another proud tradition of parallel computing, see Bailey's paper [3]. However, most of the tasks listed do trivially parallelize in time. The real problem is that communication on an actual HPC system is aggravatingly not really instantaneous.
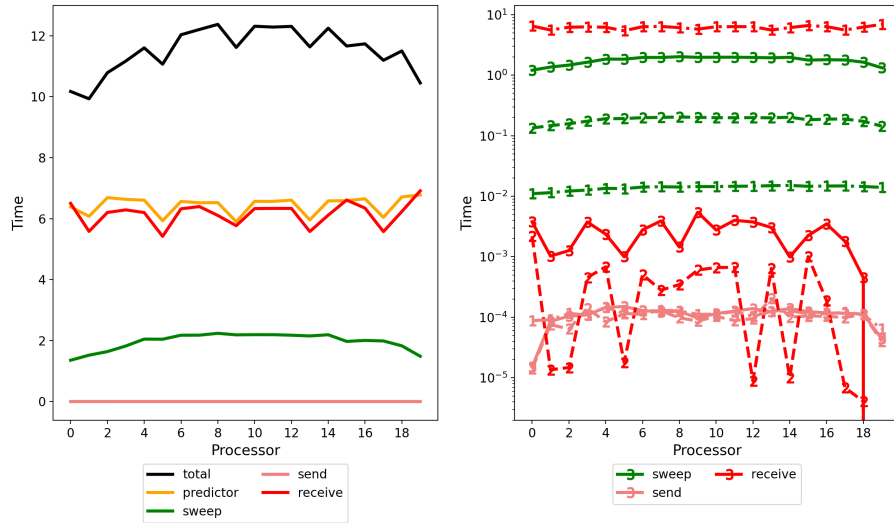

Fig. 5: Wall clock times of the different algorithmic steps for the linear heat equation example on $\Omega = [0, 1]^3$ and $T = 1$. Left: total times. Right: times per level (1 is coarsest level, 3 finest). Note the "receive" times are not negligible as discussed in *Way* 9.

Looking at detailed timings for PFASST, Fig. 5 shows that the issue truly is in communication costs, which clearly cannot be neglected. In fact, more time is spent on blocking coarse grid communication than on fine sweeps. Note also that, due to the coupled forward-backward solves, each processor requires similar computation and communication times. The following performance model

$$S = \frac{N_P}{\frac{N_P \alpha}{K_S} + \frac{K_P}{K_S}(1 + \alpha + \beta)}$$

accounts for overheads in the β term. Matching the measured speedups requires setting $\beta = 3$ or three times the cost of one sweep on the fine level! This is neither small nor negligible by any measure.

2.5 Choose the measure of speedup to your advantage!

Technically, parallel speedup should be defined as the ratio of the run time of your parallel code to the run time required by the **best** available serial method. But who has the time or energy for such a careful comparison? Instead, it is convenient to choose a baseline to get as much speedup as possible.

In the example above, MLSDC was used as a baseline since it is essentially the sequential version of PFASST and allows for a straightforward comparison and the use of a theoretical speedup. However, MLSDC might not be the fastest serial method to solve state and adjoint equations to some prescribed tolerance. For illustration, we consider solving an optimal control problem for a nonlinear heat equation with $f(u) = -\frac{1}{3}u^3 + u$ on $\Omega \times (0,T) = [0,20] \times (0,5)$. Wall clock times were measured for IMEX-Euler, a 4th-order additive Runge-Kutta scheme (ARK-4), and 3-level IMEX-MLSDC with 3/5/9 Lobatto IIIA collocation points, with each method reaching a similar final accuracy in the computed control (thus, using different number of time steps, but the same spatial discretization). For the IMEX methods, the nonlinearity as well as the control were treated explicitly. IMEX-Euler was fastest with $102.5s$, clearly beating MLSDC ($169.8s$) despite using significantly more time steps. The ARK-4 method here required $183.0s$, as the non-symmetric stage values slow down the forward-backward solves due to the required dense output. With PFASST on 32 CPUs requiring $32s$, the speedup reduces from 5.3 for MLSDC as a reference to 3.2 when compared to IMEX-Euler. By choosing the sequential baseline method wisely, we can increase the reported speedup in this example by more than 65%!

A very similar slight of hand is hidden in Sect. 2.2, where only theoretical speedups are reported. In the PFASST examples, the SDC iteration counts are used as the baseline results, although in most cases MLSDC required up to 50% fewer iterations to converge. Using MLSDC as a baseline here would reduce the theoretical speedups significantly in all cases. Whether this still holds when actual run times are considered, is, as we have just seen, part of a different story.

