ParaDiag and Collocation Methods: Theory and Implementation

Zur Erlangung des akademischen Grades eines

DOKTORS DER NATURWISSENSCHAFTEN

von der KIT-Fakultät für Mathematik des
Karlsruher Instituts für Technologie (KIT)
genehmigte

DISSERTATION

von

Gayatri Čaklović

Tag der mündlichen Prüfung: 28. Juni 2023

1. Referent: Prof. Dr. Martin Frank
2. Referent: Prof. Dr. Willy Dörfler
3. Referent: Prof. Dr. Martin J. Gander

Betreuer: Dr. Robert Speck

Dedicated to
Sanja and Saša Singer

Acknowledgements

First and foremost, I would like to thank my supervisor Martin Frank for the opportunity to pursue my projects at KIT and for constructive feedback that has greatly improved the quality of this work.

Furthermore, I would also like to thank my second supervisor, Robert Speck, who introduced me to time-parallel integration while starting my project in Forschungszentrum[1] Jülich. I appreciate all the creative and mathy brainstorming conversations we had and the immense support I received throughout my Ph.D. During my time at FZJ, I shared a lot of fun times with my colleagues Giorgio, Ruth[2], Alba and Lukas. I will never forget our coffee breaks, adventures, and lake trips when the summers got too hot for the little container building we sat in. I would also like to thank Hannah for her great support and help.

I want to express my sincere gratitude to all who have helped me to complete this dissertation, especially Sebastian and Thibaut from TUHH whose comments and suggestions greatly improved the content of this work. I would also like to thank them for the fun times during conferences as well as for supporting my research. A special thanks to Thibaut for coming up with a lot of mini-projects that shifted my focus to new exciting topics.

Working at KIT allowed me to meet many new colleagues, and I would like to extend my gratitude to Steffen and an occasional special guest Antonia, for all the brainstorming sessions and shared activities which made my office days interesting and fun. I would also like to thank Chinmay and Pia for making my office days more lively.

I also want to thank my friends Anja, Hana, Lena, and Petra for bearing with me and supporting me through all the challenges a Ph.D. holds. I am also grateful for the support of my friends from Zagreb and Karlsruhe, especially Kirsten and Olivia.

Lastly, I am deeply grateful for the love and support of my family Vanda, Lavoslav, and Dea. Nick provided constant support and encouragement, especially in these last stages of writing this dissertation, and reminded me there is more to life.

[1] I think I still can't spell this correctly.

[2] Special thanks to Ruth for sneaking out the couch.

Preface

In a world in which the price of calculation continues to decrease rapidly, but the price of theorem proving continues to hold steady or increase, elementary economics indicates that we ought to spend a larger and larger fraction of our time on calculation.

JOHN TUKEY,
American Statistician 40 (1986).

The numerical computation of time-dependent differential equations has been studied for a long time. Euler was one of the famous mathematicians working on this topic whose method we use until this day. Since Euler's era, significant advancements have been made in theory and technology, allowing the field of numerical analysis to advance further. In 1976, an important milestone in recent history was the development of the first CRAY-1 supercomputer, ushering in a new era of high-performance computing. Since then, the progress in supercomputer development has been relentless, with newer and more powerful systems being developed continuously. With these high-performance machines, new branches of mathematics are formed, designing algorithms tailored for exascale computing.

Time-parallel integration falls under this category. It is a technique for solving time-dependent differential equations that tries to break an inherently sequential structure of the classical time-stepping methods, such as Euler's method, by redesigning the mathematics behind it. Even before the assembly of the first supercomputer, Nievergelt explored the idea of time-parallel integration already in 1964 [1]. The introduction of IBM's multicore `Power 4` chip in 2001 marked another significant milestone in computing, and it happened that traditional spatial parallelization methods started to approach their scaling limits. As a result, the field of time-parallel integration gained even more attention, as it offers a promising approach to overcome the limitations of spatial parallelization and achieve even more significant computational speedup.

Excluding Nievergelt's idea of a shooting method, the first ideas of time-parallel integration rely on coarsening in time. While the coarsening aspect does not usually impact parabolic-type equations, it can severely deteriorate the convergence when applied to hyperbolic-type equations. ParaDiag is an iterative time-parallel integrator that does not rely on coarsening, making it a promising parallel-in-time method for hyperbolic equations. Put simply, the method can be seen as a recipe for coupling time steps of existing integrators and performing a parallel fast Fourier transform in time. In this stage, the time steps become decoupled and are solved in parallel. To complete the iteration, an inverse Fast Fourier transform in time is performed, bringing the iterate closer to the numerical solution obtainable by a classical time-stepping approach.

Despite the elementary economics from Tuckey's quote, we first turn to the theoretical aspects of ParaDiag. First, we fill some missing gaps in error analysis for ParaDiag and extend its skeleton to collocation methods, a subclass of implicit Runge-Kutta methods. Collocation methods cover a class of high-order integrators allowing better approximations of solutions using fewer nodes per time step. With this, we analyze yet another level of time parallelism inside the decoupled problems: across the collocation nodes. Second, we develop a theoretical framework based on implicit-explicit splitting that enables us to analyze the proposed approach for solving nonlinear evolutionary differential equations. Lastly, we provide an open-source parallel implementation and benchmarks to support our theoretical claims. Furthermore, we show that speedup for hyperbolic equations is possible, and it is possible even after the spatial parallelism has been saturated.

The outline of the thesis is as follows. Chapter 1 contains an overview of time-parallel integration and summarizes the contributions of this work. Chapter 2 defines collocation methods and ParaDiag for linear equations. A round-off error analysis is developed to examine its impacts on convergence. Moreover, Chapter 3 extends the method to nonlinear equations through the prism of implicit-explicit iterations and studies its convergence. A parallel MPI implementation is described in Chapter 4. The method's capabilities are tested in Chapter 5, where we also stress the importance of relevant test cases and how they affect the results. Chapter 6 provides an overall conclusion and outlook.

Novelty and credit statement

This thesis contains a series of new research studies conducted with various co-authors and placed in the context of prior research. In this section, I will specify which parts of the thesis are new research and emphasize the contributions made by both myself and my co-authors.

Chapter 2 combines ParaDiag with the collocation problem for linear equations, and it was a collaborative effort between myself and my supervisors Robert Speck and Martin Frank [2]. Robert Speck provided general ideas and direction and helped me mathematically formulate the process of parameter selection in Subsection 2.5.3. In general, my co-authors provided valuable guidance and input while I worked out most the theory and implementation. Section 5.3 presents the outcomes obtained in [2], but it also includes additional explanations, expanding the section's scope.

Chapter 3 contains unpublished results. However, the idea of treating the nonlinearities explicitly came from Robert Speck. However, the theoretical framework was provided by me. Shu Lin Wu pointed out the idea of using iterative refinement to carry out the fixed-point iterations.

Subsection 5.4.2 presenting the results for the Boltzmann equation was developed with the help of Tianbai Xiao, who helped me couple my implementation with the `Julia` based package he developed [3].

Appendix B is part of [4], where I am the sole author.

The remainder of the thesis is original work that has not been published previously. After introducing time-parallel integration, the scientific contributions that build on existing ideas are presented in Subsection 1.2.3.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction to time-parallel integration

High-performance computing (HPC) uses advanced computer systems, such as supercomputers, to solve various problems on multiple processing units. The primary usage of parallel computing is often producing results faster by adapting a given algorithm. For some problems, this is straightforward. One of these examples is the Monte Carlo method, an *embarrassingly parallelizable* algorithm. Time-parallel integration, on the other hand, is on the opposite side of the spectrum. Coupling HPC and computation of parallel-in-space solutions of differential equations has become standard practice, however, the number of processors in supercomputers keeps increasing, and scalability limits are quickly getting maxed out. In cases like these, time-parallel integration methods may offer additional scaling opportunities.

Let

$$u_t = f(t, u(t)), \quad u(0) = u_0$$

be an evolutionary differential equation. Classical numerical methods that solve such equations rely on a time propagator $\mathcal{F}$ that propagates the initial condition forward in time for some time step ΔT. Obtaining an approximation of the solution in point $L\Delta T$ requires L propagations, where each of these approximations depends on the previous one;

$$u_{\ell+1} = \mathcal{F}(u_\ell).$$

Time-parallel integration tries to break this inherently sequential nature of the problem. Throughout the years, people came up with many ideas, and in 2015, a broad overview of the field Parallel-in-Time (PinT) has been written: *50 Years of time-parallel integration* [6]. Another review article from 2019 summarizing the many advances in time-parallel computations can be found in [7]. Today, we mark almost 60 years with multiple usages of PinT methods, ranging from weather prediction [8], robotics [9], computational fluid dynamics [10], and many more [11, 12, 13]. According to the Parallel-in-Time web page[1], more than 500 publications have been published so far, with the number exponentially growing from as early as 1964. The field properly took off in 2001 with the introduction of the *parareal*

[1] http://parallel-in-time.org/references/

algorithm [14], a simple yet powerful idea of using two types of integrators, a coarse and a fine numerical propagator, with the hope of achieving speedup. The same year, IBM introduced the world's first multicore processor chip, allowing higher performance at lower energy, ideal for large scale applications.

A classical way to distinguish parallel-in-time methods, following K. Burrage's terminology, is: *parallelism across the system*, *parallelism across the method* and *parallelism across time steps* [15]. Solving an ordinary differential equation where the right-hand side is a vector-valued function can be performed via Picard iterations, in parallel for every component, and is an example of parallelism across the system. Solving a diagonal IRK method in parallel across the stages is parallelism across the method, whereas the parareal algorithm, introduced in 1.1.1, is an example of parallelism across time steps. Some methods use coarsening in time or space or both. A rule of thumb is that coarsening methods do not work well with hyperbolic equations but do go hand in hand with parabolic-type equations. Therefore, a different classification is more valuable for our purposes. We divide time-parallel integration methods into two categories: methods that rely on coarsening and single-level methods. A concise summary of the most prominent methods of the two groups is provided in Section 1.1 and 1.2. Each group has advantages and disadvantages, depending on which differential equation we want to solve. In this thesis, we will develop, implement and test the limits of a single-level approach coupled with a high-order implicit time integrator for linear and nonlinear equations.

1.1. Coarsening methods

1.1.1. Parareal

Parareal was introduced by Lions et al. in 2001 [14], and nowadays almost became a synonym for parallel-in-time integration. Parallelization is achieved using two time integrators: a fine yet expensive propagator $\mathcal{F}$ and a coarse cheap propagator $\mathcal{G}$. In French: 'fine' and 'gross'. The fine propagator usually has a smaller step size and is of a higher order of accuracy than the coarse one. Parareal is an iterative method where an approximation of the solution $u(T_\ell) \approx u_\ell^k$ in iteration k is computed as

$$u_{\ell+1}^{k+1} = \mathcal{F}\left(u_\ell^k\right) + \mathcal{G}\left(u_\ell^{k+1}\right) - \mathcal{G}\left(u_\ell^k\right), \quad 0 \leq \ell < L.$$

With this, the expensive computations of $\mathcal{F}(u_\ell^k)$ can be performed in parallel, followed by sequential computations of $\mathcal{G}(u_\ell^{k+1})$. By design, the sequential part should be inexpensive, and after K iterations on L cores, one for each time step, an upper bound for speedup is L/K.

Parareal's convergence to the solution of the fine integrator is guaranteed after L iterations, however, speedup is only possible if the method converges faster. The method is known to work very well for diffusive problems, and a detailed convergence analysis in the linear case is provided in [16], both for the diffusion and the advection equation. In addition to the superlinear convergence bound for linear problems, it is shown that parareal can be viewed as a two-level space-time multigrid or as a shooting method. The convergence analysis for the nonlinear case can be found in [17].

However, parareal often fails to produce speedup for hyperbolic type equations, and convergence is sometimes achieved just in the last iteration. A study in [18] touches this subject for the advection equation and broadens the study for the wave equation, concluding that the method is *not very useful for parallelizing the solution of the wave equation in time [...]*. Later, a detailed analysis of wave propagation due to coarsening in time confirms this, concluding that *a key finding is that the source of the instability is different discrete phase speeds on the coarse and fine level, which cause instability of higher wave number modes*

even thought the *amplitude errors are quickly corrected by the iteration* [19]. Over the years, extensions and enhancements have been made [20, 21, 22]. Unfortunately, these extensions often lead to additional computations that have to be performed, yielding reduced efficiency altogether.

The method is often coupled with coarsening in space, yet another trick is how to make the $\mathcal{G}$ integrator even cheaper. Unfortunately, even without the time-coarsening aspect, spatial coarsening can also play a significant role, deteriorating the convergence of the method [23].

Currently, an implementation of the method can be found as a `Julia` package [24], a package written in `C` [25], and as a `Fortran 90` implementation `LibPFASST` [26].

1.1.2. Multigrid reduction in time

Inspired by the parareal algorithm, a multigrid approach MultiGrid Reduction In Time (MGRIT) [27] was developed, extending parareal to a multi-level algorithm that can use more than two levels of coarsening. The general idea is to partition an equidistant subdivision of the time domain into C-points, the coarse mesh, and F-points, the fine mesh. With this, corresponding restriction and prolongation operators are defined. A two-level multigrid can be explained as interpolating the solution from the coarse level to the fine level to correct the result on the fine level. This is called a coarse grid correction (CGC). In a multi-level multigrid in time, this process visits the coarsest level and then revisits the finer levels in some order, visiting the finest level again, yielding different cycle types.

The approach produces good speedups for parabolic problems for long time intervals when sufficiently many cores are used. Implementations can be found in `Xbraid` [25], written in `MPI/C` with `C++`, `Fortran 90`, and `Python` interfaces, and as a standalone `Python` package `pyMGRIT` [28]. Both implementations provide nonintrusive implementations as well as a variety of time stepping, relaxation, and temporal and spatial coarsening options.

However, the problems of not handling hyperbolic-type equations very well are inherited from parareal. The main part where MGRIT struggles with is the coarse-grid correction term which seemingly does not adequately correct smooth Fourier modes when using standard coarsening strategies. Extensive analysis for the linear advection equation is still ongoing, producing new coarsening strategies [29, 30, 31, 32] and achieveing modest speedups of around 8.5 on 1024 cores [33].

1.1.3. Parallel Full Approximation Scheme in Space and Time

The Parallel Full Approximation Scheme in Space and Time (PFASST) was initially proposed in 2012 by Emmett and Minion [34]. The idea of PFASST is to use Spectral Deferred Corrections (SDC) [35], an iterative approach to solve a collocation problem on a single time interval coupled with parareal. The collocation problem is formally introduced in Section 2.1. However, for now, it can be viewed as a high-order integration method, under certain conditions, equivalent to an Implicit Runge–Kutta (IRK) method. A version of SDC with coarsening in space and/or time is called Multi-Level Spectral Deferred Corrections (MLSDC) and is also used within PFASST. One iteration of these algorithms on a single time interval is called a *sweep*, and instead of using coarse and fine propagators $\mathcal{F}$ and $\mathcal{G}$, PFASST uses SDC sweeps to update $u_{\ell+1}^{k+1}$. This can be viewed as a modified coarse grid correction.

A multigrid point of view was explored in [36], concluding that *under certain assumptions, the PFASST algorithm can be conveniently and rigorously described as a multigrid-in-time method*. However, due to coarsening, the same problem of not handling hyperbolic-type equations very well is inherited from

parareal. One can find an asymptotic convergence theory and convergence comparisons for the linear heat and the linear advection equation in [37], in which the authors also conclude that *the advective case performs much worse than the diffusive case*. A case study comparing space-time-multigrid methods and PFASST is conducted for the Allen–Cahn equation in [38]. The most favorable setup for implicit Euler produced a speedup of almost 11 on 64 cores for PFASST [38, Table 5.8] and 9 on 128 cores for a seven level space-time multigrid without temporal coarsening [38, Table 5.7]. Speedup for 2 or more Radau-Right nodes is reduced.

The method produces significant speedups on longer time intervals, and a `Python` implementation of SDC and PFASST is available as `pySDC` [39]. An implementation written in `Fortran 90` can be found in `LibPFASST` [26], as well as a `C++` implementation [40][2].

1.2. Single-level methods

One of the promising approaches to tackle this underlying problem is to remove coarsening altogether. Because of this, one does not need to spend time on finding a good coarse-fine combination of propagators. Some methods that do not use coarsening are ParaExp [41] and REXI [42]. Other single-single level methods exploit small scale parallelism across the method. Some prominent examples include parallel RK methods [43, 44, 45] and parallel SDC [46]. Many more exist, however, we will briefly explain the idea of ParaExp and mostly focus on diagonalization-based methods.

1.2.1. ParaExp

ParaExp [41] is a direct time-parallel integrator defined for linear problems of form

$$u_t(t) = \mathbf{A}u(t) + g(t), \quad u(0) = u_0.$$

First, one solves the equation in parallel for every slice $[T_{\ell-1}, T_\ell]$ with a zero initial condition, obtaining solutions v_ℓ. Second, the homogeneous problems

$$w_t(t) = \mathbf{A}w(t), \quad w(T_{\ell-1}) = v_{\ell-1}(T_{\ell-1})$$

are solved for a time slice $[T_{\ell-1}, T_{\text{end}}]$, again in parallel, obtaining solutions w_ℓ. These parallel steps are carried out efficiently by approximating the matrix exponential. The solution for a slice $[T_{\ell-1}, T_\ell]$ is then obtained using the linearity of the problem as

$$u\big|_{[T_{\ell-1},T_\ell]} = v_\ell + \sum_{k=1}^{\ell} w_k\big|_{[T_{\ell-1},T_\ell]}.$$

Since it is a direct method, ParaExp is very suitable for hyperbolic problems. Because of this, ParaExp works excellently with all linear problems, achieving remarkable speedups.

[2] Not maintained since 2016.

1.2.2. Diagonalization-based methods

Another way to circumvent the coarsening is to use diagonalization techniques. An all-in-common base idea of these methods is an assembly of the all-at-once system of a time-stepping method,

$$\mathbf{Cu} := \begin{bmatrix} I & & & \\ -F & I & & \\ & \ddots & \ddots & \\ & & -F & I \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_L \end{bmatrix} = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_L \end{bmatrix}, \tag{1.1}$$

where $u_\ell \approx u(t_\ell)$, followed by either a direct system solve by diagonalization (when operators F differ across time steps) or an iterative method with an α-circulant preconditioner. The system (1.1) is given in its simplest form, satisfying propagations $u_{\ell+1} = \mathcal{F}(u_\ell) =: Fu_\ell + g_\ell$. Writing the problem in form (1.1) is a crucial reformulation, allowing us to analyze the problem from a different perspective. It is also used to analyze and explain time-parallel multigrid methods [27, 36].

A pioneering report from 2007 [47] explored the idea of diagonalization for the first time for linear problems, using a series of geometrically increasing large time step sizes. Because of this, the matrix in the all-at-once system (1.1) is diagonalizable. Later, a more in-depth application for wave equations was made, concluding that speedup is possible for a limited number of time steps due to *balancing carefully truncation and roundoff error* [48].

An iterative approach using the α-circulant preconditioner

$$\mathbf{C}_\alpha := \begin{bmatrix} I & & & -\alpha F \\ -F & I & & \\ & \ddots & \ddots & \\ & & -F & I \end{bmatrix} \tag{1.2}$$

for the all-at-once system was explored inside GMRES or MINRES iterations starting from 2003 [49, 50, 51, 52], showing in [52] an analysis of the spectrum and clustering of eigenvalues, an important factor leading to fast convergence of Krylov space methods. The preconditioner is of specific interest because it can be diagonalized using a parallel fast Fourier transform and it's inverse. After diagonalizing across time steps, the problems on the diagonal are decoupled and can be solved in parallel.

One can use the same preconditioner inside the Preconditioned Richardson Iterations (PRI), leading to a second iterative approach with good convergence properties and less required memory than GMRES. The idea of using PRI was already explored in 2018, in combination with parareal, as a coarse grid correction [53]. In 2019, the waveform relaxation technique in combination with diagonalization [54] was published, and later, in 2020, it was plugged into parareal as a coarse propagator $\mathcal{G}$ with the same time step size as the fine one [55]. The GMRES and the PRI approach were compared in [56], concluding that the number of iterations for both methods is almost identical.

A convergence analysis from 2021 can be found in [57], followed by an analysis from 2022 [52], showing that the spectral radius of the iteration matrix using PRI is bounded as $\rho((\mathbf{I} - \mathbf{C}_\alpha)^{-1}\mathbf{C}) \leq \alpha/(1-\alpha)$ for both first-order and second-order evolutionary problems, provided that the time integrator is stable. An overview of diagonalization-based PinT algorithms can be found in [58], including some comparisons to MGRIT, and followed on the PinT web page [59]. The forming umbrella term for these methods is ParaDiag.

1.2.3. Contribution and outline

So far, only simple time-stepping integrators were examined: a general two-point approximation of the integral which covers methods such as implicit or explicit Euler method and the trapezoidal rule. [3] The work on our topic with an idea to examine the PRI with the α-circulant preconditioner as a standalone solver outside the parareal framework and extend it to high-order integrators, namely the collocation problem, also covering high-order IRK methods.