Way 10. *If you report speedup based on actual timings, compare your code to the method run on one processor and never against a different and possibly more efficient serial method.*

2.6   Use low order serial methods!

A low-order temporal method is a choice convenient for PinT methods because they are easier to implement and allow one to take many time steps without falling prey to *Way* 5, especially when you want to show how the speedup increases as you take ever more time steps for a problem on a fixed time interval. After all, it is the parallel scaling that is exhilirating, not necessarily how quickly one can compute a solution to a given accuracy.

For this example we will again use Parareal applied to the Kuramoto-Sivashinsky Equation. The K-S equation is a good choice to impress an audience because it gives rise to chaotic temporal dynamics (avoiding *Way* 1). The equation in one dimension reads

$$u_t = -uu_x - u_{xx} - u_{xxxx},$$

which we solve on the spatial interval $x \in [0, 32\pi]$ and temporal interval $t \in [0, 40]$. Since the fourth-derivative term is linear and stiff, we choose a first order exponential integrator in a spectral-space discretization where the linear operators diagonalize and hence the exponential of the operator is trivial to compute. We use 512 points in space, and in this study will compare a serial first-order method with Parareal using the same first-order method in terms of cost per accuracy. Using 32 time processors for all runs, we increase the number of steps for the fine Parareal propagator $\mathcal{F}$ and hence the total number of time steps. The theoretical speedups (ignoring *Way* 9) are displayed on the left panel of Fig. 6. One can see that the Parareal method provides speedup at all temporal resolutions up to a maximum of about 5.85 at the finest resolution (where α is the smallest). So we have achieved meaningful speedup with a respectable efficiency for a problem with complex dynamics. Best to stop the presentation here.

If we are a little more ambitious, we might replace our first-order integrator with the 4th-order exponential Runge-Kutta (ERK) method from [12]. Now we need to be more careful about *Way* 5 and hence won't be able to take nearly as many time steps. In the right panel of Fig. 6 we show the 1st-order and 4th-order results together. The maximum theoretical speedup attained with the 4th-order method is only about 3.89 at the finest resolution, which is probably reason enough not to do the comparison. But there is the additional irritation that at any accuracy level, the serial 4th-order method is significantly faster than the Parareal 1st-order method.

> **Way 11.** *It is best to show speedup for first-order time integrators since they are a bit easier to inflate. If you want to show speedup for higher-order methods as well, make it impossible to compare cost versus accuracy between first-order and higher-order methods.*

3   Parting thoughts

The careful reader may have noticed that in all the examples above, a single PinT method is used for each *Way.* This brings us finally to
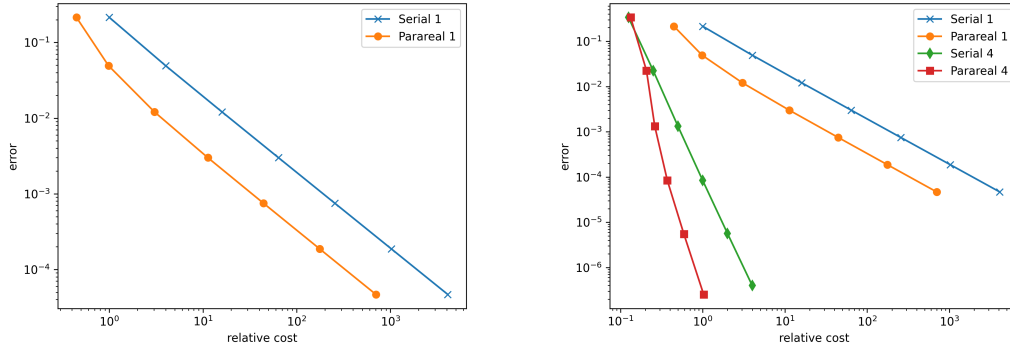
Fig. 6: Comparison of serial and Parareal execution time for the K-S example using a first- and fourth-order ERK integrators. Note that the serial fourth-order integrator is always faster for a given accuracy than the parallel first-order method (*Way* 11)

Way 12. *Never compare your own PinT method to a different PinT method.*

The problem, as we have seen, is that assessing performance for a single PinT method is already not straightforward. Comparing the performance of two or more different methods makes matters even more difficult. Although it has been often discussed within the PinT community, efforts to establish a set of benchmark test examples have, to date, made little head way. The performance of methods like PFASST and Parareal considered here are highly sensitive to the type of equation being solved, the type of spatial discretization being used, the accuracy desired, and the choice of problem and method parameters. In this study we purposely choose examples that lead to inflated reported speedups, and doing this required us to use our understanding of the methods and the equations chosen. Conversely, in most instances, a simple change in the experiment leads to much worse reported speedups. Different PinT approaches have strengths and weaknesses for different benchmark scenarios, hence establishing a set of benchmarks that the community would find *fair* is a very non-trivial problem.