Chapter 2 introduces and analyzes ParaDiag for linear equations. After diagonalizing the block circulant preconditioner bearing the collocation problem, the decoupled problems are also solved via diagonalization of the perturbed collocation matrix, yielding a doubly time-parallel method: across the time steps and across the collocation nodes. The diagonalization of inner systems is theoretically supported in Subsection 2.4. We do so by finding the scaled characteristic polynomial of a general perturbed collocation problem for every time step and α and use it to show under which conditions are the inner systems diagonalizable.

Unfortunately, the preconditioner (1.2) is ill-conditioned, and the round-off error analysis for this type of preconditioner needs to be improved. We extend it in Subsection 2.5.2 by examining the error originating from the poorly conditioned diagonalization while considering the inexactness of system solves in the diagonalized form. Using these results in Subsection 2.5.3, we show how one can balance the convergence rate and the round-off errors by choosing a different α in each iteration, leading to a reduced number of iterations when solving linear equations. This leads to improved performance, and we support this finding with a benchmark.

An extension on how to solve nonlinear equations is developed in Chapter 3. One way to solve a system of nonlinear equations arising from the all-at-once problem (1.1) is to use the inexact Newton's method [56, 55] or Newton–Krylov methods [52]. For the PRI, a natural extension is to use the former. In addition to that, in Subsection 3.1, a Jacobian-free iterative approach is introduced and analyzed for a single time step, treating the nonlinear part in an explicit way. We show that both approaches can be formulated as implicit-explicit (IMEX) preconditioning for the composite collocation problem in Subsection 3.2. With that, a convergence analysis on Dahlquist's test equation is presented in Subsection 3.2.1, unifying the convergence of both approaches: the inexact Newton's method and the Jacobian-free approach, and in the same form, includes convergence for the fully linear equations as a special case. We show that the upper bound for the error of these iterations depends almost linearly on the number of parallel time steps. In theory, Newton's method leads to faster convergence, as evident from our results since it needs fewer iterations. On the other hand, treating the nonlinear terms explicitly is used for equations when the Jacobian is a computational bottleneck or unavailable, e.g., the Boltzmann equation, since the right-hand side is a nonlinear collision term that dominates the computation time.

The reports of parallel implementations are almost as important as the theory itself, and they are often lacking when a new method or theory is presented. Therefore, we also contribute with a parallel implementation (in space and time), benchmarking the method for the linear heat, the linear advection equation, the nonlinear Allen–Cahn, and the nonlinear Boltzmann equation. The runs also consider the inexactness of system solves, leading to less efficient but more representative results of speedup. The implementation, thoroughly described in Chapter 4, is written in `Python` with the help of `mpi4py` and `petsc4py`. It is open source and available on GitHub [60].

[3] A parallel implementation of this integrator as preconditioned FGMRES iterations is available since 2021 within the Firedrake project, an automated system for the portable solution of partial differential equations using the finite element method. Avaliable under https://github.com/firedrakeproject/asQ.

Chapter 5 contains the presentation of the results. In Section 5.2.1, we emphasize the significance of selecting relevant test cases and parameter configurations when generating results. The section is structured using examples that demonstrate overperformance. Finally, our implementation's outcomes are exhibited in Sections 5.3 and 5.4.

We achieve significant speedups of around 10 on 64 cores when parallelizing across steps and up to 14 on 128 cores when additionally parallelizing across the method for the linear advection equation. With this, we outperform the mentioned results for MGRIT. Furthermore, when parallelizing in space, we can scale the algorithm further in time after the spatial cores reach the saturation point, achieving a speedup of 90 on 1536 cores. For the Allen–Cahn test case, an advection-diffusion equation, we achieve a speedup of 9 on 128 cores when solving with implicit Euler and Newton's method and around 5.5 using the Jacobian-free approach. When additionally parallelizing across 3 Radau-Right nodes, the speedup is more significant, resulting in around 50 when solving with Newton's method and 35 using the Jacobian-free approach on 192 cores. Compared to the previously mentioned results of PFASST, our results are comparable to the state-of-the-art algorithms.

CHAPTER 2

The ParaDiag method for linear equations

First, we start by explaining our base numerical integrator: the collocation problem. Collocation methods can be seen as a subclass of IRK methods, and we provide a discussion to show this in Section 2.1. Furthermore, the all-at-once system is constructed in Section 2.2, and an iterative approach is presented using the preconditioned Richardson iterations with an α-circulant preconditioner in Section 2.3.

We extend the original approach of ParaDiag that uses only two-point approximations of the integral with a collocation method as our base integrator. This choice not only yields time integrators of arbitrary order but also adds another level (and opportunity) of parallelization in time: across the collocation nodes for each time step via diagonalization of the perturbed collocation matrix. The detalis are described in Section 2.4. Following the taxonomy in [15], this yields a doubly-time parallel method: combining parallelization across the steps and parallelization across the method first proposed in [61].

Section 2.5 describes the convergence of ParaDiag coupled with the collocation problem for linear equations and the analysis of round-off errors arising from the diagonalization of the preconditioner. We present and analyze an adaptive strategy to select a new parameter α for each iteration in order to balance (1) the convergence rate of the method with (2) round-off errors arising from the diagonalization of the preconditioner, and (3) inner system solves of the decoupled time-steps. Finally, we present an algorithm for the practical selection of the adaptive preconditioners and demonstrate the speedup gains these methods can provide over sequential methods in Subsection 2.5.3.

2.1. Collocation methods

2.1.1. General definition

In this section, a collocation method for an initial value problem

$$u_t = f(t, u), \quad u(0) = u_0 \in \mathbb{C}, \tag{2.1}$$

is presented.

Let $0 \leq t_1 < t_2 < \cdots < t_M \leq \Delta T$ be a subdivision of $[0, \Delta T]$ using $M \in \mathbb{N}$ additional collocation nodes. Because of variable change in the integral, we can without loss of generality, assume that $\Delta T = 1$. A Picard integral formulation over an interval $[0, t_m]$ for (2.1) is

$$u(t_m) = u(0) + \int_0^{t_m} f(s, u(s))ds, \quad m = 1, \ldots, M. \tag{2.2}$$

Let c_i denote the i^{th} Lagrange polynomial in points $t_1, \ldots, t_M$ defined as

$$c_i(t) = \prod_{m=1, m\neq i}^{M} \frac{t - t_m}{t_i - t_m}, \quad i = 1, \ldots, M. \tag{2.3}$$

A polynomial approximation of the integrand f in (2.2) in these same collocation nodes $t_1, \ldots, t_M$ is

$$f(s, u(s)) \approx \sum_{i=1}^{M} f(t_i, u(t_i))c_i(s), \quad s \in [0, 1].$$

This yields an approximation of the whole integral:

$$\int_0^{t_m} f(s, u(s))ds \approx \int_0^{t_m} \Big(\sum_{i=1}^{M} c_i(s) f(t_i, u(t_i))ds \Big) \tag{2.4a}$$

$$= \sum_{i=1}^{M} \int_0^{t_m} c_i(s) f(t_i, u(t_i))ds \tag{2.4b}$$

$$= \sum_{i=1}^{M} \Big(\int_0^{t_m} c_i(s)ds \Big) f(t_i, u(t_i)). \tag{2.4c}$$

Combining the last equation (2.4c) with the integral formulation (2.2) yields

$$u(t_m) \approx u_0 + \sum_{i=1}^{M} \Big(\int_0^{t_m} c_i(s)ds \Big) f(t_i, u_i), \quad m = 1, \ldots, M,$$

and the approximation is as good as the integral approximation in (2.4a). With this, we can compute the approximate solutions by solving M equations

$$u_m = u_0 + \sum_{i=1}^{M} \Big(\int_0^{t_m} c_i(s)ds \Big) f(t_i, u_i), \quad m = 1, \ldots, M, \tag{2.5}$$

in variables u_m that approximate $u(t_m)$. The equations (2.5) can compactly be written in a matrix formulation known as the *collocation problem*:

$$\mathbf{u} - \mathbf{Q}\mathbf{f}(\mathbf{u}) = \mathbf{u}_0, \tag{2.6}$$

where $\mathbf{u} = [u_1, \ldots, u_M]^\top$, $\mathbf{f}(\mathbf{u}) = [f(u_1), \ldots, f(u_M)]^\top$, $\mathbf{u}_0 = [u_0, \ldots, u_0]^\top$ and the matrix $\mathbf{Q} \in \mathbb{R}^{M\times M}$ is defined as

$$[\mathbf{Q}]_{ij} = \int_0^{t_i} c_j(s)ds. \tag{2.7}$$

Generally, the approximate solution of the initial value problem on an interval of length $\Delta T \neq 1$, in (2.6) is written as

$$\mathbf{u} - \Delta T \mathbf{Q}\mathbf{f}(\mathbf{u}) = \mathbf{u}_0, \tag{2.8}$$

which can be verified by the corresponding change of variables in the integral.

The M equations (2.5) propagating a linear initial value problem

$$u_t = \mathbf{A}u + b(t), \quad u(0) = u_0, \tag{2.9}$$

where now $u \in \mathbb{C}^N$, $\mathbf{A} \in \mathbb{C}^{N\times N}$ is a constant matrix and $b : \mathbb{R} \to \mathbb{C}^N$, can be written as

$$(\mathbf{I}_{NM} - \Delta T \mathbf{Q} \otimes \mathbf{A})\mathbf{u} = \mathbf{u}_0 + \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{b}.$$

Here, $\mathbf{u} \in \mathbb{C}^{MN}$ and $\mathbf{b} = [b_1, \ldots, b_M]^\top \in \mathbb{C}^{MN}$ denotes a vector of the function b evaluated in the collocation nodes. Symbol $\otimes$ denotes a *Kronecker product* defined below.

Definition 2.1
The Kronecker product of $\mathbf{A} \in \mathbb{C}^{n\times n}$ *and* $\mathbf{B} \in \mathbb{C}^{m\times m}$ *is defined as*

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \ldots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \ldots & a_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}\mathbf{B} & a_{n2}\mathbf{B} & \ldots & a_{nn}\mathbf{B} \end{bmatrix} \in \mathbb{C}^{nm\times nm}.$$

2.1.2. Collocation methods as a subclass of Runge–Kutta methods

Collocation methods are linked to Implicit Runge–Kutta methods (IRK), more specifically, they are a subclass of IRK methods [62, Collocation methods]. The collocation matrix $\mathbf{Q} \in \mathbb{R}^{M\times M}$ and the IRK matrix as in the Butcher tableau are the same, when the IRK method is of order at least M. The collocation method is equivalent to an IRK method when the last node contains an interval end-point $t_M = \Delta T$. Otherwise, if $t_M < \Delta T$, the solutions $u_1, \ldots, u_M$ of the collocation problem are just internal stages of a M-stage IRK scheme, and the last step of constructing the approximation in point ΔT is missing. We will support our claims with a discussion, simultaniously proving a theorem that states necessary order requirements from the literature.

A generic M-stage Runge–Kutta method is defined as

$$k_i = f\left(T_n + c_i\Delta T, y_n + \Delta T \sum_{j=1}^{M} \widetilde{a}_{ij}k_j\right), \quad i = 1, \ldots, M,$$

accompanied by a prolongation equation

$$y_{n+1} = y_n + \Delta T \sum_{i=1}^{M} b_i k_i. \tag{2.10}$$

The vectors values b_i, c_i and $\widetilde{a}_{ij}$ are formally stored in the Butcher tableau as:

$$\begin{array}{c|cccc} c_1 & \widetilde{a}_{11} & \widetilde{a}_{12} & \ldots & \widetilde{a}_{1M} \\ c_2 & \widetilde{a}_{21} & \widetilde{a}_{22} & \ldots & \widetilde{a}_{2M} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_M & \widetilde{a}_{M1} & \widetilde{a}_{M2} & \ldots & \widetilde{a}_{MM} \\ \hline & b_1 & b_2 & \ldots & b_M \end{array},$$

or, written compactly as

$$\begin{array}{c|c} \mathbf{c} & \widetilde{\mathbf{A}} \\ \hline & \mathbf{b}^\top \end{array}.$$

Let

$$u_i = y_n + \Delta T \sum_{j=1}^{M} \widetilde{a}_{ij} k_j,$$

and the definitions of k_i now read

$$k_i = f(T_n + c_i \Delta T, u_i).$$

Alltogether, written line by line in a matrix formulation, we have

$$\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_M \end{bmatrix} = \begin{bmatrix} y_n \\ y_n \\ \vdots \\ y_n \end{bmatrix} + \Delta T \widetilde{\mathbf{A}} \begin{bmatrix} f(t_1, u_1) \\ f(t_2, u_2) \\ \vdots \\ f(t_M, u_M) \end{bmatrix},$$

where $t_i = T_n + c_i \Delta T$. Observed more closely, this is exactly the formulation of the collocation problem, where y_n denotes the approximation of the solution in point t_n. The prolongation equation (2.10) now reads

$$\begin{aligned} y_{n+1} &= y_n + \Delta T \sum_{i=1}^{M} b_i k_i \\ &= y_n + \Delta T \sum_{i=1}^{M} b_i f(t_i, u_i). \end{aligned}$$

Now, let $\mathbf{Q}$ be a collocation matrix defined in (2.7). It holds

$$\mathbf{Q} \begin{bmatrix} t_1^m \\ t_2^m \\ \vdots \\ t_M^m \end{bmatrix} = \begin{bmatrix} \int_{T_n}^{t_1} \sum_j t_j^m c_j(x) dx \\ \int_{T_n}^{t_2} \sum_j t_j^m c_j(x) dx \\ \vdots \\ \int_{T_n}^{t_M} \sum_j t_j^m c_j(x) dx \end{bmatrix} = \begin{bmatrix} \int_{T_n}^{t_1} p(x) dx \\ \int_{T_n}^{t_2} p(x) dx \\ \vdots \\ \int_{T_n}^{t_M} p(x) dx \end{bmatrix},$$

where p is a polynomial of degree less or equal than $M - 1$ (since the Lagrange polynomials are of degree $M - 1$) containing points $\left(t_1, t_1^m\right), \ldots, \left(t_M, t_M^m\right)$. From here, we conclude that $p(x) = x^m$. Now, for $T_n = 0$, the integrals equal to

$$\int_{T_n}^{t_i} p(x) dx = \int_0^{t_i} x^m dx = \frac{t_i^{m+1}}{m+1}, \quad m = 0, \ldots, M-1. \tag{2.11}$$

In conclusion, the IRK matrix $\widetilde{\mathbf{A}}$ and the collocation matrix $\mathbf{Q}$ are the same when the order conditions (2.11) are satisfied. Another condition is that t_i have to be distinct, otherwise the Lagrange polynomials forming a basis cannot be formed. Formally, this proves the following theorem from [63, Theorem 1.1.1].

Theorem 2.2
The M-stage implicit Rung–Kutta method with distinct parameters c_i and order at least M can be obtained by collocation if and only if the relations (2.11) *hold.*

2.1.3. High-order methods

Solving the collocation problem (2.8) yields an approximation of the solution, and the distance to the exact solution of the equation (2.1) can be expressed through the order of the underlying quadrature rule. For this purpose, we state the local error theorem for collocation methods from [62, p. 500, Theorem 4.9], rephrased to fit our object definitions.

Theorem 2.3
If $t_m \neq 0$ for all m and $t_M = 1$, then the local error of the collocation method (2.8) *satisfies*

$$u(\Delta T) - u_m = O\left(\Delta T^{p+1}\right),$$

where p is the order of the underlying quadrature formula (2.4a).

The theorem provides us with a direct recipe for how to derive high-order methods: we can choose a high-order quadrature rule that includes the right side of the interval. Then, the weights of he quadrature rule are stored in the last row of the matrix $\mathbf{Q}$. Some widely used quadrature rules are the Radau-Right quadrature with a local error $O\left(\Delta T^{2M}\right)$ or the Lobatto quadrature with a local error $O\left(\Delta T^{2M-1}\right)$. Many more exist and can be found in [62, Chapter VII.4] under the subsection *Collocation methods*.

Viewing the basic numerical propagator as the collocation method provides us with a structured matrix $\mathbf{Q}$, simplifying the theoretical analysis, and yet, it leaves room for generalization of the collocation nodes.

2.2. The linear composite collocation problem

Let $\mathbf{C}_{\text{coll}} := \mathbf{I}_{MN} - \Delta T \mathbf{Q} \otimes \mathbf{A} \in \mathbb{C}^{MN \times MN}$ and let $\mathbf{H} := \mathbf{H}_M \otimes \mathbf{I}_N$, where

$$\mathbf{H}_M = \begin{bmatrix} 0 & \dots & 1 \\ \vdots & & \vdots \\ 0 & \dots & 1 \end{bmatrix} \in \mathbb{R}^{M \times M}. \tag{2.12}$$

A classical (sequential) approach to solve equation (2.9) with the collocation method for L time steps is first solving

$$\mathbf{C}_{\text{coll}} \mathbf{u}_1 = \mathbf{u}_0 + \mathbf{v}_1, \tag{2.13}$$

and then sequentially solving

$$\mathbf{C}_{\text{coll}} \mathbf{u}_\ell = \mathbf{H} \mathbf{u}_{\ell-1} + \mathbf{v}_{\ell-1}, \tag{2.14}$$

for $\ell = 2, \dots, L$. Here, $\mathbf{v}_\ell = \Delta T \left(\mathbf{Q} \otimes \mathbf{I}_N\right) \mathbf{b}_\ell$, where $\mathbf{b}_\ell$ is the function b evaluated at the corresponding collocation nodes $T_\ell + t_1 < T_\ell + t_2 < \dots < T_\ell + t_M = T_{\ell+1}$. The matrix $\mathbf{H}$ serves to utilize the previously obtained solution $\mathbf{u}_{\ell-1}$ as the initial condition for computing $\mathbf{u}_\ell$. For L time steps, equations (2.13) and (2.14) can be cast as an *all-at-once* or a *composite collocation* system:

$$\begin{bmatrix} \mathbf{C}_{\text{coll}} & & & \\ -\mathbf{H} & \mathbf{C}_{\text{coll}} & & \\ & \ddots & \ddots & \\ & & -\mathbf{H} & \mathbf{C}_{\text{coll}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_L \end{bmatrix} = \begin{bmatrix} \mathbf{u}_0 + \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_L \end{bmatrix}. \tag{2.15}$$

More compactly, this can be written as

$$\mathbf{C}\vec{\mathbf{u}} := (\mathbf{I}_L \otimes \mathbf{C}_{\text{coll}} + \mathbf{E} \otimes \mathbf{H})\vec{\mathbf{u}} = \vec{\mathbf{w}} \tag{2.16}$$

with $\vec{\mathbf{u}} = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_L]^\top \in \mathbb{C}^{LMN}$, $\vec{\mathbf{w}} = [\mathbf{u}_0 + \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_L]^\top \in \mathbb{C}^{LMN}$, and the matrix

$$\mathbf{E} := \begin{bmatrix} 0 & & & \\ -1 & 0 & & \\ & \ddots & \ddots & \\ & & -1 & 0 \end{bmatrix} \in \mathbb{R}^{L \times L},$$

accounting for the transfer of the solution from one step to another.

Solving the composite collocation problem via block forward substitutions produces the sequential approach (2.13)–(2.14). Often, to solve a linear system, iterative methods that rely on preconditioning are deployed. A standard fixed point iteration method, Preconditioned Richardson Iteration (PRI) has the form

$$\mathbf{P}\vec{\mathbf{u}}^{(k+1)} = (\mathbf{P} - \mathbf{C})\,\vec{\mathbf{u}}^{(k)} + \vec{\mathbf{w}},$$

where the preconditioning matrix $\mathbf{P}$ should be cheaper to invert than the system matrix $\mathbf{C}$, but 'close' to it in some sense. Time-parallel integration aims to manufacture a method that can be implemented in parallel across time steps, computing the solution faster than what is obtainable with the block forward substitutions. Strictly speaking, an iteration matrix $\mathbf{I} - \mathbf{P}^{-1}\mathbf{C}$ that has a small spectral radius yields fast convergence. Simultaneously, the idea is to find a preconditioner for which $\mathbf{P}^{-1}$ can be computed in parallel. If the preconditioner can be block-diagonalized, each system solve can be handled in parallel. With this, the most costly part of the computation is solved across multiple cores.

2.3. The ParaDiag approach for the linear composite collocation problem

For ParaDiag methods, the preconditioner is a block-circulant Toeplitz matrix and has been previously analyzed and used [64, 65, 50, 55, 52], however, it was yet not coupled with collocation methods. In our case, the preconditioner is defined as

$$\mathbf{C}_\alpha := \mathbf{I}_L \otimes \mathbf{C}_{\text{coll}} + \mathbf{E}_\alpha \otimes \mathbf{H}, \quad \mathbf{E}_\alpha := \begin{bmatrix} 0 & & & -\alpha \\ -1 & 0 & & \\ & \ddots & \ddots & \\ & & -1 & 0 \end{bmatrix}. \tag{2.17}$$

This yields an iteration of the form

$$\mathbf{C}_\alpha\vec{\mathbf{u}}^{(k+1)} = (\mathbf{C}_\alpha - \mathbf{C})\vec{\mathbf{u}}^{(k)} + \vec{\mathbf{w}}. \tag{2.18}$$

Conveniently, $\mathbf{E}_\alpha$ can be diagonalized.

Lemma 2.4
The matrix $\mathbf{E}_\alpha \in \mathbb{R}^{L\times L}$ *defined in* (2.17) *can be diagonalized as* $\mathbf{E}_\alpha = \mathbf{V}\mathbf{D}\mathbf{V}^{-1}$, *where* $\mathbf{V} = \frac{1}{L}\mathbf{J}\mathbf{F}$ *and*

$$\begin{aligned}
\mathbf{D}_\alpha &= \operatorname{diag}(d_1, \dots, d_L),\ d_\ell = -\alpha^{\frac{1}{L}} e^{-2\pi \mathrm{i}\frac{\ell-1}{L}},\\
\mathbf{J} &= \operatorname{diag}\left(1, \alpha^{-\frac{1}{L}}, \dots, \alpha^{-\frac{L-1}{L}}\right),\\
[\mathbf{F}]_{jk} &= e^{2\pi \mathrm{i}\frac{(j-1)(k-1)}{L}},\ 1 \le j,k \le L,\\
\mathbf{V}^{-1} &= \mathbf{F}^*\mathbf{J}^{-1}.
\end{aligned}$$

Proof: The proof can be found in [66, Lemma 4]. □

The diagonalization property of $\mathbf{E}_\alpha$ involving the scaled Fourier matrix allows parallelization across time steps. Using the diagonalization property, each iteration in (2.18) can be computed in three steps:

$$\vec{\mathbf{x}} = (\mathbf{V}^{-1} \otimes \mathbf{I}_{MN})((\mathbf{C}_\alpha - \mathbf{C})\vec{\mathbf{u}}^{(k)} + \vec{\mathbf{w}}), \tag{2.19a}$$

$$(\mathbf{D}_\alpha \otimes \mathbf{H} + \mathbf{I}_L \otimes \mathbf{C}_{\text{coll}})\vec{\mathbf{y}} = \vec{\mathbf{x}}, \tag{2.19b}$$

$$\vec{\mathbf{u}}^{(k+1)} = (\mathbf{V} \otimes \mathbf{I}_{MN})\vec{\mathbf{y}}. \tag{2.19c}$$

Step (2.19b), which we assume will be the most time-consuming one since it involves system solves, can now be performed in parallel because the matrix is block-diagonal (see Figure 2.1) and steps (2.19a) and (2.19c) can be computed with a parallel Fast Fourier Transform (FFT) in time.

Using preconditioners in the fixed point iterations often has a price between cheap computation but slow convergence and more expensive computation yet better convergence. In this case, the difference between the system matrix and the preconditioner is α, and for a very small value, one would expect blazing convergence. However, that is not the case since for very small α values since the system is also 'closer' to a Jordan block, i.e., a non-diagonalizable structure. Because of this, the diagonalization of the preconditioner introduces round-off errors that degrade convergence. We perform a more in depth analysis of how α influences the iterations in Section 2.5.

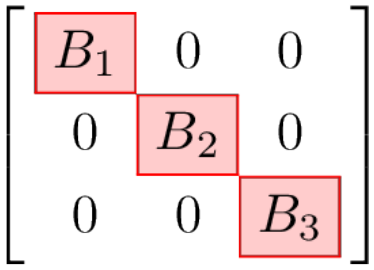


Figure 2.1.: A block-diagonal matrix.

The value α also has a physical interpretation. Before transferring the vector $(\mathbf{C}_\alpha - \mathbf{C})\vec{\mathbf{u}}^{(k)} + \vec{\mathbf{w}}$ into a discrete Fourier space (which is essentially a change of function space in a discrete way) in equation (2.19a), the ℓ^{th} block-entry is scaled with $\alpha^{(\ell-1)/L}$, originating from the matrix $\mathbf{J}^{-1}$. The parameter $\alpha < 1$ can be seen as a dampening factor of sorts because it dictates how much the blocks dampen across time steps before solving the systems in the frequency space. Most of the round-off errors are then introduced by rescaling the vector back, i.e., multiplying with $\mathbf{J}$.

To solve (2.19b), we propose yet another diagonalization strategy. The block diagonal problems in equation (2.19b) can be expressed as

$$(d_\ell\mathbf{H} + \mathbf{C}_{\text{coll}})\mathbf{y}_\ell = \mathbf{x}_\ell, \quad \ell = 1, \dots, L,$$

for $d_\ell \in \mathbb{C}$, or in other words,

$$((d_\ell\mathbf{H}_M + \mathbf{I}_M) \otimes \mathbf{I}_N - \Delta T\mathbf{Q} \otimes \mathbf{A})\mathbf{y}_\ell = \mathbf{x}_\ell, \quad \ell = 1, \dots, L. \tag{2.20}$$

Here, $\mathbf{x}_\ell \in \mathbb{C}^{MN}$ denotes the ℓ^{th} block of the vector $\vec{\mathbf{x}} \in \mathbb{C}^{LMN}$. Let us define

$$\mathbf{G}_\ell := \mathbf{I}_M + d_\ell \mathbf{H}_M = \begin{bmatrix} 1 & & & d_\ell \\ & 1 & & d_\ell \\ & & \ddots & \vdots \\ & & & 1 + d_\ell \end{bmatrix}, \quad d_\ell = -\alpha^{\frac{1}{L}} e^{-2\pi \mathrm{i} \frac{\ell-1}{L}}, \tag{2.21}$$

for each $\ell = 1, \ldots, L$, an upper triangular matrix of size $M \times M$, nonsingular when $\alpha \neq 1$, with an inverse

$$\mathbf{G}_\ell^{-1} := \mathbf{I}_M - r_\ell \mathbf{H}_M = \begin{bmatrix} 1 & & & -r_\ell \\ & 1 & & -r_\ell \\ & & \ddots & \vdots \\ & & & 1 - r_\ell \end{bmatrix}, \quad r_\ell := \frac{d_\ell}{1 + d_\ell}. \tag{2.22}$$

Now, we propose to solve the linear systems (2.20) in two steps:

$$\left(\mathbf{I}_{MN} - \Delta T \left(\mathbf{Q}\mathbf{G}_\ell^{-1}\right) \otimes \mathbf{A}\right) \mathbf{z}_\ell = \mathbf{x}_\ell, \tag{2.23a}$$

$$(\mathbf{G}_\ell \otimes \mathbf{I}_N)\mathbf{y}_\ell = \mathbf{z}_\ell. \tag{2.23b}$$

Let us suppose that $\mathbf{Q}\mathbf{G}_\ell^{-1}$ is a diagonalizable matrix for a given ℓ and α. In that case, one can solve (2.23a) by diagonalizing a much smaller $M \times M$ matrix and inverting the matrices on the diagonal in parallel, but now across the collocation nodes (or, as cast in [15], *across the method*). More precisely, let $\mathbf{Q}\mathbf{G}_\ell^{-1} = \mathbf{S}_\ell \mathbf{D}_\ell \mathbf{S}_\ell^{-1}$ denote the diagonal factorization, where $\mathbf{D}_\ell = \operatorname{diag}(d_{\ell 1}, \ldots, d_{\ell M})$. Hence, the inner systems in (2.23a) can now be solved in three steps for each $\ell = 1, \ldots, L$ as

$$(\mathbf{S}_\ell \otimes \mathbf{I}_N)\, \mathbf{x}_\ell^1 = \mathbf{x}_\ell, \tag{2.24a}$$

$$(\mathbf{I}_N - d_{\ell m} \Delta T \mathbf{A})\, \mathbf{x}_{\ell m}^2 = \mathbf{x}_{\ell m}^1, \quad m = 1, \ldots, M, \tag{2.24b}$$

$$\left(\mathbf{S}_\ell^{-1} \otimes \mathbf{I}_N\right) \mathbf{z}_\ell = \mathbf{x}_\ell^2, \tag{2.24c}$$

where $\mathbf{x}_{\ell m} \in \mathbb{C}^N$ denotes the m^{th} block of $\mathbf{x}_\ell \in \mathbb{C}^{NM}$.

With this, we end up with a doubly-time-parallel method: parallel across time steps and across the method, that iteratively solves an initial system (2.16) of size LMN but relies on solving systems of size N in each iteration. If LM cores are used, the systems are solved simultaneously, and ideally, one iteration should be as time-consuming as solving the smaller linear system of size N. The exact circumstances under which $\mathbf{Q}\mathbf{G}_\ell^{-1}$ is diagonalizable are discussed in Section 2.4 and a summary of the whole method is presented as a pseudo-code in Algorithm 1. It traces the steps described above and indicates which parts can be run in parallel. We will describe how a sequence of preconditioners, $(\mathbf{C}_{\alpha_k})_{k \in \mathbb{N}}$, can be prescribed, balancing the round-off errors and the convergence rate, as well as a stopping criterion in Section 2.5.3.

2.4. Diagonalization of $\mathbf{Q}\mathbf{G}_\ell^{-1}$

The section addresses the diagonalization of inner systems. It makes significant contributions, opening a possibility to solve the perturbed collocation problems in parallel across the method for every time step and α. Even though the section's highlight is Lemma 2.5, the whole subsection is based on setting up the theory to prove it and a recipe for utilizing it. For the Radau-Right quadrature with 2 and 3 collocation nodes, we show that for a certain number of time steps L, the diagonalization is possible for values of α

Algorithm 1 Linear ParaDiag iterations with the collocation problem and an $(\alpha_k)_{k\in\mathbb{N}}$ sequence.

Input: $(\mathbf{C}_{\alpha_k})_{k\in\mathbb{N}}$, $\mathbf{C}$, $\vec{\mathbf{w}}$, $\vec{\mathbf{u}}^{(0)}$
Output: a solution to $\mathbf{C}\vec{\mathbf{u}} = \vec{\mathbf{w}}$.

$k = 0$
while not done **do**
  $\mathbf{D}_{\alpha_k} = \mathrm{diag}(d_1, \dots, d_L)$, $d_\ell = -\alpha_k^{\frac{1}{L}} e^{-2\pi i \frac{l-1}{L}}$
  $\vec{\mathbf{r}} = (\mathbf{C}_{\alpha_k} - \mathbf{C})\vec{\mathbf{u}}^{(k)} + \vec{\mathbf{w}}$ # compute the right-hand side
  $\widetilde{\vec{\mathbf{r}}} = (\mathbf{J}^{-1} \otimes \mathbf{I}_{MN})\vec{\mathbf{r}}$
  $\vec{\mathbf{x}} = \mathbf{FFT}(\vec{\mathbf{r}})$ # perform the parallel FFT

  for $l = 1, \dots, L$ **in parallel do** # solve on L parallel steps
    $\mathbf{G}_\ell = d_\ell \mathbf{H}_M + \mathbf{I}_M$
    $\mathbf{QG}_\ell^{-1} = \mathbf{S}_\ell \mathbf{D}_\ell \mathbf{S}_\ell^{-1}$, $\mathbf{D}_\ell = \mathrm{diag}(d_{l1}, \dots, d_{lm})$ # digonalize the matrix $\mathbf{QG}_\ell^{-1}$
    $\mathbf{x}_\ell^1 = (\mathbf{S}_\ell^{-1} \otimes \mathbf{I}_N)\mathbf{x}_\ell$

    for $m = 1, \dots, M$ **in parallel do** # solve on M parallel nodes
      solve $(\mathbf{I}_N - d_{lm}\Delta T \mathbf{A})\mathbf{x}_{lm}^2 = \mathbf{x}_{lm}^1$

    $\mathbf{z}_\ell = (\mathbf{S}_\ell \otimes \mathbf{I}_N)\mathbf{x}_\ell^2$
    $\mathbf{y}_\ell = (\mathbf{G}_\ell^{-1} \otimes \mathbf{I}_N)\mathbf{z}_\ell$

  $\widetilde{\vec{\mathbf{u}}}^{(k+1)} = \mathbf{IFFT}(\vec{\mathbf{y}})$ # perform the parallel IFFT
  $\vec{\mathbf{u}}^{(k+1)} = (\mathbf{J} \otimes \mathbf{I}_{MN})\widetilde{\vec{\mathbf{u}}}^{(k+1)}$ # get the new iterate
  $k = k + 1$
return $\vec{\mathbf{u}}^{(k)}$

that do not expect to generate a too-large round-off error. Using Lemma 2.5, the same analysis can be extended to any quadrature, not necessarily high-order, such as equidistant nodes.

In order to solve (2.23a) via diagonalization, we need to study when diagonal factorization of $\mathbf{QG}_\ell^{-1}$ is possible. A sufficient condition is that the eigenvalues of $\mathbf{QG}_\ell^{-1}$ are distinct. Here, we will show that for our matrix $\mathbf{QG}_\ell^{-1}$ this is also a necessary condition.

First, let us observe what does a matrix-vector multiplication with the matrix $\mathbf{Q}$ mean. Let $\mathbf{y} \in \mathbb{C}^M$, recalling the definition of $\mathbf{Q}$ (2.7), we have

$$[\mathbf{Qy}]_m = \int_0^{t_m} \sum_{i=1}^{M} \mathbf{y}_i c_i(s) ds, \tag{2.25}$$

where c_i are the Lagrange interpolation polynomials defined in the collocation nodes. In other words, $\mathbf{Q}$ is integrating a polynomial of degree $M-1$ interpolated through points $(t_1, \mathbf{y}_1) \dots, (t_M, \mathbf{y}_M)$. The only assumption on the nodes in this subsection is that they are distinct and positive $0 \le t_1 < \dots < t_M$.

Now, let us recompute $\mathbf{QG}_\ell^{-1}$, where $\mathbf{G}_\ell^{-1}$ is written in (2.22). We know that for $\mathbf{D}_t := \mathrm{diag}(t_1, \dots, t_M)$, $\mathbf{QH}_M = \mathbf{D}_t \mathbf{H}_M$ holds, since $\mathbf{Q}$ is the integration matrix. In conclusion, we can write

$$\mathbf{QG}_\ell^{-1} = \mathbf{Q} - r_\ell \mathbf{QH}_M = \mathbf{Q} - r_\ell \mathbf{D}_t \mathbf{H}_M. \tag{2.26}$$

To formulate the eigenproblem, we need to find $\lambda \in \mathbb{C}$ and $\mathbf{y} \neq \mathbf{0}$ such that $\mathbf{Q}\mathbf{G}_\ell^{-1}\mathbf{y} = \lambda\mathbf{y}$. Because of the reformulation of $\mathbf{Q}\mathbf{G}_\ell^{-1}$ in (2.26) and (2.25), the eigenproblem can be reformulated as: finding a polynomial h of degree $M-1$ such that

$$\int_0^{t_m} h(s)ds + r_\ell t_m h(t_M) = \lambda h(t_m), \quad m = 1, \dots, M.$$

Substituting h with a derivative g', where the polynomial g is of degree M, yields

$$g(t_m) - g(0) + r_\ell t_m g'(t_M) = \lambda g'(t_m), \quad m = 1, \dots, M. \tag{2.27}$$

From here we see that the eigenvector represented as a polynomial g is not uniquely defined, because if g is a solution of (2.27), then $a_M g(t) + a_0$ is also a solution. Without loss of generality, we set $g(0) = 0$ so that g takes the form

$$g(t) = t^M + a_{M-1}t^{M-1} + \cdots + a_1 t.$$

With this, we have M nonlinear equations (2.27) and M unknowns $(a_{M-1}, \dots, a_1, \lambda)$. The reformulation of the problem leads us to the following lemma from which we will see that g is uniquely defined. This is similar to the statement that an eigenproblem with distinct eigenvalues has a unique eigenvector if it is a unit vector.

Lemma 2.5
Define

$$w_M(t) := (t - t_M)\dots(t - t_1) = t^M + b_{M-1}t^{M-1} + \cdots + b_0.$$

Then, the eigenvalues of $\mathbf{Q}\mathbf{G}_\ell^{-1}$ *are the roots of*

$$p_M(\lambda) = M!\lambda^M + c_{M-1}\lambda^{M-1} + \cdots + c_0,$$

where the coefficients are defined as $c_0 = (r_\ell + 1)b_0$,

$$c_m = m!\, b_m - r_\ell \sum_{j=1}^{M-m} \frac{(m+j)!}{j!} b_{m+j}, \quad 1 \le m \le M-1,$$

and $b_M = 1$.

Proof: Without loss of generality let $t_M = 1$. The eigenproblem is equivalent to solving M nonlinear equations

$$g(t_m) + r_\ell t_m g'(1) - \lambda g'(t_m) = 0, \quad m = 1, \dots, M, \tag{2.28}$$

where $g(t) = t^M + a_{M-1}t^{M-1} + \cdots + a_1 t$ and $\lambda \in \mathbb{C}$. Now, we define

$$G(t) := g(t) + r_\ell t g'(1) - \lambda g'(t).$$

It holds that $G(t_m) = w_M(t_m)$ for $m = 1, \dots M$ and that the difference $G - w_M$ is a polynomial of degree at most $M-1$, since both G and w_M are monic polynomials of degree M. Because $G - w_M$ is zero in M distincs points, we conclude that $G = w_M$. Equations (2.28) can compactly be rewritten as

$$g(t) + r_\ell t g'(1) - \lambda g'(t) = w_M(t). \tag{2.29}$$

Since the polynomial coefficients on the left-hand side of (2.29) are equal to the ones on the right-hand side, we get

$$\begin{aligned}
-\lambda a_1 &= b_0 && \text{(*0)}\\
a_1 + r_\ell g'(1) - 2\lambda a_2 &= b_1 && \text{(*1)}\\
a_2 - 3\lambda a_3 &= b_2 && \text{(*2)}\\
&\vdots\\
a_{M-2} - (M-1)\lambda a_{M-1} &= b_{M-2} && \text{(*M-2)}\\
a_{M-1} - M\lambda &= b_{M-1}. && \text{(*M-1)}
\end{aligned}$$

Telescoping these equations starting from (*M-1) up to (*m) for $m \geq 2$, we get

$$a_m = \sum_{j=0}^{M-m} \lambda^j \frac{(m+j)!}{m!} b_{m+j},$$

and from (*0) we get $a_1 = -b_0/\lambda$. Note that $\lambda \neq 0$ since both $\mathbf{Q}$ and $\mathbf{G}_\ell$ are nonsingular. The fact that $\mathbf{Q}$ is a nonsingular matrix is visible because it is a mapping in a fashion

$$\left(t_1^m, \ldots, t_M^m\right) \to \frac{1}{m+1}\left(t_1^{m+1}, \ldots, t_M^{m+1}\right), \quad 0 \leq m \leq M-1. \tag{2.31}$$

The vectors in (2.31) form columns of the Vandermonde matrix $\mathbf{W}$ which is known to be nonsingular when $0 < t_1 < \cdots < t_M$. Because of (2.31), we have $\mathbf{Q}\mathbf{W} = \mathbf{D}_M\mathbf{W}$, where $\mathbf{D}_M = \operatorname{diag}(1, 1/2, \ldots, 1/M)$. Now, from here we see that $\mathbf{Q}$ is nonsingular as a product of nonsingular matrices.

Substituting the expression for a_2 and a_1 into (*1) and multiplying it with $-\lambda \neq 0$ yields

$$-\lambda r_\ell g'(1) + \sum_{m=0}^{M} m!\, \lambda^m b_m = 0. \tag{2.32}$$

It remains to compute $g'(1)$. We have

$$\begin{aligned}
g'(1) &= M + (M-1)a_{M-1} + \cdots + 2a_2 + a_1\\
&= M + \sum_{m=2}^{M-1} m \sum_{j=0}^{M-m} \lambda^j \frac{(m+j)!}{m!} b_{m+j} - \frac{b_0}{\lambda},
\end{aligned}$$

yielding

$$\lambda g'(1) = \sum_{m=2}^{M} \sum_{j=0}^{M-m} \lambda^{j+1} \frac{(m+j)!}{(m-1)!} b_{m+j} - b_0.$$

Now we have to reorder and shift the summations by 1:

$$\lambda g'(1) + b_0 = \sum_{m=1}^{M-1} \sum_{j=1}^{M-m} \lambda^j \frac{(m+j)!}{m!} b_{m+j} = \sum_{j=1}^{M-1} \lambda^j \sum_{m=1}^{M-j} \frac{(m+j)!}{m!} b_{m+j}. \tag{2.33}$$

Combining (2.32) and (2.33) gives a polynomial in λ with coefficients being exactly as stated in the lemma. □

Remark 2.6
When $r_\ell = 0$, the resulting polynomial p_M from Lemma 2.5 is a scaled characteristic polynomial of the collocation matrix $\mathbf{Q}$*. This is true because of relation* (2.26)*.*

From the proof of lemma 2.5, it is visible that each λ generates exactly one polynomial g representing an eigenvector, therefore the matrix $\mathbf{Q}\mathbf{G}_\ell^{-1}$ is diagonalizable if and only if the eigenvalues are distinct.

Lemma 2.5 also provides an analytic way of pinpointing the values r_ℓ where $\mathbf{Q}\mathbf{G}_\ell^{-1}$ is not diagonalizable. First, we have to formulate the polynomial of roots w_M of the underlying quadrature. The M^{th} orthognal polynomial R_M, which roots are the quadrature nodes, can be obtained with a recursive formula [67]. Since R_M usually has roots in the interval $[-1, 1]$, a linear substitution $x(t)$ is required to obtain the corresponding collocation nodes in $[0, 1]$. Then, $w_M(t)$ is the monic polynomial colinear to $(R_M \circ x)(t)$.