Roughly, the ways we present can be grouped into three categories: "choose your problem" (ways 1–4), "over-resolve" (ways 5–8) and "choose your performance measure" (ways 9–11). This classification is not perfect as some of the *Ways* overlap. Some of the dubious tricks presented here are intentionally obvious to detect, while others are more subtle. As in the original "twelve ways" article, and those it inspired, the examples are meant to be light-hearted. However, many of the *Ways* have been (unintentionally) used when reporting numerical results, and the authors are not without guilt in this respect. Admitting that, we hope this article will be read the way we intended: as a demonstration of some

of the many pitfalls one faces when assessing PinT performance and a reminder that considerable care is required to obtain truly meaningful results.

Acknowledgements

References

1. Aktosun, T., Demontis, F., Van Der Mee, C.: Exact solutions to the focusing nonlinear Schrödinger equation. Inverse Problems **23**(5), 2171 (2007)
2. Ascher, U.M., Ruuth, S.J., Spiteri, R.J.: Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. Appl. Numer. Math. **25**(2), 151–167 (1997)
3. Bailey, D.H.: Twelve ways to fool the masses when giving performance results on parallel computers. Supercomputing Review **4**(8) (1991)
4. Chawner, J.: Revisiting "twelve ways to fool the masses when describing mesh generation performance". `https://blog.pointwise.com/2011/05/23/revisiting-%e2%80%9ctwelve-ways-to-fool-the-masses-when-describing-mesh-generation-performance%e2%80%9d/` (2011). Accessed: 2020-4-28
5. Dongarra, J., et al.: Applied Mathematics Research for Exascale Computing. Tech. Rep. LLNL-TR-651000, Lawrence Livermore National Laboratory (2014). URL `http://science.energy.gov/%7E/media/ascr/pdf/research/am/docs/EMWGreport.pdf`
6. 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). DOI 10.2140/camcos.2012.7.105. URL `http://dx.doi.org/10.2140/camcos.2012.7.105`
7. Gander, M.J.: 50 years of Time Parallel Time Integration. In: Multiple Shooting and Time Domain Decomposition. Springer (2015). DOI 10.1007/978-3-319-23321-5_3. URL `http://dx.doi.org/10.1007/978-3-319-23321-5_3`
8. Gear, C.W.: Parallel methods for ordinary differential equations. CALCOLO **25**(1-2), 1–20 (1988). DOI 10.1007/BF02575744. URL `http://dx.doi.org/10.1007/BF02575744`
9. Globus, A., Raible, E.: Fourteen ways to say nothing with scientific visualization. Computer **27**(7), 86–88 (1994)
10. Gustafson, J.L.: Twelve ways to fool the masses when giving performance results on traditional vector computers. `http://www.johngustafson.net/fun/fool.html` (1991). Accessed: 2020-4-28
11. Hoefler, T.: Twelve ways to fool the masses when reporting performance of deep learning workloads! (not to be taken too seriously) (2018). ArXiv, 1802.09941

12. Krogstad, S.: Generalized integrating factor methods for stiff PDEs. J. Comput. Phys. **203**(1), 72–88 (2005)
13. 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). DOI 10.1016/S0764-4442(00)01793-6. URL `http://dx.doi.org/10.1016/S0764-4442(00)01793-6`
14. Minhas, F., Asif, A., Ben-Hur, A.: Ten ways to fool the masses with machine learning (2019). ArXiv, 1901.01686
15. Nievergelt, J.: Parallel methods for integrating ordinary differential equations. Commun. ACM **7**(12), 731–733 (1964). DOI 10.1145/355588.365137. URL `http://dx.doi.org/10.1145/355588.365137`
16. Ong, B.W., Schroder, J.B.: Applications of time parallelization. Computing and Visualization in Science **648** (2019)
17. Pakin, S.: Ten ways to fool the masses when giving performance results on GPUs. HPCwire, December **13** (2011)
18. Tautges, T.J., White, D.R., Leland, R.W.: Twelve ways to fool the masses when describing mesh generation performance. IMR/PINRO Joint Rep. Ser. pp. 181–190 (2004)

Appendix 1

The value of σ in the first example is 0.02.