Now it remains to find the values of r_ℓ for which p_M defined in Lemma 2.5 has distinct roots. This can be done using the discriminant of the polynomial and computing these finitely many values since the discriminant of a polynomial Δ_{p_M} is nonzero if and only if the roots are distinct. The discriminant of a polynomial is defined as

$$\Delta_{p_M} := \frac{(-1)^{M(M-1)/2}}{c_M \operatorname{Res}\left(p_M, p'_M\right)},$$

where c_M is the leading coefficient of p_M and Res is the residual between two polynomials. More formally, the residual is a polynomial $r(\lambda) := \operatorname{Res}\left(p_M(\lambda), p'_M(\lambda)\right)$ that satisfies

$$p_M(\lambda) = q(\lambda)p'_M(\lambda) + r(\lambda), \quad \deg(r) < \deg\left(p'_M\right),$$

where q is the quotient in the polynomial division.

For $M = 2, 3$ points in the Radau-Right quadrature, we give the analysis here. Note that for $M = 1$, the diagonalization is trivial since the matrix $\mathbf{Q}$ is of size 1×1.

Example 2.7
When $M = 2$, the corresponding polynomial of the Radau-Right quadrature is $w_2(t) = t^2 - \frac{4}{3}t + \frac{1}{3}$, generating $p_2(\lambda) = 2\lambda^2 - (\frac{4}{3} + 2r)\lambda + \frac{r+1}{3}$. Solving the equation $\Delta_{p_2} = 0$ is equivalent to $9r^2 + 6r - 2 = 0$ and the solutions are $r_* = \frac{-1\pm 3\sqrt{3}}{13}$. From here, we can compute α_* which could generate these values. Then, the matrix $\mathbf{Q}\mathbf{G}_\ell^{-1}$ is not diagonalizable for $\alpha_* \approx 0.323^L$, 0.477^L and these values should be avoided. These values are already of order 10^{-8} for $L \approx 25$ which is something that should be avoided anyways since such a smalle α tends to generate a large round-off error for the outer diagonalization.

Example 2.8
When $M = 3$, the corresponding polynomial Radau-Right quadrature is $w_3(t) = t^3 - \frac{9}{5}t^2 + \frac{9}{10}t - \frac{1}{10}$, generating $p_3(\lambda) = 6\lambda^3 - (\frac{18}{5} + 6r)\lambda^2 + (\frac{9}{10} + \frac{3}{5}r)\lambda - \frac{r+1}{10}$. Solving the equation $\Delta_{p_3} = 0$ is equivalent to $1700r^4 + 3560r^3 + 1872r^2 + 18r + 9 = 0$ and the solutions are $r_* \approx -1.0678, -1.0259, -0.000214 \pm 0.069518i$. This generates $\alpha_* \approx 0.516^L$, 0.504^L, 0.069^L, for which the diagonalization of $\mathbf{Q}\mathbf{G}_\ell^{-1}$ is not possible. These alphas are already very small for $L \approx 25$ and should not be used anyways.

Examples 2.7 and 2.8 show that for $M = 2, 3$ and a sufficient number of time steps L, the diagonalization of $\mathbf{Q}\mathbf{G}_\ell^{-1}$ is possible for every time step. We also identified for which parameters α the inner method's diagonalization is impossible. Even though Lemma 2.5 does not provide a direct answer, it is still helpful to compute finitely many values that should be avoided. Without it, a computational effort of checking for every α, ℓ and $\mathbf{Q}$ whether $\mathbf{Q}\mathbf{G}_\ell^{-1}$ is diagonalizable would have been necessary.

2.5. Parameter selection

Numerical error is introduced after each outer iteration in Algorithm 1, which has been observed in various works [53] and [50, Remark 3]. In Subsection 2.5.2 we comment on it more, but for now, let $\Delta\vec{\mathbf{u}}^{(k+1)}$ denote the error arising after performing the $(k+1)^{\text{th}}$ iteration for input $\vec{\mathbf{u}}^{(k)}$. The goal is to find α for each iteration that balances the perturbation errors and the convergence rate of the method. The distance between our $(k+1)^{\text{th}}$ perturbed iterate and the solution $\vec{\mathbf{u}}^*$ of system (2.15) can be expressed and bounded as

$$\|\vec{\mathbf{u}}^{(k+1)} + \Delta\vec{\mathbf{u}}^{(k+1)} - \vec{\mathbf{u}}^*\| \leq c_\alpha \|\vec{\mathbf{u}}^{(k)} - \vec{\mathbf{u}}^*\| + \|\Delta\vec{\mathbf{u}}^{(k+1)}\| \tag{2.34}$$

for some constant $c_\alpha > 0$. The constant c_α is the contraction rate of iterations (2.18) and satisfies the inequality $\|\vec{\mathbf{u}}^{(k+1)} - \vec{\mathbf{u}}^*\| \leq c_\alpha \|\vec{\mathbf{u}}^{(k)} - \vec{\mathbf{u}}^*\|$ on unperturbed values. The link between α and c_α is discussed in Section 2.5.1. Moreover, as $\alpha \to 0$, we expect $\|\Delta\vec{\mathbf{u}}^{(k+1)}\| \to \infty$ and $c_\alpha \to 0$, making the decision on which parameter α to choose for the next iteration highly relevant for the convergence of the method. Bounding the second α-dependent term $\|\Delta\vec{\mathbf{u}}^{(k+1)}\|$ on the right hand side of (2.34) and approximating c_α will bring us closer in finding suitable α_{k+1} to use for the computation of $\vec{\mathbf{u}}^{(k+1)}$.

2.5.1. Convergence

In order to formulate this problem more precisely, we will utilize the error analysis from [57], which we restate here for completeness.

Theorem 2.9
Assume that the matrix $\mathbf{A} \in \mathbb{C}^{N\times N}$ is diagonalizable as $\mathbf{A} = \mathbf{V}_A \mathbf{D}_A \mathbf{V}_A^{-1}$ and define $\mathbf{W} = \mathbf{I}_L \otimes \mathbf{V}_A$. Let $\mathbf{u}^*$ denote the solution of the composite collocation problem (2.16)*. Then for any $k \geq 1$ it holds*

$$\|\vec{\mathbf{u}}^{(k+1)} - \vec{\mathbf{u}}^*\|_{\mathbf{W},\infty} \leq \frac{\alpha}{1-\alpha} \|\vec{\mathbf{u}}^{(k)} - \vec{\mathbf{u}}^*\|_{\mathbf{W},\infty},$$

provided that the spectral radius of the collocation matrix (2.6) *satisfies $\rho(\mathbf{C}_{\text{coll}}) < 1$. Here, $\|\vec{\mathbf{u}}\|_{\mathbf{W},\infty} := \|\mathbf{W}\vec{\mathbf{u}}\|_\infty$.*

Proof: See [57, Theorem 2.1]. □

Using the above result, we can write

$$\begin{aligned}
\|\vec{\mathbf{u}}^{(k+1)} - \vec{\mathbf{u}}^*\|_\infty &\leq \|\mathbf{W}^{-1}\|_\infty \|\mathbf{W}(\vec{\mathbf{u}}^{(k+1)} - \vec{\mathbf{u}}^*)\|_\infty \\
&\leq \|\mathbf{W}^{-1}\| \frac{\alpha}{1-\alpha} \|\mathbf{W}(\vec{\mathbf{u}}^{(k)} - \vec{\mathbf{u}}^*)\|_\infty \\
&\leq \kappa_\infty(\mathbf{W}) \frac{\alpha}{1-\alpha} \|\vec{\mathbf{u}}^{(k)} - \vec{\mathbf{u}}^*\|_\infty.
\end{aligned}$$

This then leaves the question of how to bound the second error term $\|\Delta\vec{\mathbf{u}}^{(k+1)}\|$ in equation (2.34).

Remark 2.10
We derive a similar bound, specific to the collocation problem being the underlying integrator, independently of [57, Theorem 2.1]. Additionally, our analysis also covers the bound on the spectal radius, that was also published in [52, Theorem 2.1]. Despite that, it is included in the manuscript and confirms, in a different way, that for linear problems the convergence can be understood as $o(\alpha)$. The only difference is that we do not require the matrix $\mathbf{A}$ to be diagonalizable. See Appendix A for details.

2.5.2. Bounds for computation errors

In this subsection, we show that the relative error bound for the diagonalization of the preconditioner is $O(L(3\varepsilon + \tau)/\alpha)$, where ε denotes machine precision and τ is the relative error of inner system solves. Previously, there were several attempts to study the round-off error emerging from diagonalization of the preconditioner $\mathbf{C}_\alpha$. In [53], Wu mentions that the round-off error is of order $1/\alpha$ and refers to a technical report [68, Theorem 6 and 7]. However, the analysis is performed on triangular matrices $\mathbf{V}$, which emerge from a geometric mesh discretizing the time domain. A similar analysis has been done in [48], however, again performed on triangular matrices. In [55] the authors state that the round-off error is $O(2L\varepsilon/\alpha)$, referring to [69, Lemma 2.6 and 2.10], however, the matrix is again triangular. Another mentioned reference is [54, Theorem 2.1], where the relative error bound is $O\left(\varepsilon(2L+1)/\alpha^2\right)$, much closer to what we show, however, the bound is larger and presumes exact system solves via the LU-factorization with pivoting.

Because the following error bounds are very general and can be applied to any diagonalization-based algorithm, we state them in simplified notation. The three steps in the diagonalization computation (2.19) can be generally analyzed in this order: a matrix-vector multiplication $y = \hat{\mathbf{A}}x$, solving a system $\hat{\mathbf{B}}z = y$, where $\hat{\mathbf{B}}$ is a block diagonal matrix, followed by a matrix-vector multiplication $w = \hat{\mathbf{C}}z$, where in our particular case $\hat{\mathbf{C}}$ is the inverse of $\hat{\mathbf{A}}$. The errors in each step can be expressed as

$$y + \Delta y = (\hat{\mathbf{A}} + \Delta\hat{\mathbf{A}})x \tag{2.35a}$$

$$(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})(z + \Delta z) \approx y + \Delta y \tag{2.35b}$$

$$w + \Delta w = (\hat{\mathbf{C}} + \Delta\hat{\mathbf{C}})(z + \Delta z) \tag{2.35c}$$

with relative errors satisfying

$$\|\Delta\hat{\mathbf{A}}\| \le \varepsilon\|\hat{\mathbf{A}}\|,\ \|\Delta\hat{\mathbf{B}}\| \le \varepsilon\|\hat{\mathbf{B}}\|,\ \|\Delta\hat{\mathbf{C}}\| \le \varepsilon\|\hat{\mathbf{C}}\|, \tag{2.36}$$

for some $\varepsilon > 0$, representing machine precision.

Lemma 2.11
Let the system solve in (2.35b) *be inexact, in other words* $\|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})(z + \Delta z) - (y + \Delta y)\| \le \tau\|y + \Delta y\|$ *for some* $\tau > 0$ *and let* (2.36) *hold. Then the norm of the absolute error of w after the diagonalization process* (2.35) *satisfies*

$$\|\Delta w\| \le \frac{\|\hat{\mathbf{B}}^{-1}\|\,\|\hat{\mathbf{A}}\|\,\|\hat{\mathbf{C}}\|}{1 - \varepsilon\kappa(\hat{\mathbf{B}})}\left(2\varepsilon + \tau + \varepsilon\kappa(\hat{\mathbf{B}})\right)\|x\| + O(\tau\varepsilon + \varepsilon^2),$$

where the matrix norm is consistent, i.e. $\|\hat{\mathbf{A}}x\| \le \|\hat{\mathbf{A}}\|\,\|x\|$*, and the condition number is* $\kappa(\hat{\mathbf{A}}) := \|\hat{\mathbf{A}}\|\,\|\hat{\mathbf{A}}^{-1}\|$.

Proof: With $\hat{\mathbf{C}}z = w$ it holds

$$\Delta w = \hat{\mathbf{C}}\Delta z + \Delta\hat{\mathbf{C}}(z + \Delta z).$$

Using the triangle inequality, we have

$$\|\Delta w\| \le \|\hat{\mathbf{C}}\|\,\|\Delta z\| + \varepsilon\|\hat{\mathbf{C}}\|\,\|\Delta z + z\|. \tag{2.37}$$

Now we need to find a way to bound $\|\Delta z\|$ and $\|\Delta z + z\|$. Since the system solving is inexact, there exists a vector ξ such that $(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})(z + \Delta z) = y + \Delta y + \xi$ with $\|\xi\| \le \tau\|y + \Delta y\|$. Since $(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})\Delta z + \Delta\hat{\mathbf{B}}z = \Delta y + \xi$,

$$\Delta z = (\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}(\Delta y + \xi - \Delta\hat{\mathbf{B}}z)$$

which yields the bound

$$\begin{aligned}\|\Delta z\| &\leq \|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}\| \left(\|\Delta y\| + \|\xi\| + \varepsilon\|\hat{\mathbf{B}}\|\,\|z\|\right) \\ &\leq \|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}\| \left(\|\Delta y\| + \tau\|\Delta y + y\| + \varepsilon\kappa(\hat{\mathbf{B}})\|y\|\right), \end{aligned} \tag{2.38}$$

where the last inequality comes from the fact that $\|z\| \leq \|\hat{\mathbf{B}}^{-1}\|\|y\|$. Note that $\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}}$ is invertible for small perturbations if $\hat{\mathbf{B}}$ is invertible. On the other hand, we have

$$z + \Delta z = (\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}(y + \Delta y + \xi)$$

which gives

$$\begin{aligned}\|z + \Delta z\| &\leq \|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}\| \left(\|y + \Delta y\| + \|\xi\|\right) \\ &\leq (1 + \tau)\|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}\|\,\|y + \Delta y\|. \end{aligned} \tag{2.39}$$

Combining (2.38) and (2.39) with (2.37) yields

$$\|\Delta w\| \leq \|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}\|\,\|\hat{\mathbf{C}}\| \left(\|\Delta y\| + (\tau + \varepsilon + \varepsilon\tau)\|y + \Delta y\| + \varepsilon\kappa(\hat{\mathbf{B}})\|y\|\right). \tag{2.40}$$

Since $y = \hat{\mathbf{A}}x$ and $\|\Delta\hat{\mathbf{A}}\| \leq \varepsilon\|\hat{\mathbf{A}}\|$, we get the inequalities

$$\begin{aligned}\|y\| &\leq \|\hat{\mathbf{A}}\|\,\|x\|, \\ \|\Delta y\| &= \|\Delta\hat{\mathbf{A}}x\| \leq \varepsilon\|\hat{\mathbf{A}}\|\,\|x\|, \\ \|y + \Delta y\| &\leq (1 + \varepsilon)\|\hat{\mathbf{A}}\|\,\|x\|. \end{aligned}$$

Including these inequalities in (2.40) yields

$$\begin{aligned}\|\Delta w\| &\leq \|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}\|\,\|\hat{\mathbf{C}}\|\,\|\hat{\mathbf{A}}\| \left(\varepsilon + (\tau + \varepsilon + \varepsilon\tau)(1 + \varepsilon) + \varepsilon\kappa(\hat{\mathbf{B}})\right)\|x\| \\ &\leq \|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}\|\,\|\hat{\mathbf{C}}\|\,\|\hat{\mathbf{A}}\| \left(2\varepsilon + \tau + \varepsilon\kappa(\hat{\mathbf{B}})\right)\|x\| + O(\varepsilon\tau + \varepsilon^2). \end{aligned}$$

It remains to bound $\|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}\|$. From $(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1} = \hat{\mathbf{B}}^{-1}(\mathbf{I} + \hat{\mathbf{B}}^{-1}\Delta\hat{\mathbf{B}})^{-1}$ we have

$$\|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}\| \leq \|\hat{\mathbf{B}}^{-1}\|\|(\mathbf{I} + \hat{\mathbf{B}}^{-1}\Delta\hat{\mathbf{B}})^{-1}\| \leq \frac{\|\hat{\mathbf{B}}^{-1}\|}{1 - \|\hat{\mathbf{B}}^{-1}\Delta\hat{\mathbf{B}}\|}.$$

Since the function $1/(1 - x)$ is monotonically increasing, we get

$$\|(\hat{\mathbf{B}} + \Delta\hat{\mathbf{B}})^{-1}\| \leq \frac{\|\hat{\mathbf{B}}^{-1}\|}{1 - \varepsilon\kappa(\hat{\mathbf{B}})}$$

which completes the proof. □

Theorem 2.12
After one iteration of (2.18)*, for* $\alpha \in (0, 1)$*, under the assumption that the inner system solves in step* (2.19b) *satisfy* $\|\hat{\mathbf{B}}(z + \Delta z) - (y + \Delta y)\|_\infty \leq \tau\|y + \Delta y\|$*, the error of the* $(k + 1)$*th iterate can be bounded by*

$$\|\Delta\vec{\mathbf{u}}^{(k+1)}\|_\infty \leq \frac{\|\hat{\mathbf{B}}^{-1}\|_\infty}{1 - \varepsilon\kappa_\infty(\hat{\mathbf{B}})} \frac{L(2\varepsilon + \tau + \varepsilon\kappa_\infty(\hat{\mathbf{B}}))}{\alpha} \|\vec{\mathbf{r}}^{(k)}\|_\infty + O(\varepsilon\tau + \varepsilon^2), \tag{2.41}$$

where ε *is machine precision as in* (2.36)*,* $\mathbf{r}^{(k)} = (\mathbf{C}_\alpha - \mathbf{C})\vec{\mathbf{u}}^{(k)} + \vec{\mathbf{w}}$ *and* $\hat{\mathbf{B}} = \mathbf{D}_\alpha \otimes \mathbf{H} + \mathbf{I}_L \otimes \mathbf{C}_{\text{coll}}$*. Furthermore, for* $\ell = 1, \ldots, L$ *we have*

$$\|\Delta\mathbf{u}_\ell^{(k+1)}\|_\infty \leq \frac{\|\hat{\mathbf{B}}^{-1}\|_\infty}{1 - \varepsilon\kappa_\infty(\hat{\mathbf{B}})} \alpha^{\frac{-(\ell-1)}{L}} L(2\varepsilon + \tau + \varepsilon\kappa_\infty(\hat{\mathbf{B}}))\|\vec{\mathbf{r}}^{(k)}\|_\infty + O(\varepsilon\tau + \varepsilon^2). \tag{2.42}$$

Proof: Let relation (2.36) hold. Define $\hat{\mathbf{A}} := \mathbf{V}^{-1} \otimes \mathbf{I}_{MN}$, $\hat{\mathbf{B}} = \mathbf{D}_\alpha \otimes \mathbf{H} + \mathbf{I}_L \otimes \mathbf{C}_{\text{coll}}$ and $\hat{\mathbf{C}} := \mathbf{V} \otimes \mathbf{I}_{MN}$ where without loss of generality we can assume $\hat{\mathbf{A}} = \mathbf{V}^{-1}$ and $\hat{\mathbf{C}} = \mathbf{V}$. From Lemma 2.4 we see that

$$\|\hat{\mathbf{A}}\|_\infty \|\hat{\mathbf{C}}\|_\infty \leq \frac{1}{L}\|\mathbf{J}\|_\infty \|\mathbf{J}^{-1}\|_\infty \|\mathbf{F}\|_\infty \|\mathbf{F}^*\|_\infty \leq \frac{L}{\alpha}$$

since $\|\mathbf{F}\|_\infty \leq L$ and $\|\mathbf{F}^*\|_\infty \leq L$. The proof for (2.41) now follows directly from Lemma 2.11. Furthermore, if we define $\hat{\mathbf{A}} = \mathbf{V}^{-1}$ and $\hat{\mathbf{C}} = \frac{1}{L}\mathbf{F}$, then

$$\|\hat{\mathbf{A}}\|_\infty \|\hat{\mathbf{C}}\|_\infty \leq \frac{1}{L}\|\mathbf{J}^{-1}\|_\infty \|\mathbf{F}\|_\infty \|\mathbf{F}^*\| \leq L.$$

If now we define $\Delta w = (\mathbf{J}^{-1} \otimes \mathbf{I}_{MN})\Delta\vec{\mathbf{u}}^{(k+1)}$, we have

$$\|\mathbf{u}_\ell^{(k+1)}\|_\infty = \alpha^{\frac{-(l-1)}{L}} \|\Delta w_\ell\|_\infty \leq \alpha^{\frac{-(l-1)}{L}} \|\Delta w\|_\infty$$

and the proof for (2.42) again follows directly from Lemma 2.11. □

Remark 2.13
The IEEE standard guarantees that relation (2.36) *holds for the infinity norm and some* $\varepsilon = 2^{-p}$ *representing machine precision.*

The above theorem provides an idea of how the round-off error propagates across the vector $\vec{\mathbf{u}}^{(k+1)}$. The larger the number of the time steps, the larger the round-off error we can expect. This means that if the error between consecutive iterates is monitored, it is sufficient enough to do so just on the last time-step. Thus, we can avoid computing a residual and use the difference between two consecutive iterates at the last time-step as a termination criterion.

2.5.3. Choosing the $(\alpha_k)_{k\in\mathbb{N}}$ sequence

While Theorem 2.9 confirms that a small α yields fast convergence, Theorem 2.12 indicates that a smaller α leads to a more significant numerical error per iteration. Suitable choices of α should balance both aspects. To achieve that, define $m_0 \in \mathbb{R}$ so that $\|\vec{\mathbf{u}}^{(0)} - \vec{\mathbf{u}}_*\|_\infty \approx m_0$. Using the results of Theorems 2.9 and 2.12, we can approximate

$$\|\vec{\mathbf{u}}^{(1)} - \vec{\mathbf{u}}^*\|_\infty \lessapprox \alpha m_0$$

for $\alpha \approx 0$, and

$$\|\Delta\vec{\mathbf{u}}^{(1)}\|_\infty \lessapprox \frac{\|\hat{\mathbf{B}}^{-1}\|_\infty}{1 - \varepsilon\kappa_\infty(\hat{\mathbf{B}})} \frac{L(2\varepsilon + \tau + \varepsilon\kappa_\infty(\hat{\mathbf{B}}))}{\alpha} \|\vec{\mathbf{r}}^{(0)}\|_\infty \lessapprox \frac{L}{\alpha}(3\varepsilon + \tau)\|\vec{\mathbf{r}}^{(0)}\|_\infty,$$

where, $\vec{\mathbf{r}}^{(0)} = (\mathbf{C}_\alpha - \mathbf{C})\vec{\mathbf{u}}^{(0)} + \vec{\mathbf{w}}$. The first estimate originates from the fact that $\kappa_\infty(\mathbf{W})\alpha/(1-\alpha) = o(\alpha)$. We will later verify that the convergence rate is unaffected by the simplification (see Figure 2.3). This assumption does not change the asymptotics for small values of α, but it allows us to draw more conclusions about the trade-off of errors, especially around $\alpha \approx 0$. The sharpness of the convergence bound from Theorem 2.9 is not very relevant, since the convergence factor is not practical to use it in that form. Combining these estimates, we get

$$\|\vec{\mathbf{u}}^{(1)} + \Delta\vec{\mathbf{u}}^{(1)} - \vec{\mathbf{u}}^*\|_\infty \lessapprox \alpha m_0 + \frac{L}{\alpha}(3\varepsilon + \tau)\|\vec{\mathbf{r}}^{(0)}\|_\infty. \tag{2.43}$$

The aim is to find α such that the right-hand side of (2.43) is minimized for given L, ε, and τ. Since $(\mathbf{C}_\alpha - \mathbf{C})\vec{\mathbf{u}}^{(0)}$ is a block-vector with just $-\alpha\mathbf{H}\mathbf{u}_L^{(0)}$ being a nonzero block, we can treat the $\|\vec{\mathbf{r}}^{(0)}\|_\infty$ term in the minimization process roughly as $\|\vec{\mathbf{w}}\|_\infty$ since this part is more relevant. Defining $\gamma := L(3\varepsilon + \tau)\|\mathbf{w}\|_\infty$ for simplicity, we can state the problem as: find $\alpha \in (0, 1)$ such that $m_1 = \alpha m_0 + \frac{\gamma}{\alpha}$ is minimized. The solution is visible from the fact that $m_0\alpha^2 - m_1\alpha + \gamma = 0$ is a parabola with a discriminant $\Delta = m_1^2 - 4m_0\gamma$ and that the smallest value for m_1 so that $\Delta \geq 0$ is when $\Delta = 0$. This yields $m_1 = 2\sqrt{m_0\gamma}$ and a unique root $\alpha = \sqrt{\frac{\gamma}{m_0}}$. To interpret this result, we expect the error after one iteration to be around m_1 if the parameter $\alpha_1 = \sqrt{\frac{\gamma}{m_0}}$ is used.

Recursively using (2.43), we can approximate

$$\|\vec{\mathbf{u}}^{(k+1)} + \Delta\vec{\mathbf{u}}^{(k+1)} - \vec{\mathbf{u}}^*\| \lessapprox \alpha m_k + \frac{\gamma}{\alpha}.$$

Solving the same minimization problem now for $m_{k+1} = \alpha m_k + \frac{\gamma}{\alpha}$ yields $m_{k+1} = 2\sqrt{m_k\gamma}$ and $\alpha = \alpha_{k+1} = \sqrt{\frac{\gamma}{m_k}}$.

If $4\gamma \leq m_k$ holds, then the approximations of errors m_k are decreasing. Consequently, the sequence of $(\alpha_k)_{k\in\mathbb{N}}$ is increasing since $m_{k+1} = 2\sqrt{m_k\gamma}$. Because $(\alpha_k)_{k\in\mathbb{N}}$ is additionally bounded from above with $\sqrt{\gamma/m_k} \leq 1/2$, we see that it is a convergent sequence with a limit in $(0, 1/2]$. Telescoping the recursion for m_k we get $m_k = (4\gamma)^{1-2^{-k}} m_0^{2^{-k}}$, $k \geq 1$, showing that the sequence is asymptotically bounded by 4γ. The convergence speed seems to be slower as the method iterates, but this does not have to be the case in practice. A remedy to this is to monitor the error of consecutive iterates and because of Theorem 2.12 we can do so just for the last time-step. If this error is far lower than m_k, it is a good sign the convergence is much better than anticipated. Usually, monitoring just the error of consecutive iterates does not detect stagnation, but sees it as convergence. However, the combination of both can provide solid information. This discussion can be summarized in the following algorithm for the stopping criterion.

Algorithm 2 A stopping criterion combining approximations of worst-case convergence and an error of consecutive iterates, with a given m_0. The stopping tolerance is ζ.

$\gamma := L(3\varepsilon + \tau)\|\vec{\mathbf{w}}\|_\infty$
$k = 0$
while $m_k > \zeta$ **do**
  $m_{k+1} = 2\sqrt{m_k\gamma}$
  $\alpha_{k+1} = \sqrt{\gamma/m_k}$
  $\vec{\mathbf{u}}^{(k+1)} = \mathbf{iterate}(\vec{\mathbf{u}}^{(k)}, m_{k+1}, \alpha_{k+1})$ # perform one iteration as in Algorithm 1
  if $\|\mathbf{u}_L^{(k+1)} - \mathbf{u}_L^{(k)}\|_\infty \leq \zeta$ **then**
    return $\vec{\mathbf{u}}^{(k+1)}$
  $k = k + 1$

The only problem remains how to approximate m_0. Since it is an approximation of $\|\vec{\mathbf{u}}^{(0)} - \vec{\mathbf{u}}^*\|_\infty$ it is convenient to define a vector filled with initial conditions $\vec{\mathbf{u}}_0$ as the initial iterate $\vec{\mathbf{u}}^{(0)}$. Then, from the integral formulation, we have $m_0 \approx \max_{[0,T]} |u(t) - u(0)| \leq TL_u$ where L_u is the local Lipschitz constant of the solution on $[0, T]$. If this is inconvenient, one can also use that $m_0 \approx \|\vec{\mathbf{u}}^{(0)} - \vec{\mathbf{u}}_*\|_\infty \leq T \max_{[0,T]} \|Au(t) + b(t)\|_\infty \leq T(\|\mathbf{A}\|_\infty M_u + M_b)$ where M_u denotes the approximation of a maximum of the solution and M_b is the maximum for the function b. The best approximate convergence curves are when m_0 is a tight approximation. One can say that the errors are around values m_k as much as m_0 is

around $\|\vec{\mathbf{u}}^{(0)} - \vec{\mathbf{u}}_*\|_\infty$. In practice, we observed that if m_0 is an upper bound, then all m_k are as well an upper bound.

This concludes the discussion about the $(\alpha_k)_{k\in\mathbb{N}}$ sequence. In this section, we have proposed an $(\alpha_k)_{k\in\mathbb{N}}$ sequence that is supposed to minimize the number of outer iterations, balancing the unwanted numerical errors and the convergence rate.

Now we will highlight the importance of α-adaptivity with numerical tests. The first test equation is the heat equation, governed by

$$u_t = \Delta u + \sin(2\pi x)\sin(2\pi y)(8\pi^2\cos(t) - \sin(t)), \quad (t, x, y) \in [\pi, \pi + T] \times [0, 1]^2, \tag{2.44}$$

with the exact solution is $u(t, x, y) = \sin(t)\sin(2\pi x)\sin(2\pi y)$. This equation has periodic boundary conditions which were used to form the discrete periodic Laplacian with central differences bringing the equation into the generic form (2.9). The second equation is the advection equation defined as

$$u_t + u_x + u_y = 0, \quad (t, x, y) \in [0, T] \times [0, 1]^2, \tag{2.45}$$

with exact solution $u(t, x, y) = \sin(2\pi x - 2\pi t)\sin(2\pi y - 2\pi t)$. Here, we again have periodicity on the boundaries which was used to form an upwind scheme [5]. This yields a sparse circulant spatial discretization matrix since the approximation of the solution in the end interval point is treated as an approximation in the left interval point of the spatial discretization.

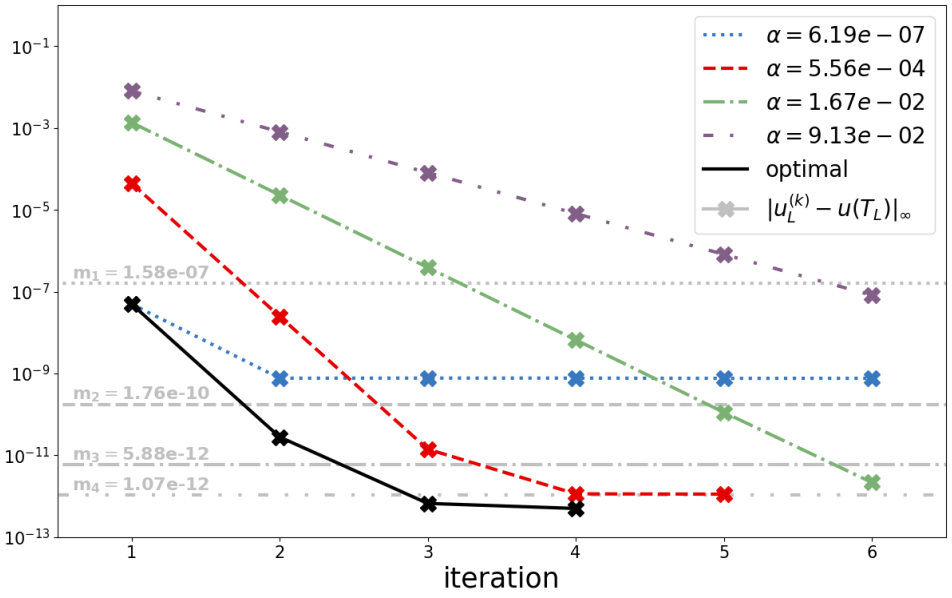


Figure 2.2.: Adaptive strategy vs. convergence for fixed α_k from the sequence. The y-axis represents the error in $\log_{10}$ scale whereas the vertical lines represent the m_k sequence starting with $m_0 = 10\Delta T$. The solid line is the convergence history with the sequence of α_k given as $(6.19 \times 10^{-7}, 5.56 \times 10^{-4}, 1.67 \times 10^{-2}, 9.13 \times 10^{-2})$.

Figure 2.2 compares the convergence for the linear advection equation when the adaptive $(\alpha_k)_{k\in\mathbb{N}}$ sequence generated on runtime is used, to benchmarks where α_k is a fixed parameter, originating from that same generated sequence. For larger values of α, we can see a slow convergence speed that can reach better accuracy, while for smaller α values, the convergence is extremely steep but short living. Using

the adaptive $(\alpha_k)_{k\in\mathbb{N}}$ requires fewer total iterations to recover a small error tolerance compared to a fixed α. The numbers of discretization points in space and time are chosen so that the error with respect to the exact solution (infinity norm) is below $\zeta = 10^{-12}$, without over-resolving in space or time. The parameters chosen are $T = 0.0128$, $L = 64$, $M = 3$, and a 5^{th}-order upwind scheme in space with $N = 700$ spatial points. The scheme is formed as described in [5]. The benchmarks were performed in parallel across time steps using 64 cores. The inner solver is GMRES (without a preconditioner) with a relative tolerance $\tau = 10^{-15}$.

Counting the iterations seems promising, however, we must also look at the runtimes. When parallelizing across time steps, the parallel wall clock times accompanying the runs presented in Figure 2.2 are summarized in Table 2.1. We can now indeed conclude that using α-adaptivity reduces the number of iterations and computation time by $30 - 40\%$.

α	optimal	6.19×10^{-7}	5.56×10^{-4}	1.67×10^{-2}	9.13×10^{-2}
parallel wall clock time [s]	25.34	36.15	31.72	38.22	38.33

Table 2.1.: Parallel wall clock times for parallelizing the advection equation across $L = 64$ time steps. The data exactly accompanies Figure 2.2.

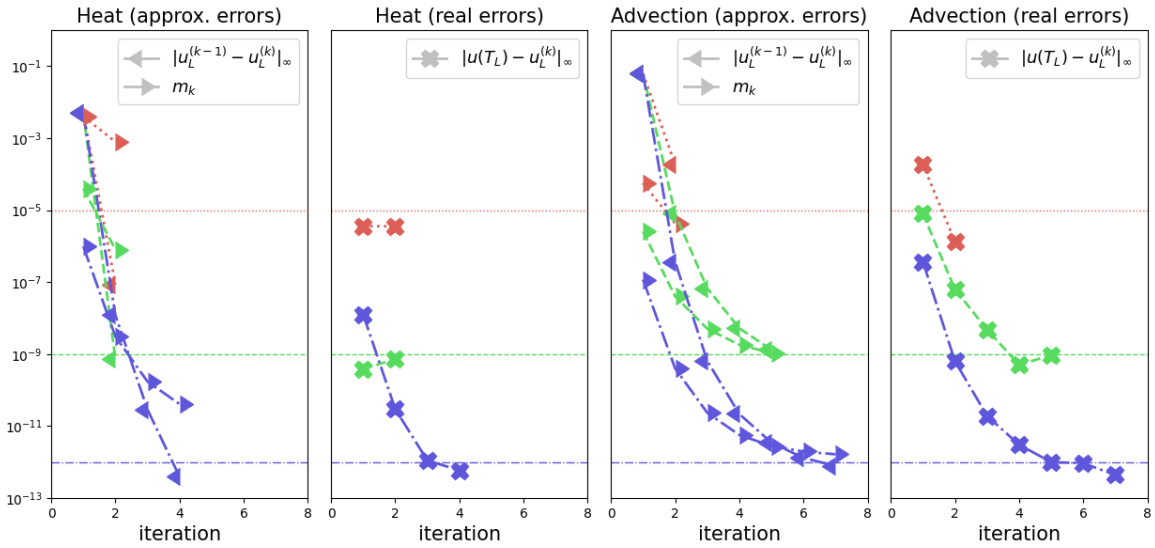


Figure 2.3.: Convergence with the adaptive strategy for different thresholds. The y-axis represents the error in $\log_{10}$ scale, and vertical lines represent the thresholds for the stopping criteria: red for reaching thresholds under 10^{-5}, green for 10^{-9} and blue for 10^{-12}. 'approx. errors' graphs contain the information available on runtime, which is: the errors of consecutive iterates in the last time-step (marker pointing left) and the approximations of the upper bound for the error in each iteration (marker pointing right), the m_k values starting with $m_0 = \Delta T$. These values are generated in Algorithm 2 alongside the corresponding α_k. 'real errors' graphs show corresponding errors to the exact solution, which is generally unavailable at runtime.

Understanding the convergence behavior allows us to predict the number of iterations for a given threshold roughly. In order to examine the convergence of the method, we test it when solving the same initial value problem with the same domain for three thresholds $\zeta = 10^{-5}, 10^{-9}, 10^{-12}$. One challenge for an

actual application is when to stop the iterations since the actual error is not available. As discussed in Section 2.5.2, a valid candidate is comparing successive iterates at the last time-step, cf. Lemma 2.11. To check the impact of this choice, Figure 2.3 shows the convergence behavior for the two test problems with the adaptive-α strategy, both for the actual errors and the difference between successive iterates. The discretization parameters for each threshold are chosen accordingly for a fixed T (see Table 2.2 for details). The convergence study shows that the m_k values are indeed following the actual errors, unavailable at runtime, very well. However, values m_k should be treated in combination with the consecutive iterates as in Algorithm 2 since they are an overestimate.

	Heat, $T = 0.1$			Advection, $T = 10^{-2}$		
tol. to reach ζ	10^{-5}	10^{-9}	10^{-12}	10^{-5}	10^{-9}	10^{-12}
no. of spatial points N	450	400	300	350	350	600
order in space κ	2	4	6	2	4	5
no. of collocation nodes M	1	2	3	2	2	3
no. of time steps L	32	32	16	8	16	32
linear solver tolerance τ	10^{-6}	10^{-10}	10^{-13}	10^{-8}	10^{-11}	10^{-14}

Table 2.2.: Parameter choice for solving the heat and advection equation in order to reach an error $\|\mathbf{u}(T) - \mathbf{u}_L\|_\infty < \zeta$ when solving with a standard sequential approach. κ denotes the discretization order in space, where upwind was chosen for the advection equation and centered differences for the heat equation.

CHAPTER 3

Extensions to nonlinear equations

Let us now consider an initial value problem of the form

$$u_t = \mathbf{A}u + f(u) + b(t), \quad u(0) = u_0, \tag{3.1}$$

where $f : \mathbb{R}^N \to \mathbb{R}^N$. The matrix $\mathbf{A}$ is a linear term that can be treated implicitly and might come from discretization or linearization. The part $f(u)$ does not strictly have to be nonlinear in u, however, it may contain such a part. This term will be treated explicitly in the iterative process proposed in Section 3.2.

Imitating the sequential time stepping process (2.13)–(2.14), the collocation equations (3.1) have the form

$$\mathbf{C}_{\text{coll}}\mathbf{u}_1 - \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}(\mathbf{u}_1) = \mathbf{u}_0 + \mathbf{v}_1 \tag{3.2a}$$

$$\mathbf{C}_{\text{coll}}\mathbf{u}_\ell - \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}(\mathbf{u}_\ell) = \mathbf{H}\mathbf{u}_{\ell-1} + \mathbf{v}_\ell, \quad \ell = 2, \ldots, L. \tag{3.2b}$$

In order to use the diagonalization trick for the preconditioner $\mathbf{C}_\alpha$ as in (2.19), the preconditioning matrix has to have constant block entries.

One way to build $\mathbf{C}_\alpha$ for this problem is to utilize inexact Newton's iterative method. Let $\mathbf{J}_\ell^{(k)}$ denote the exact or an approximate Jacobian for the function $\mathbf{f}$ computed in the point $\mathbf{u}_\ell^{(k)}$. The approximation, for example, can be a Jacobian computed just in the last internal stage $\mathbf{H}\mathbf{u}_\ell^{(k)}$. Now, for each iteration k, let $\bar{\mathbf{J}}^{(k)}$ denote the average Jacobian across all time steps as:

$$\bar{\mathbf{J}}^{(k)} = \frac{1}{L}\sum_{\ell=1}^{L} \mathbf{J}_\ell^{(k)} \in \mathbb{R}^{MN}. \tag{3.3}$$

An approach examined in a lot of algorithms [56, 52] is to use the preconditioner $\mathbf{C}_\alpha$ as

$$\mathbf{C}_\alpha = \mathbf{I}_L \otimes \left(\mathbf{C}_{\text{coll}} - \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\bar{\mathbf{J}}^{(k)}\right) + \mathbf{E}_\alpha \otimes \mathbf{H}. \tag{3.4}$$

Then, the diagonal is composed of constant block matrices, making the diagonalization using Lemma 2.4 possible. The main drawback is that the Jacobian has to be recomputed in every iteration. This means that, if implemented in parallel, the cores have to perform an `MPI_Allreduce` communication pattern to update the next Jacobian approximation $\bar{\mathbf{J}}^{(k)}$.

We propose to treat the nonlinear part in (3.2) explicitly. This approach avoids the Jacobian updates altogether. Preconditioning the equations (3.2) with the same preconditioner $\mathbf{C}_\alpha$, defined as for the linear case in (2.17), yields iterations

$$\mathbf{C}_\alpha \vec{\mathbf{u}}^{(k+1)} = \vec{\mathbf{b}}^{(k)}, \tag{3.5}$$

where

$$\vec{\mathbf{b}}^{(k)} = \vec{\mathbf{w}} + \begin{bmatrix} \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}(\mathbf{u}_1^{(k)}) - \alpha \mathbf{H}\mathbf{u}_L^{(k)} \\ \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}(\mathbf{u}_2^{(k)}) \\ \vdots \\ \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}(\mathbf{u}_L^{(k)}) \end{bmatrix},$$

and $\vec{\mathbf{w}}$ is the same as before, defined in equation (2.16). In practice, this approach needs more iterations, meaning that the convergence rate is sacrificed in order to avoid dealing with the Jacobians. However, it is suitable for problems where the Jacobian matrix is too tedious to compute, such as the collision term in the Boltzmann equation.

To analyze the convergence of both schemes, the inexact Newton's method approach and the implicit-explicit approach, we first need to examine the underlying single time-step iterations of solving (3.2) that emerge from the global iterations (2.18) when using preconditioners $\mathbf{C}_\alpha$ from (2.17) and (3.4). We write down the sequential computation of equations (3.2) in an Implicit-Explicit (IMEX) iterative way

$$\mathbf{C}_{\text{coll}}\mathbf{u}_1^{(k+1)} - \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}_{\text{I}}\left(\mathbf{u}_1^{(k+1)}\right) = \mathbf{u}_0 + \mathbf{v}_1 + \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}_{\text{E}}\left(\mathbf{u}_1^{(k)}\right) \tag{3.6a}$$

$$\mathbf{C}_{\text{coll}}\mathbf{u}_\ell^{(k+1)} - \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}_{\text{I}}\left(\mathbf{u}_\ell^{(k+1)}\right) = \mathbf{H}\mathbf{u}_{\ell-1} + \mathbf{v}_\ell + {}_{\ell-1} + \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}_{\text{E}}\left(\mathbf{u}_\ell^{(k)}\right), \quad \ell = 2, \ldots, L, \tag{3.6b}$$

where we split the function f into $f = f_{\text{I}} + f_{\text{E}}$ to indicate that f_{I} is treated implicitly and f_{E} explicitly. Here, we also assume that $\mathbf{H}\mathbf{u}_{\ell-1}$ is constructed from the previously converged time step. The reason behind this notation is that this splitting covers both approaches: in the inexact Newton's method, we have

$$\mathbf{f}_{\text{I}}(\mathbf{u}) := \left(\mathbf{A} + \bar{\mathbf{J}}^{(k)}\right)\mathbf{u}, \quad \mathbf{f}_{\text{E}}(\mathbf{u}) := \mathbf{f}(\mathbf{u}) - \bar{\mathbf{J}}^{(k)}\mathbf{u},$$

whereas for treating the nonlinear part f explicitly would yield a splitting

$$\mathbf{f}_{\text{I}}(\mathbf{u}) := \mathbf{A}\mathbf{u}, \quad \mathbf{f}_{\text{E}}(\mathbf{u}) := \mathbf{f}(\mathbf{u}).$$

The iterative form (3.6) is motivated by semi-implicit spectral deferred corrections (SISDC), where the f_{I} part is usually used to precondition the stiff term and the f_{E} the non-stiff term [70]. Another interpretation is splitting f into parts corresponding to slow-wave and fast-wave formats as done in [71].

Splitting f into implicit and explicit parts allows us not only to analyze both approaches, but any kind of splitting, generalizing the analysis even further. If the fixed point iterations (3.6) converge, then the solution also satisfies equations (3.2). Our next task is to determine under which conditions are these fixed point iterations on a single time step a contraction.

3.1. The collocation problem iterations: a single time step

For simplicity, we rewrite the IMEX iterations (3.6a) as

$$\mathbf{u}^{(k+1)} - \Delta T \mathbf{Q}\mathbf{f}_{\mathrm{I}}(\mathbf{u}^{(k+1)}) = \mathbf{u}_0 + \Delta T \mathbf{Q}\mathbf{f}_{\mathrm{E}}(\mathbf{u}^{(k)}). \tag{3.7}$$

These iterations already presume to have an exact initial condition $\mathbf{u}_0$, compared to the schemes (3.5) and (3.4) where the initial conditions are approximated as $\mathbf{H}\mathbf{u}_\ell^{(k)}$, however, the convergence in the purely linear case is fast, which makes this approach look promising for equations where the Lipschitz constant of f_{E} in (3.7) is not 'too large'.

Remark 3.1
The collocation problem can be solved iteratively in many ways, one of which includes choosing additional preconditioners $\mathbf{Q}_{\mathrm{I}}$ *and* $\mathbf{Q}_{\mathrm{E}}$ *for the implicit and the explicit parts* f_{I} *and* f_{E} *in* (3.7) *as*

$$\mathbf{u}^{(k+1)} - \Delta T \mathbf{Q}_{\mathrm{I}}\mathbf{f}_{\mathrm{I}}(\mathbf{u}^{(k+1)}) - \Delta T \mathbf{Q}_{\mathrm{E}}\mathbf{f}_{\mathrm{E}}(\mathbf{u}^{(k+1)}) = \mathbf{u}_0 + \Delta T(\mathbf{Q} - \mathbf{Q}_{\mathrm{I}})\mathbf{f}_{\mathrm{I}}(\mathbf{u}^{(k)}) + \Delta T(\mathbf{Q} - \mathbf{Q}_{\mathrm{E}})\mathbf{f}_{\mathrm{E}}(\mathbf{u}^{(k)}).$$

These fixed point iterations form the most general form of a method called Spectral Deferred Corrections (SDC), originally presented in [35]. The link to the matrix form presented here is described in [72, Chapter 3]. The usual choices for the preconditioners are lower triangular matrices originating from Euler stepping schemes, or the LU-trick, allowing the systems on the left-hand side to be solved via forward substitutions. However, because of how the inner systems are solved via the additional two substitutions with the help of the upper triangular matrix $\mathbf{G}_\ell$ *(see* (2.23a)*), the structures of* $\mathbf{Q}_{\mathrm{I}}\mathbf{G}_\ell^{-1}$ *and* $\mathbf{Q}_{\mathrm{E}}\mathbf{G}_\ell^{-1}$ *would be lost, except in the case, when they are also upper triangular matrices. This makes the whole idea of using structured matrices as preconditioners, in our case, unfeasable. Formally, our idea to treat the nonlinear part explicitly as in* (3.5) *can be expressed as using* $\mathbf{Q}_{\mathrm{I}} = \mathbf{Q}$ *and* $\mathbf{Q}_{\mathrm{E}} = \mathbf{0}$.

In order to show that iterations (3.7) converge, we need to examine under which circumstances are they a contraction. This analysis is usually done for the Dahlquist's test equation $u_t = (\lambda_{\mathrm{E}}+\lambda_{\mathrm{I}})u$[1]. Let us now define our function f as $f_{\mathrm{I}}(u) := \lambda_{\mathrm{I}}u$ and $f_{\mathrm{E}}(u) := \lambda_{\mathrm{E}}u$. The iterations now become

$$(\mathbf{I} - \Delta T \lambda_{\mathrm{I}}\mathbf{Q})\mathbf{u}^{(k+1)} = \mathbf{u}_0 + \Delta T \lambda_{\mathrm{E}}\mathbf{Q}\mathbf{u}^{(k)}, \tag{3.8}$$

and we are interested when the iterations $\mathbf{u}^{(k)}$ converge to a fixed point solution $\mathbf{u}^*$ of

$$(\mathbf{I} - \Delta T(\lambda_{\mathrm{I}} + \lambda_{\mathrm{E}})\mathbf{Q})\mathbf{u}^* = \mathbf{u}_0. \tag{3.9}$$

Let $\mathbf{e}^{(k)} = \mathbf{u}^{(k)} - \mathbf{u}^*$ be the error vector. The error behaves as

$$(\mathbf{I} - \Delta T \lambda_{\mathrm{I}}\mathbf{Q})\mathbf{e}^{(k+1)} = \Delta T \lambda_{\mathrm{E}}\mathbf{Q}\mathbf{e}^{(k)},$$

from where

$$\mathbf{e}^{(k+1)} = \Delta T \lambda_{\mathrm{E}}(\mathbf{I} - \Delta T \lambda_{\mathrm{I}}\mathbf{Q})^{-1}\mathbf{Q}\mathbf{e}^{(k)}. \tag{3.10}$$

By examining the factors on the right, we can easily see that when $\Delta T \lambda_{\mathrm{E}}$ is small and $(\mathbf{I} - \Delta T \lambda_{\mathrm{I}}\mathbf{Q})^{-1}$ behaves in a damping way, a contraction is possible. First, to rigorously prove this, we need a preliminary result.

[1] For this analysis, one assumes that the linear system is diagonalizable where λ_{I} represents an eigenvalue.

Lemma 3.2
Let the matrix $\mathbf{Q}$ *be built from quadrature nodes* $0 \leq t_1 \leq \cdots \leq t_M \leq 1$ *originating from the Gauss–Lobatto quadrature or any other quadrature of degree* $2M - 2$ *or higher. Then the following inequality holds:*

$$t_M \leq \|\mathbf{Q}\|_\infty \leq \sqrt{t_M}. \tag{3.11}$$

Proof: See Appendix B. □

From now on, we will assume that the matrix $\mathbf{Q}$ satisfies the assumptions from Lemma 3.2. The following lemma guarantees under which conditions the iterations (3.8) are a contraction.

Lemma 3.3
Let the matrix $\mathbf{Q}$ *satisfy the conditions in Lemma 3.2. When* $\Delta T \sqrt{t_M}(|\lambda_\mathrm{E}| + |\lambda_\mathrm{I}|) < 1$ *holds, the iterations* (3.8) *are a contraction. Furthermore,* $\|\mathbf{e}^{(k+1)}\|_\infty \leq \eta \|\mathbf{e}^{(k)}\|_\infty$*, where*

$$\eta \leq \frac{\Delta T |\lambda_\mathrm{E}| \sqrt{t_M}}{1 - \Delta T |\lambda_\mathrm{I}| \sqrt{t_M}} < 1.$$

Proof: From (3.10) we have

$$\|\mathbf{e}^{(k+1)}\|_\infty \leq \Delta T |\lambda_\mathrm{E}| \, \|(\mathbf{I} - \Delta T \lambda_\mathrm{I} \mathbf{Q})^{-1}\|_\infty \|\mathbf{Q}\|_\infty \|\mathbf{e}^k\|_\infty.$$

Because of the assumption $\Delta T \sqrt{t_M}(|\lambda_\mathrm{E}| + |\lambda_\mathrm{I}|) < 1$ we know that $\|\Delta T \lambda_\mathrm{I} \mathbf{Q}\|_\infty \leq \Delta T \sqrt{t_M} |\lambda_\mathrm{I}| < 1$ holds. This means that the inverse $(\mathbf{I} - \Delta T \lambda_\mathrm{I} \mathbf{Q})^{-1}$ exists and

$$\|(\mathbf{I} - \Delta T \lambda_\mathrm{I} \mathbf{Q})^{-1}\|_\infty \leq \frac{1}{1 - \Delta T |\lambda_\mathrm{I}| \|\mathbf{Q}\|_\infty} \leq \frac{1}{1 - \Delta T |\lambda_\mathrm{I}| \sqrt{t_M}}.$$

Thus,

$$\|\mathbf{e}^{(k+1)}\|_\infty \leq \frac{\Delta T |\lambda_\mathrm{E}| \sqrt{t_M}}{1 - \Delta T |\lambda_\mathrm{I}| \sqrt{t_M}} \|\mathbf{e}^{(k)}\|_\infty.$$

Here we can see that $\Delta T \sqrt{t_M}(|\lambda_\mathrm{E}| + |\lambda_\mathrm{I}|) < 1$, concluding that $\eta < 1$ which completes the proof. □

The condition $\Delta T \sqrt{t_M}(|\lambda_\mathrm{E}| + |\lambda_\mathrm{I}|) < 1$ in Lemma 3.3 is quite limiting. The following lemma explains contractions for large ΔT.

Lemma 3.4
Let the matrix $\mathbf{Q}$ *be nonsingular. When* $|\lambda_\mathrm{E}| < |\lambda_\mathrm{I}|$*, the iterations* (3.8) *are a contraction in the stiff limit, i.e.,* $\Delta T \to \infty$*. Furthermore, for any norm* $\|\cdot\|$*, it holds that* $\|\mathbf{e}^{(k+1)}\| \leq \eta \|\mathbf{e}^{(k)}\|$*, where*

$$\eta \leq \frac{|\lambda_\mathrm{E}|}{|\lambda_\mathrm{I}|} < 1.$$

Proof: From (3.10) we have

$$\|\mathbf{e}^{(k+1)}\| = \Delta T |\lambda_\mathrm{E}| \, \|(\mathbf{I} - \Delta T \lambda_\mathrm{I} \mathbf{Q})^{-1} \mathbf{Q}\| \, \|\mathbf{e}^k\|.$$

Inserting ΔT into the norm, we get

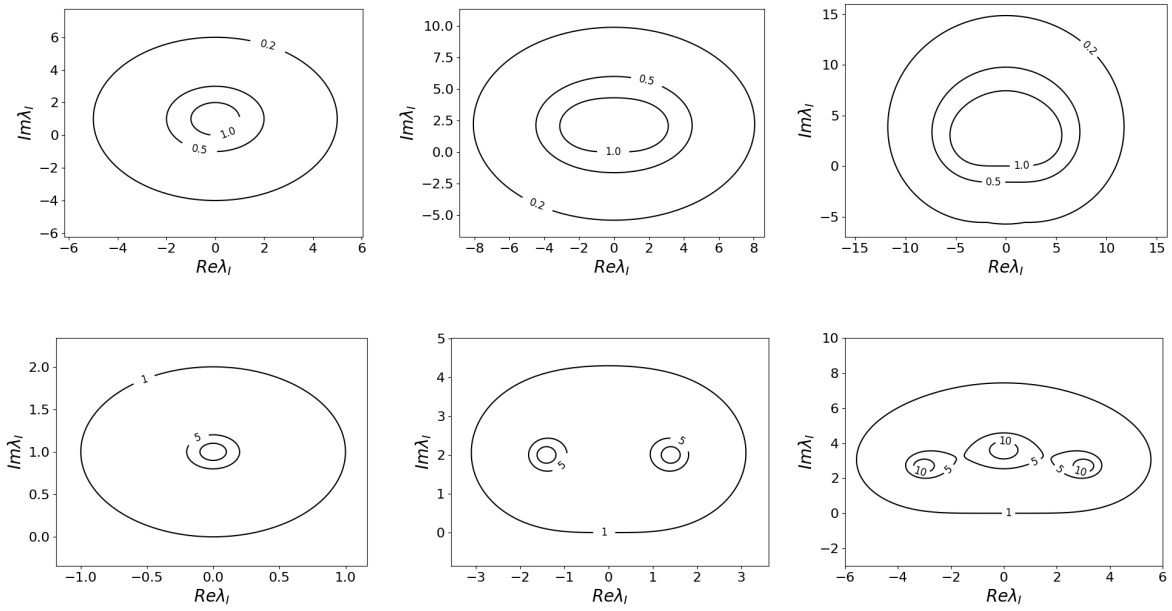
$$\|\mathbf{e}^{(k+1)}\| = |\lambda_\mathrm{E}| \left\| \left(\frac{1}{\Delta T} \mathbf{I} - \lambda_\mathrm{I} \mathbf{Q} \right)^{-1} \mathbf{Q} \right\| \|\mathbf{e}^k\|.$$

Now, for $\Delta T \to \infty$, the norm of the iteration matrix becomes

$$|\lambda_{\mathrm{E}}| \left\| \left(\frac{1}{\Delta T}\mathbf{I} - \lambda_{\mathrm{I}}\mathbf{Q} \right)^{-1} \mathbf{Q} \right\| \to \frac{|\lambda_{\mathrm{E}}|}{|\lambda_{\mathrm{I}}|} < 1,$$

which concludes the proof. □

Remark 3.5
The condition $\Delta T \sqrt{t_M}(|\lambda_{\mathrm{E}}| + |\lambda_{\mathrm{I}}|) < 1$ *in Lemma 3.3 guarantees* $\eta < 1$*, a contraction of a single time-step method* (3.8)*, however, the condition seems to be too strict than what is observed in practice. On the other hand, Lemma 3.4 ensures contractions for very large time steps, and it is difficult to show what happens for intermediate step sizes theoretically. Figure 3.1 illustrates different values of* $\left\|(\mathbf{I} - \lambda_{\mathrm{I}}\mathbf{Q})^{-1}\mathbf{Q}\right\|_\infty$*. For a given* λ_{E}*, we can gain more information whether the iterations are a contraction by validating when* $\left\|(\mathbf{I} - \lambda_{\mathrm{I}}\mathbf{Q})^{-1}\mathbf{Q}\right\|_\infty < 1/|\lambda_{\mathrm{E}}|$ *is true.*



(a) $M = 1$ (b) $M = 2$ (c) $M = 3$

Figure 3.1.: Contours of $\left\|(\mathbf{I} - \lambda_{\mathrm{I}}\mathbf{Q})^{-1}\mathbf{Q}\right\|_\infty$ for the Radau-Right quadrature. Upper plots are contours for values $0.2, 0.5, 1$ and lower plots are contours with values $1, 5, 10$.

3.2. The composite collocation problem iterations

We want to extend the IMEX iterations for a single time step (3.6) in a structure that can exploit parallelism across time steps. We again utilize the block α-circulant matrix and define composite collocation problem iterations as

$$\begin{bmatrix} \mathbf{C}^{\mathrm{I}}_{\mathrm{coll}} & & & -\alpha\mathbf{H}_M \\ -\mathbf{H}_M & \mathbf{C}^{\mathrm{I}}_{\mathrm{coll}} & & \\ & \ddots & \ddots & \\ & & -\mathbf{H}_M & \mathbf{C}^{\mathrm{I}}_{\mathrm{coll}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_L \end{bmatrix}^{(k+1)} = \begin{bmatrix} \mathbf{u}_0 + \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}_{\mathrm{E}}(\mathbf{u}_1^{(k)}) - \alpha\mathbf{H}_M\mathbf{u}_L^{(k)} \\ \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}_{\mathrm{E}}(\mathbf{u}_2^{(k)}) \\ \vdots \\ \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}_{\mathrm{E}}(\mathbf{u}_L^{(k)}) \end{bmatrix}, \tag{3.12}$$

where we assume that

$$\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}} := \mathbf{C}_{\mathrm{coll}} - \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}_{\mathrm{I}}(\cdot)$$

is a linear operator. Compared to the single time step iterations (3.6), the composite problem iterations (3.12) use an approximation of $\mathbf{H}\mathbf{u}_\ell$. These iterations can be seen as a generalization of the method used for solving linear equations, because, when $\lambda_{\mathrm{E}} = 0$, we end up with the already introduced algorithm for linear equations. A straightforward convergence analysis is difficult, and because of this, we again turn to the analysis for the Dahlquist's test equation.

3.2.1. Convergence of the composite collocation iterations

Let us rewrite the main idea of iterations (3.12) for Dahlquist's test equation and let $\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}} = \mathbf{I}_M - \Delta T \lambda_{\mathrm{I}} \mathbf{Q}$ be the collocation problem for the implicit part of the equation. The global iterations then read

$$\begin{bmatrix} \mathbf{C}^{\mathrm{I}}_{\mathrm{coll}} & & & -\alpha\mathbf{H}_M \\ -\mathbf{H}_M & \mathbf{C}^{\mathrm{I}}_{\mathrm{coll}} & & \\ & \ddots & \ddots & \\ & & -\mathbf{H}_M & \mathbf{C}^{\mathrm{I}}_{\mathrm{coll}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_L \end{bmatrix}^{(k+1)} = \begin{bmatrix} \mathbf{u}_0 + \Delta T \lambda_{\mathrm{E}} \mathbf{Q}\mathbf{u}_1^{(k)} - \alpha \mathbf{H}_M \mathbf{u}_L^{(k)} \\ \Delta T \lambda_{\mathrm{E}} \mathbf{Q}\mathbf{u}_2^{(k)} \\ \vdots \\ \Delta T \lambda_{\mathrm{E}} \mathbf{Q}\mathbf{u}_L^{(k)} \end{bmatrix}. \tag{3.13}$$

When analyzing convergence, an important aspect to consider is the stability requirement of the underlying propagator. In particular, for the implicit propagator, we require some form of stability condition, which can be expressed as $\|(\mathbf{I}_M - \Delta T \lambda_{\mathrm{I}} \mathbf{Q})^{-1}\mathbf{H}\|_\infty \leq 1$. This matrix can be seen as a propagating matrix that propagates the solutions of equation $u_t = \lambda_{\mathrm{I}} u$ as $\mathbf{u}_{\ell+1} = (\mathbf{I}_M - \Delta T \lambda_{\mathrm{I}} \mathbf{Q})^{-1}\mathbf{H}\mathbf{u}_\ell$. This stability assumption is essential for proving convergence, regardless of the value of λ_{E}. Similar stability assumptions have already been made when analyzing and bounding the spectrum of the all-at-once systems in [52, 49]. Additionally, the stability requirement also appears in our analysis for the convergence of linear problems in Appendix A. Refer to Remark 3.7 for further information on the difference in stability definitions. The following theorem demonstrates the contraction rate of iterations (3.13).

Theorem 3.6
Let the iteration for a single time-step as in (3.8) *converge with a contraction factor*

$$\eta := \|\Delta T \lambda_{\mathrm{E}} (\mathbf{I} - \Delta T \lambda_{\mathrm{I}} \mathbf{Q})^{-1} \mathbf{Q}\|_\infty \leq 1$$

and let

$$\|(\mathbf{I}_M - \Delta T \lambda_{\mathrm{I}} \mathbf{Q})^{-1}\mathbf{H}\|_\infty \leq 1$$

be the stability requirement for the implicit part.

Define $\xi^{(k)} = \|\vec{\mathbf{u}}^{(k)} - \vec{\mathbf{u}}^*\|_\infty$*, where* $\vec{\mathbf{u}}^*$ *denotes the solution of fixed point iterations in* (3.13)*. Then*

$$\xi^{(k+1)} \leq \frac{\eta L + \alpha}{1 - \alpha} \xi^{(k)},$$

where L is the number of time steps.

Proof: Let the error be defined as $\mathbf{e}^{(k)} = \vec{\mathbf{u}}^{(k)} - \vec{\mathbf{u}}^{*}$. Reading equations in (3.13) block by block, we get

$$\begin{aligned}
\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}}\mathbf{e}_1^{(k+1)} - \alpha\mathbf{H}_M\mathbf{e}_L^{(k+1)} &= \Delta T\lambda_{\mathrm{E}}\mathbf{Q}\mathbf{e}_1^{(k)} - \alpha\mathbf{H}_M\mathbf{e}_L^{(k)},\\
\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}}\mathbf{e}_\ell^{(k+1)} - \mathbf{H}_M\mathbf{e}_{\ell-1}^{(k+1)} &= \Delta T\lambda_{\mathrm{E}}\mathbf{Q}\mathbf{e}_\ell^{(k)}, \quad 2 \le \ell \le L.
\end{aligned}$$

Rearranging the terms yields

$$\begin{aligned}
\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}}\mathbf{e}_1^{(k+1)} &= \Delta T\lambda_{\mathrm{E}}\mathbf{Q}\mathbf{e}_1^{(k)} - \alpha\mathbf{H}_M\mathbf{e}_L^{(k)} + \alpha\mathbf{H}_M\mathbf{e}_L^{(k+1)},\\
\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}}\mathbf{e}_\ell^{(k+1)} &= \Delta T\lambda_{\mathrm{E}}\mathbf{Q}\mathbf{e}_\ell^{(k)} + \mathbf{H}\mathbf{e}_{\ell-1}^{(k+1)}, \quad 2 \le \ell \le L.
\end{aligned}$$

Multiplying the equations by $(\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}})^{-1}$ from left gives

$$\begin{aligned}
\mathbf{e}_1^{(k+1)} &= \Delta T\lambda_{\mathrm{E}}(\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}})^{-1}\mathbf{Q}\mathbf{e}_1^{(k)} - \alpha(\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}})^{-1}\mathbf{H}\mathbf{e}_L^{(k)} + \alpha(\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}})^{-1}\mathbf{H}\mathbf{e}_L^{(k+1)},\\
\mathbf{e}_\ell^{(k+1)} &= \Delta T\lambda_{\mathrm{E}}(\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}})^{-1}\mathbf{Q}\mathbf{e}_\ell^{(k)} + (\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}})^{-1}\mathbf{H}\mathbf{e}_{\ell-1}^{(k+1)}, \quad 2 \le \ell \le L.
\end{aligned}$$

Now, applying the infinity norm, recalling the definition of η, and the fact that $\|(\mathbf{C}^{\mathrm{I}}_{\mathrm{coll}})^{-1}\mathbf{H}_M\|_\infty \le 1$ holds, we have

$$\begin{aligned}
\|\mathbf{e}_1^{(k+1)}\|_\infty &\le \eta\|\mathbf{e}_1^{(k)}\|_\infty + \alpha\|\mathbf{e}_L^{(k)}\|_\infty + \alpha\|\mathbf{e}_L^{(k+1)}\|_\infty,\\
\|\mathbf{e}_\ell^{(k+1)}\|_\infty &\le \eta\|\mathbf{e}_\ell^{(k)}\|_\infty + \|\mathbf{e}_{\ell-1}^{(k+1)}\|_\infty, \quad 2 \le \ell \le L.
\end{aligned}$$

Let us define $\xi_\ell^{(k)} := \|\mathbf{e}_\ell^{(k)}\|_\infty$. The equations now read

$$\begin{aligned}
\xi_1^{(k+1)} &\le \eta\xi_1^{(k)} + \alpha\xi_L^{(k)} + \alpha\xi_L^{(k+1)},\\
\xi_\ell^{(k+1)} &\le \eta\xi_\ell^{(k)} + \xi_{\ell-1}^{(k+1)}, \quad 2 \le \ell \le L.
\end{aligned}$$

Telescoping the inequalities, we can get a bound for $\xi_\ell^{(k+1)}$ as

$$\begin{aligned}
\xi_\ell^{(k+1)} &\le \eta\xi_\ell^{(k)} + \xi_{\ell-1}^{(k+1)}\\
&\le \eta\left(\xi_\ell^{(k)} + \xi_{\ell-1}^{(k)}\right) + \xi_{\ell-2}^{(k)} + \xi_{\ell-2}^{(k+1)}\\
&\vdots\\
&\le \eta\sum_{j=1}^{\ell}\xi_j^{(k)} + \alpha\xi_L^{(k)} + \alpha\xi_L^{(k+1)}.
\end{aligned} \tag{3.14}$$

Specifically, for $\ell = L$ we have

$$\xi_L^{(k+1)} \le \eta\sum_{j=1}^{L}\xi_j^{(k)} + \alpha\xi_L^{(k)} + \alpha\xi_L^{(k+1)},$$

which yields a bound for $\xi_L^{(k+1)}$ as

$$\xi_L^{(k+1)} \le \frac{1}{1-\alpha}\left(\eta\sum_{j=1}^{L}\xi_j^{(k)} + \alpha\xi_L^{(k)}\right).$$

Since $\xi^{(k)} = \max_\ell \xi_\ell^{(k)}$, we can bound $\xi_L^{(k+1)}$ as

$$\xi_L^{(k+1)} \le \frac{\eta L + \alpha}{1-\alpha}\xi^{(k)}. \tag{3.15}$$

Now, using the inequality for $\xi_\ell^{(k+1)}$ from (3.14) and combining it with (3.15), we have

$$\begin{aligned}\xi_\ell^{(k+1)} &\leq (\eta\ell + \alpha)\xi^{(k)} + \alpha\xi_L^{(k+1)} \\ &\leq (\eta\ell + \alpha)\xi^{(k)} + \alpha\frac{\eta L + \alpha}{1-\alpha}\xi^{(k)} \\ &\leq (\eta L + \alpha)\left(1 + \frac{\alpha}{1-\alpha}\right)\xi^{(k)} = \frac{\eta L + \alpha}{1-\alpha}\xi^{(k)}.\end{aligned}$$

Because the upper inequality holds for any ℓ, this concludes the proof. □

Remark 3.7
The stability requirement $\|(\mathbf{I}_M - \Delta T\lambda_{\mathrm{I}}\mathbf{Q})^{-1}\mathbf{H}\|_\infty \leq 1$ *from Theorem 2.9 is stricter than the stability requirement of a numerical method in a classical sense. The latter is motivated by making sure that the approximations, for stiff problems, satisfy* $|u(T_{\ell+1})| \leq |u(T_\ell)|$*. For collocation or RK methods, this considers just the last internal stage or their convex combination. In addition to this classical stability definition, here we assume that the inequality holds for all internal stages:* $|u(T_\ell + t_m)| \leq |u(T_\ell + t_{m+1})|$*. In practice, when comparing both stability areas for the test equation, the difference is minimal.*

Remark 3.8
How to interpret λ_{E} *and* λ_{I} *from Theorem 3.6? Firstly, let us comment on the case when* $f_{\mathrm{I}}(u) = \mathbf{A}u$ *and* $f_{\mathrm{E}} = f$*. Let* $\mathrm{Lip}_{\mathrm{E}}$ *be the Lipschitz constant of* f_{E}*:*

$$\|f_{\mathrm{E}}(x) - f_{\mathrm{E}}(y)\|_\infty \leq \mathrm{Lip}_{\mathrm{E}}\|x - y\|_\infty.$$

Defining $\lambda_{\mathrm{E}} := \mathrm{Lip}_{\mathrm{E}}$*, the proof of Lemma 3.3 and Lemma 3.4 and the convergence Theorem 3.6 still hold as stated.* λ_{I} *can be interpreted as an eigenvalue of the matrix* $\mathbf{A}$*, specifically, the stability condition that then needs to be satisfied in Theorem 3.6 reads*

$$\|(\mathbf{I}_M - \Delta T\mathbf{Q} \otimes \mathbf{A})^{-1}\mathbf{H}\|_\infty \leq 1,$$

and the contraction condition is

$$\eta := \|\Delta T\lambda_{\mathrm{E}}(\mathbf{I} - \Delta T\mathbf{Q} \otimes \mathbf{A})^{-1}\mathbf{Q}\|_\infty \leq 1.$$

This covers the interpretation of any case where f_{I} *is a linear function.*

Secondly, we examine the case when solving the all-at-once problem with an inexact Newton's method, meaning that we use the preconditioner for the composite collocation problem as defined in (3.4)*. Then,* $f_{\mathrm{I}}(u) = \mathbf{A} + \bar{\mathbf{J}}^{(k)}u$*, where* $\bar{\mathbf{J}}^{(k)}$ *is an approximation of the Jacobian matrix* (3.3)*, and* $f_{\mathrm{E}}(u) = f(u) - \bar{\mathbf{J}}^{(k)}u$*. This is now again covered in the previous discussion: the value* λ_{E} *is the Lipschitz constant of* f_{E}*. When the inexact Newton's method has a 'good' approximation of the Jacobian matrix,* λ_{E} *gets smaller as well, reducing the leading parameter* η *in the convergence bound of Theorem 3.6. Because of this, we expect faster convergence with the inexact Newton's method.*

The last observation worth mentioning is that when $\lambda_{\mathrm{E}} = 0$*, the bound from Theorem 3.6 then becomes the same as already proven bounds for the linear case.*

3.2.2. Iterative refinement

Until now, we have built an iterative process:

$$\mathbf{C}_\alpha \vec{\mathbf{u}}^{(k+1)} = \mathbf{C}_\alpha \vec{\mathbf{u}}^{(k)} - \mathbf{C}_f(\vec{\mathbf{u}}^{(k)}) + \vec{\mathbf{w}}, \tag{3.16}$$

that solves $\mathbf{C}_f(\vec{\mathbf{u}}) = \vec{\mathbf{w}}$, where

$$\mathbf{C}_f(\vec{\mathbf{u}}) = \mathbf{C}\vec{\mathbf{u}} - \begin{bmatrix} \Delta T(\mathbf{Q} \otimes \mathbf{I}_M)\mathbf{f}(\mathbf{u}_1) \\ \vdots \\ \Delta T(\mathbf{Q} \otimes \mathbf{I}_M)\mathbf{f}(\mathbf{u}_L) \end{bmatrix}, \tag{3.17}$$

and $\mathbf{C}$ and $\vec{\mathbf{w}}$ are defined in (2.16). From Theorem 2.12 we see that smaller parameters α can generate larger round-off errors because the matrix $\mathbf{C}_\alpha$ has a large condition number $\kappa(\mathbf{C}_\alpha) \approx 1/\alpha$. To avoid these rounding errors, we can rewrite the problem as

$$\vec{\mathbf{res}}^{(k)} = \vec{\mathbf{w}} - \mathbf{C}_f(\vec{\mathbf{u}}^{(k)}), \quad \mathbf{C}_\alpha \vec{\mathbf{c}}^{(k)} = \vec{\mathbf{res}}^{(k)}, \quad \vec{\mathbf{u}}^{(k+1)} = \vec{\mathbf{u}}^{(k)} + \vec{\mathbf{c}}^{(k)}, \tag{3.18}$$

a formulation known as *iterative refinement* [73][2]. Note that iterations (3.16) may refer to any implicit-explicit splitting.

Algorithm 3 Iterative refinement IMEX-based ParaDiag.

Input: $\mathbf{C}_f$, $\vec{\mathbf{w}}$, $\vec{\mathbf{u}}^{(0)}$, α, ζ
Output: a solution to $\mathbf{C}_f\vec{\mathbf{u}} = \vec{\mathbf{w}}$.

1: $k = 0$
2: $\vec{\mathbf{res}}^{(k)} = \vec{\mathbf{w}} - \mathbf{C}_f(\vec{\mathbf{u}}^{(k)})$ # form a residual
3: **while** $\|\vec{\mathbf{res}}_k\| > \zeta$ **do**
4: $\quad \vec{\mathbf{c}}^{(k)} = (\mathbf{C}_\alpha)^{-1}\vec{\mathbf{res}}^{(k)}$ # one iteration of Algorithm 1
5: $\quad \vec{\mathbf{u}}^{(k+1)} = \vec{\mathbf{u}}^{(k)} + \vec{\mathbf{c}}^{(k)}$ # update the solution
6: $\quad k = k + 1$
7: $\quad \vec{\mathbf{res}}^{(k)} = \vec{\mathbf{w}} - \mathbf{C}_f(\vec{\mathbf{u}}^{(k)})$ # form a new residual
8: **return** $\vec{\mathbf{u}}^{(k)}$

Firstly we will comment where does the parameter selection described in Section 2.5.3 emerge when using the iterative refinement approach (3.18). Intuitively, the same round-off errors are now affecting the residual. To formally show this, let us assume that we do not have an explicit part, meaning $\mathbf{C}_f = \mathbf{C}$, returning to our previously introduced linear iterations. A new iterate is computed as

$$\vec{\mathbf{u}}^{(k+1)} = \mathbf{C}_\alpha^{-1}\left(\mathbf{C}_\alpha - \mathbf{C}\right)\vec{\mathbf{u}}^{(k)} + \vec{\mathbf{w}}.$$

We can see that the rounding errors originate from applying $\mathbf{C}_\alpha^{-1}$ to a vector $(\mathbf{C}_\alpha - \mathbf{C})\vec{\mathbf{u}}^{(k)}$. In the iterative refinement approach, the matrix $\mathbf{C}_\alpha^{-1}$ is applied to the residual, yielding fixed point iterations on the residuals as

$$\vec{\mathbf{res}}^{(k+1)} = \left(\mathbf{I}_{\mathbf{MN}} - \mathbf{C}\mathbf{C}_\alpha^{-1}\right)\vec{\mathbf{res}}^{(k)}. \tag{3.19}$$

Since computing an inverse of $\mathbf{C}_\alpha$ introduces round-off errors, we can rewrite (3.19) as

$$\vec{\mathbf{res}}^{(k+1)} = \left(\mathbf{I}_{\mathbf{MN}} - \mathbf{C}\left(\mathbf{C}_\alpha + \mathbf{R}\right)^{-1}\right)\vec{\mathbf{res}}^{(k)}, \tag{3.20}$$

[2]The idea came while personally corresponding with Shu-Lin Wu

where $\mathbf{R}$ is the error produced when computing the factorization of $\mathbf{C}_\alpha$. In the notation of Lemma 2.11, this would read as $(\mathbf{C}_\alpha + \mathbf{R})x = w + \Delta w$, yielding $\mathbf{R}x = \Delta w$. Let $\mathbf{U}_\alpha := \mathbf{C}_\alpha - \mathbf{C}$, then from (3.20) we get

$$\vec{\mathbf{res}}^{(k+1)} = \left(\mathbf{I_{MN}} - \mathbf{C}\left(\mathbf{C} + \mathbf{U}_\alpha + \mathbf{R}\right)^{-1}\right)\vec{\mathbf{res}}^{(k)}. \tag{3.21}$$

For any error matrix $\mathbf{E}$ that satisfies $\|\mathbf{E}\mathbf{C}^{-1}\| \leq 1$, we can bound

$$\|\mathbf{I_{MN}} - \mathbf{C}\left(\mathbf{C} + \mathbf{E}\right)^{-1}\| \leq \frac{\|\mathbf{E}\|\,\|\mathbf{C}^{-1}\|}{1 - \|\mathbf{E}\|\,\|\mathbf{C}^{-1}\|} \tag{3.22}$$

and the bound is less than 1 when $\|\mathbf{E}\|\,\|\mathbf{C}^{-1}\| \leq \frac{1}{2}$. This is visible from the relation

$$\mathbf{I}_{MN} - \mathbf{C}(\mathbf{C} + \mathbf{E})^{-1} = (\mathbf{C} + \mathbf{E} - \mathbf{C})\,(\mathbf{C} + \mathbf{E})^{-1} = \mathbf{E}(\mathbf{C} + \mathbf{E})^{-1} = \mathbf{E}\mathbf{C}^{-1}(\mathbf{I}_{MN} + \mathbf{C}^{-1}\mathbf{E})^{-1}.$$

In our case $\mathbf{E} := \mathbf{U}_\alpha + \mathbf{R}$ and we expect convergence in the infinity norm when

$$\|\mathbf{U}_\alpha + \mathbf{R}\|_\infty\,\|\mathbf{C}^{-1}\|_\infty \leq (\|\mathbf{U}_\alpha\|_\infty + \|\mathbf{R}\|_\infty)\,\|\mathbf{C}^{-1}\|_\infty \leq (\alpha + \|\mathbf{R}\|_\infty)\,\|\mathbf{C}^{-1}\|_\infty < \frac{1}{2}.$$

Using the structure of preconditioner (similarly as for proving Theorem 2.12) we get

$$\|\mathbf{R}x\|_\infty = \|\Delta w\|_\infty \leq c\frac{L(3\varepsilon + \tau)}{\alpha}\|x\|_\infty,$$

for some contant c, concluding that an upper bound for the norm of the matrix in (3.21) is

$$\left(\alpha + c\frac{L(3\varepsilon + \tau)}{\alpha}\right)\|\mathbf{C}^{-1}\|_\infty. \tag{3.23}$$

If want to minimize the rounding errors originating from the residual, we end with one parameter α, the one that minimizes $\alpha + c\frac{L(3\varepsilon+\tau)}{\alpha}$. This is the same result we would get if we were to repeat the process of forming the α-adaptive strategy as proposed in Section 2.5.3.

Let us now assume that $\mathbf{C}_f$ is a linear operator. This is equivalent to solving linear differential equations with an explicit part in the iterations. Let $\mathbf{C}_f = \mathbf{C} - \mathbf{D}_{\Delta T}$, where the matrix $\mathbf{D}_{\Delta T}$ denotes how explicit iterates are handled, see (3.17). The fixed point iteration on the residual, including the rounding errors, now reads

$$\vec{\mathbf{res}}^{(k+1)} = \left(\mathbf{I_{MN}} - \mathbf{C}_f\left(\mathbf{C}_f + \mathbf{D}_{\Delta T} + \mathbf{U}_\alpha + \mathbf{R}\right)^{-1}\right)\vec{\mathbf{res}}^{(k)}. \tag{3.24}$$

Using the same constraint as in (3.22) for $\mathbf{E} = \mathbf{D}_{\Delta T} + \mathbf{U}_\alpha + \mathbf{R}$, we get convergence if

$$\|\mathbf{D}_{\Delta T} + \mathbf{U}_\alpha + \mathbf{R}\|_\infty\,\|\mathbf{C}_f^{-1}\|_\infty \leq \left(\alpha + c\frac{L(3\varepsilon + \tau)}{\alpha} + \|\mathbf{D}_{\Delta T}\|_\infty\right)\|\mathbf{C}_f^{-1}\|_\infty < \frac{1}{2} \tag{3.25}$$

is satisfied. The round-off errors emerging from the diagonalization procedure are now the same as in the linear case. This is to be expected since the same preconditioner is applied. If we want to minimize the errors, we would have the same approach as in the linear example, since the matrix $\mathbf{D}_{\Delta T}$ is block diagonal with blocks that depend on ΔT, $\mathbf{Q}$ and some linear operator f. Thus, the minimization of the errors depends on the first two terms in the bracket of equation (3.25).

It remains to comment on the norms $\|\mathbf{C}^{-1}\|_\infty$ and $\|\mathbf{C}_f^{-1}\|_\infty$. For this, we assume a more general case, where the matrix $\mathbf{C}_f$ has a form

$$\mathbf{C}_f = \begin{bmatrix} \mathbf{C}_1 & & & \\ -\mathbf{H} & \mathbf{C}_2 & & \\ & \ddots & \ddots & \\ & & -\mathbf{H} & \mathbf{C}_L \end{bmatrix} \tag{3.26}$$

where $\mathbf{C}_\ell = \mathbf{I}_M - \Delta T \mathbf{Q} \otimes \mathbf{A}_\ell$. The matrix can be factorized as

$$\mathbf{C}_f = \begin{bmatrix} \mathbf{C}_1 & & & \\ & \mathbf{C}_2 & & \\ & & \ddots & \\ & & & \mathbf{C}_L \end{bmatrix} \begin{bmatrix} \mathbf{I} & & & \\ -\mathbf{C}_2^{-1}\mathbf{H} & \mathbf{I} & & \\ & \ddots & \ddots & \\ & & -\mathbf{C}_L^{-1}\mathbf{H} & \mathbf{I} \end{bmatrix}$$

and we need to compute the inverse of the right factor, since the inverse of the block diagonal matrix is straightforward to compute. Let $\mathbf{B}_\ell = \mathbf{C}_\ell^{-1}\mathbf{H}$. With this, the inverse of the right factor is

$$\begin{bmatrix} \mathbf{I} & & & & \\ \mathbf{B}_2 & \mathbf{I} & & & \\ \mathbf{B}_3\mathbf{B}_2 & -\mathbf{B}_3 & \mathbf{I} & & \\ \vdots & \vdots & & \ddots & \\ \mathbf{B}_L \cdots \mathbf{B}_2 & \mathbf{B}_L \cdots \mathbf{B}_3 & \mathbf{B}_L \cdots \mathbf{B}_4 & \dots & \mathbf{I} \end{bmatrix},$$

from where we can conclude that the infinity norm of it is bounded by

$$||\mathbf{I}||_\infty + ||\mathbf{B}_L||_\infty + \cdots + \prod_{\ell=2}^{L} ||\mathbf{B}_\ell||_\infty.$$

If we assume that $||\mathbf{C}_\ell^{-1}||_\infty \leq b < 1$, for every ℓ, we have

$$||\mathbf{B}_\ell||_\infty = ||\mathbf{C}_\ell^{-1}\mathbf{H}||_\infty \leq ||\mathbf{C}_\ell^{-1}||_\infty \underbrace{||\mathbf{H}||_\infty}_{=1} \leq b < 1,$$

and we can bound the inverse as

$$||\mathbf{C}_f^{-1}||_\infty \leq \frac{1}{1-b} \max_{\ell=1}^{L} ||\mathbf{C}_\ell^{-1}||_\infty \leq \frac{b}{1-b}. \tag{3.27}$$

The restriciton $b < 1$ demands more than just requiring stability of the underlying method. The stability in a classical sense would be imposing $||\mathbf{H}\mathbf{C}_\ell^{-1}\mathbf{H}||_\infty \leq 1$ when the collocation nodes contain the right end of the interval. The discussion can be summarized in the following theorem.

Theorem 3.9
Let $\mathbf{C}_f$ be a linear operator defined as in (3.26)*. Then, for a given $\mathbf{C}$ and $\mathbf{C}_\alpha$, the norm of the residual in fixed point iterations* (3.16) *is bounded as*

$$||\vec{\mathbf{res}}^{(k+1)}||_\infty \leq \frac{\rho}{1-\rho} ||\vec{\mathbf{res}}^{(k)}||_\infty,$$

and convergent when

$$\rho = \frac{b}{1-b}\left(\alpha + ||\mathbf{D}_{\Delta T}||_\infty\right) < \frac{1}{2}$$

and $b = \max_{\ell=1}^{L} ||\mathbf{C}_\ell^{-1}||_\infty < 1$. Here, $\mathbf{D}_{\Delta T} = \mathbf{C} - \mathbf{C}_f$.

Furthermore, when the round-off errors emerging from the diagonalization of $\mathbf{C}_\alpha$ are included, and the inner systems are solved to a relative tolerance τ, then ρ has the form

$$\rho = \frac{b}{1-b}\left(\alpha + c\frac{L(3\varepsilon + \tau)}{\alpha} + ||\mathbf{D}_{\Delta T}||_\infty\right),$$

for some constant c.

Observing these results and comparing them to our previous theorem, Theorem 3.6, we see some differences. This is to be expected since one theorem examines the errors $\vec{\mathbf{e}}^{(k)} = \vec{\mathbf{u}}^* - \vec{\mathbf{u}}^{(k)}$, whereas Theorem 3.9 focuses on the residuals as $\vec{\mathbf{res}}^{(k)} = \mathbf{C}_f \vec{\mathbf{e}}^{(k)}$. This is also where the factor $\|\mathbf{C}_f^{-1}\|_\infty$ comes into play in our bounds. Another important difference in these two results is that the latter theorem is difficult to expand for nonlinear equations. The theorem for the test equation is on the other hand directly expandable as described in Remark 3.8. Nevertheless, it summarizes the discussion of round-off errors in iterative refinement.

In our numerical tests, we observe that the convergence depends on the number of time-steps L as in Theorem 3.6. When some problems diverged, it was not the fault of the round-off errors as our new theorem would indicate, since we only for this purpose set a very small τ to test this. However, if we look at the bounds (3.27), when b is close to 1, our bound takes the form $\|\mathbf{C}_f^{-1}\|_\infty \leq L$, introducing the L dependency in the new bound as well.

In the prevous chapter, in Section 2.5.3, the rounding errors are handled by always selecting a different parameter α, computed on runtime. It is done in order to balance the convergence rate and the rounding errors, yielding an adaptive-α strategy with an ascending array of parameters. In contrast, the iterative refinement approach allows us to choose a fixed parameter α for each iteration, and the round-off errors are automatically reduced with each new iteration, since $\vec{\mathbf{res}}^{(k)} \to 0$. The numerical tests show that when the 'optimal' α is computed, it is often too large, and as a consequence, slowing down the overall convergence. This is because the constant c that scales the round-off error estimates is often not straightforward to compute. Even though the 'perfect' parameter is not easily achievable, we see that lowering the relative solver tolerance for the inner solver τ would overall help the convergence. In practice, the best approach was to use a rather small α, correcting it if divergence is detected.

An additional cost is induced in the iterative refinement method, the construction of the residual $\vec{\mathbf{res}}^{(k)}$, slightly increasing communication time, since the processor (or a group of processors) P_ℓ has to pass $\mathbf{Hu}_\ell^{(k)}$ to $P_{\ell+1}$. This is explained in more detail in Subsection 4.2.3. However, the residual can be used as a stopping criterion, in addition to monitoring the norm of consecutive iterates $\|\vec{\mathbf{u}}^{(k+1)} - \vec{\mathbf{u}}^{(k)}\|$. The consecutive error is sometimes not a true measure of convergence since stagnation can be perceived as convergence.

Remark 3.10
Iterative refinement in the linear case may or may not be better than the adaptive-α strategy. The additional communication cost does not seem to dominate, especially since the number of outer iterations using the iterative refinement (3.18) *approach with a small parameter $\alpha \approx 10^{-8}$ is overall lower. However, compared to using a larger parameter $\alpha \approx 10^{-3}$, the system shifts* (2.23a) *can result in longer solves. For the heat equation, the runtimes using iterative refinement seem to be longer than when using the adaptive-α strategy, although the latter has more outer iterations. This is one of the reasons why just counting iterations in Parallel-in-Time methods is not an accurate representation of actual speedup [74]. The linear advection equation seemed to profit more from iterative refinement, making it for some tests a slightly better choice. The details can be found in Section 5.2.1.*

CHAPTER 4

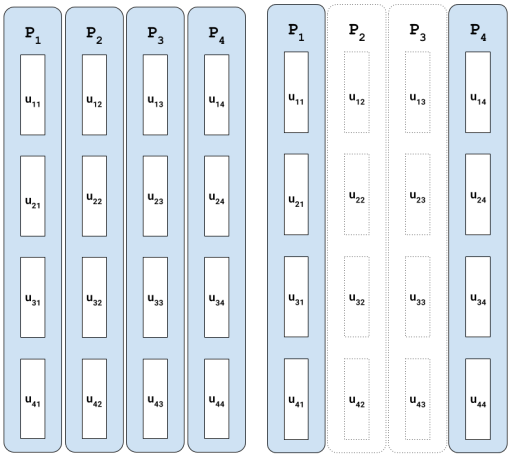
Implementation

Until now, we analyzed the idea of ParaDiag with the collocation method extensively, however, only theoretically. After setting up all the necessary mathematical structures in Chapters 2 and 3, we continue with a more applied part: the implementation. This chapter contains implementation details of a code written in `Python` in the framework of `mpi4py`, a message passing interface for `Python`, and `petsc4py`, a *Portable, Extensible Toolkit for Scientific Computation* [75, 76]. First, we discuss a parallelization strategy in Section 4.1, followed by detailed explanations of implementing some essential algorithm parts using the introduced strategy in Section 4.2. A more documentation-style section explains the code structure in `Python`, and last, speedup and efficiency estimates for the proposed ParaDiag implementations are presented in Section 4.4.

4.1. Parallelization strategies

The Message Passing Interface (MPI) is a message-passing standard designed to work on parallel computing architectures in a standardized and portable manner. This standard is defined by library routines that provide syntax and semantics. Several open-source implementations of MPI exist, encouraging the development of portable and scalable large-scale parallel applications. MPI uses distributed memory for communication among processes. In distributed memory, each processor has its own local memory, and the communication between processors occurs through passing messages. The MPI standard dictates several concepts, such as communicators, point-to-point communication, and collective communication. These features are a skeleton for all MPI-based applications.

In an MPI application, each process is assigned a number called *rank*, a unique identifier that distinguishes one process from another. The data is usually partitioned into smaller memory chunks that can be processed independently by each process. The processes can communicate point-to-point by sending data to each other, where the messages' origin and destination are specified by rank. Commonly used routines are `MPI_Send` for sending a message and `MPI_Recv` for receiving a message. The processes can also engage in a collective communication pattern within a group called a communicator. At the start of any MPI application, all processes belong to a communicator called `COMM_WORLD`. Being a member



(a) COMM_WORLD (b) COMM_LAST

Figure 4.1.: Communicator groups when the number of collocation points is $M = 4$ and the number of time steps is $L = 4$. The parallelization strategy used is $n_{\text{step}} = 4$, $n_{\text{coll}} = 1$ and $n_{\text{space}} = 1$.

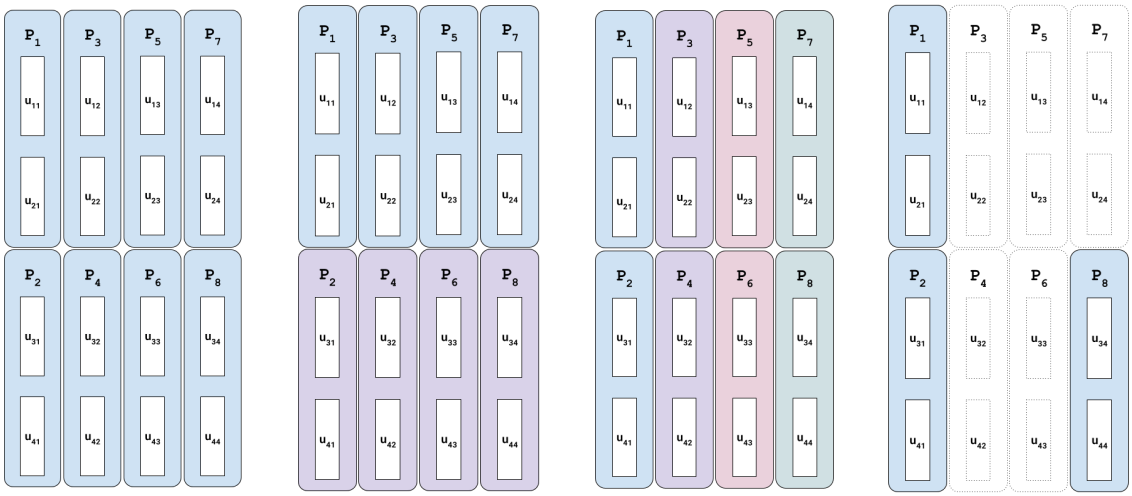
of a communicator allows processes to work together through already implemented routines, such as: broadcasting data from one process to all other processes using `MPI_Bcast` or computing a parallel sum on one process using `MPI_Reduce`. In contrast to these examples, where the message passing format is all-to-one, communication can happen in an all-to-all manner. One example is an `MPI_Allgather` after which each process in the communicator locally stores all chunks of data previously owned by all other processes. The MPI standard also has routines that can create new subcommunicators, allowing collective communication patterns to be carried out by the subgroup members. One process can be a part of multiple communicating groups, having a unique rank in every one of them. For example, Figure 4.2a shows all the processes of the communicator `COMM_WORLD` named P_1 to P_8 with ranks $1-8$. However, in the subcommunicator `COMM_ROW` in Figure 4.2b, their rank is relabeled so that P_2, P_4, P_6, P_8 have ranks $1-4$. Processor P_6 has rank 6 in the `COMM_WORLD` communicator, but rank 3 in the `COMM_ROW` communicator. Overall, writing an MPI application requires careful attention to data partitioning and process coordination to ensure efficient and correct execution.

Remark 4.1
In an MPI application, a rank is a number between 0 *and the size of the communicating group deduced by 1.*

Let n_{proc} denote the number of processors used in the application. The total number of processors is $n_{\text{proc}} = n_{\text{step}} n_{\text{coll}} n_{\text{space}}$, where n_{step} groups of processors handle the parallelization across the time steps and $n_{\text{coll}} n_{\text{space}}$ processors handle the parallelization of the collocation problem. The three-level parallelization strategy can be best explained by first explaining the parallelization across time steps, then adding the parallelization across the collocation points, and finally, by adding spatial parallelization.

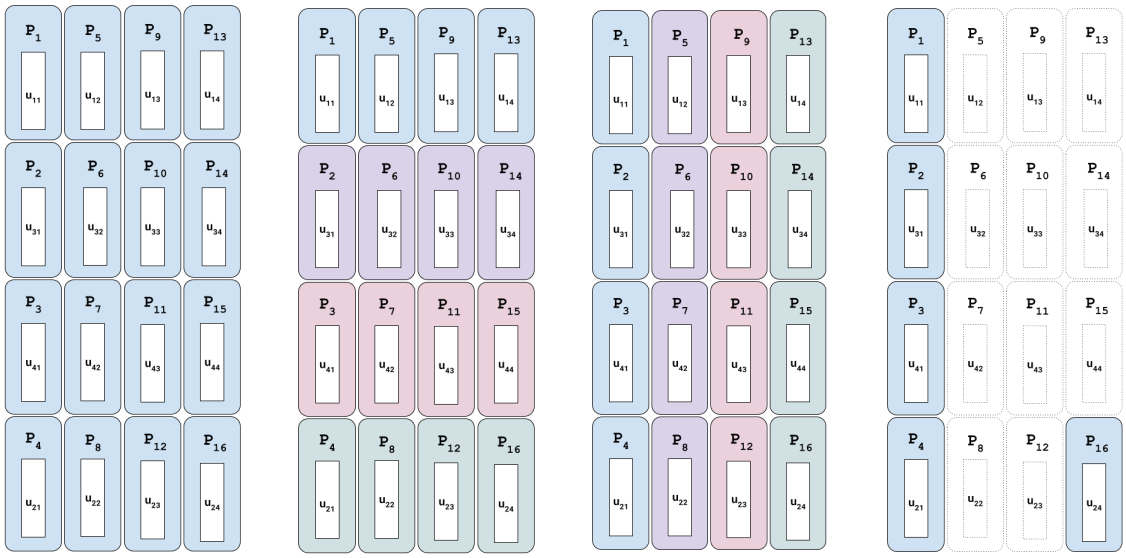
Let the number of processors be $n_{\text{proc}} = n_{\text{step}}$. In this particular implementation, we fix $n_{\text{step}} = L$ where L is a power of 2. This means that processor P_l stores one single approximation $\mathbf{u}_l^{(k)} \in \mathbb{C}^{MN}$, denoting the corresponding time step block of $\vec{\mathbf{u}}^{(k)} \in \mathbb{C}^{LMN}$ defined in (2.18). The main communication group created is the `MPI_COMM_WORLD`. For Algorithm 1 (the linear case with the adaptive α-strategy), an additional subcommunicator `COMM_LAST` is defined and used to transfer the approximation $\mathbf{u}_L^{(k)}$ in the last time step from P_L to P_1. With this, the right-hand side for the new iteration $\vec{\mathbf{r}}^{(k)}$ can be formed, and essentially,

only $\vec{\mathbf{r}}_1^{(k)}$ is changed. This corresponds to line 4 in Algorithm 1. Graphically, the communicators used are presented in Figure 4.1.



(a) COMM_WORLD (b) COMM_ROW (c) COMM_COL (d) COMM_LAST

Figure 4.2.: Communicator groups when the number of collocation points is $M = 4$ and the number of time steps is $L = 4$. The parallelization strategy used is $n_{\text{step}} = 4$, $n_{\text{coll}} = 2$ and $n_{\text{space}} = 1$.



(a) COMM_WORLD (b) COMM_ROW (c) COMM_COL (d) COMM_LAST

Figure 4.3.: Communicator groups when the number of collocation points is $M = 4$ and the number of time steps is $L = 4$. The parallelization strategy used is $n_{\text{step}} = 4$, $n_{\text{coll}} = 4$ and $n_{\text{space}} = 1$.

In addition, if available, more cores for the collocation-space problem can be utilized. Let the number of processors be $n_{\text{proc}} = n_{\text{step}} n_{\text{coll}}$, where we impose that the number of collocation points M is divisible by n_{coll}. This means that each group of n_{coll} processors, for each time step, stores M/n_{coll} internal stages of the collocation vector. In addition to the already mentioned communicating groups, we define `COMM_ROW` for communication across time steps (or rows) and `COMM_COL` for communication across the collocation nodes (or columns). Two different parallelization strategies are presented in Figures 4.2 and 4.3.

Figure 4.4.: Communicator groups when the number of collocation points is $M = 4$ and the number of time steps is $L = 4$. The parallelization strategy used is $n_{\text{step}} = 4$, $n_{\text{coll}} = 4$ and $n_{\text{space}} = 2$.

Lastly, on top of time parallelization, let n_{space} denote the number of processors handling the spatial parallelization. This means that one single approximation $\mathbf{u}_l^{(k)} \in \mathbb{C}^{MN}$ is spread across a group of $n_{\text{coll}} n_{\text{space}}$ processors. We assume the parallelization strategy for handling the collocation problem is exhausted, meaning $n_{\text{step}} = L$ and $n_{\text{coll}} = M$. We also assume that the spatial size of the problem N is divisible by n_{space}. With this setup, we define two more subcommunicators: `COM_SUBCOL_SEQ` and `COMM_SUBCOL_ALT`. Every collocation vector $\mathbf{u}_l^{(k)}$ is then spread across the `COM_SUBCOL_SEQ` communicator, which is then passed on as a communicator to `petsc4py`. `COM_SUBCOL_SEQ` is also the main communicating group the user needs to consider when assembling the matrix $\mathbf{A}$ and functions f and b. The matrix $\mathbf{A}$ is defined in row chunks, with N/n_{space} rows stored on the corresponding processors in the group. For example, the communicator `COMM_SUBCOL_ALT` is used when applying the matrix $\mathbf{Q}$ to a block vector because the corresponding internal stages of the collocation vectors need to be grouped together. See Figure 4.4 for clarity.

4.2. Employing the parallelization strategy

The parallelization strategy is designed in a user-friendly way where only n_{step}, n_{coll}, and n_{space} need to be defined. As a result, the subcommunicating groups are handled internally and automatically. In this subsection, we explain how essential steps of the proposed algorithms are executed.

4.2.1. Computing discrete Fourier transforms

The main feature of preconditioner $\mathbf{C}_\alpha$ in use is the diagonalization using the Fast Fourier Transform (FFT) and its inverse (IFFT). Applying a linear operator $\mathbf{F}^*$ in equation (2.19a) marks the start of our diagonalization process. After scaling the vector $\vec{\mathbf{r}}$ as $\widetilde{\vec{\mathbf{r}}} = (\mathbf{J}^{-1} \otimes \mathbf{I}_{MN})\vec{\mathbf{r}}$ in line 4 of Algorithm 1, we start the FFT. This subsection explains how the Radix-2 algorithm for computing the FFT works in parallel. It is a divide-and-concquer technique of multiplying a vector $\widetilde{\vec{\mathbf{r}}}$ with a matrix

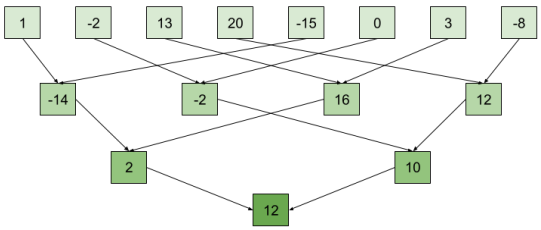


Figure 4.5.: A parallel sum.

$$\mathbf{F}^* = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(L-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \dots & \omega^{-2(L-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(L-1)} & \omega^{-2(L-1)} & \dots & \omega^{-(L-1)^2} \end{bmatrix},$$

where $\omega = e^{2\pi i/L}$ is a L^{th} primitive root and $L = 2^n$. The result $\vec{\mathbf{x}} = \mathbf{F}^* \widetilde{\vec{\mathbf{r}}}$, written by components, is

$$\mathbf{x}_l = \sum_{j=1}^{L} \omega^{-(l-1)(j-1)} \widetilde{\mathbf{r}}_j, \quad l = 1, \dots, L.$$

If we rewrite the summation over odd and even indices, we get:

$$\begin{aligned}\mathbf{x}_l &= \sum_{j=1}^{L/2} \omega^{-(l-1)(2j-2)} \widetilde{\mathbf{r}}_{2j-1} + \sum_{j=1}^{L/2} \omega^{-(l-1)(2j-1)} \widetilde{\mathbf{r}}_{2j} \\ &= \sum_{j=1}^{L/2} \left(\omega^2\right)^{-(l-1)(j-1)} \widetilde{\mathbf{r}}_{2j-1} + \omega^{-(l-1)} \sum_{j=1}^{L/2} \left(\omega^2\right)^{-(l-1)(j-1)} \widetilde{\mathbf{r}}_{2j}.\end{aligned}$$

When using two processors for each part of the summation, the complexity reduces to $O(L/2)$ algebraic operations. If we continue dividing these summations further and compute them similarly as a parallel sum on L processors, we end up with a computational cost of $O(L \log_2(L))$. The difference between the parallel sum (Figure 4.5) and the Radix-2 process is that the partial sums are scaled before summation (see line 16 in Algorithm 4) and that the computed solution is again partitioned over all processors. With this, the FFT significantly reduces the computational cost of $O(L^2)$ of a matrix-vector multiplication.

The key idea of building this communication pattern is to look at the binary representation of the time step stored on the processor. Let $l = (\overline{l_1 l_2 \dots l_n})_2$ be a binary digit representation of l, $n = \log_2(L)$. Then, a perturbed index of $\mathbf{x}_{l_i}$ is computed on the processor P_l and has a binary index $l_i = (\overline{l_n \dots l_2 l_1})_2$ (see Figure 4.6). In the end, each group of processors that stored $\widetilde{\mathbf{r}}_l$ now hold $\mathbf{x}_{l_i}$.

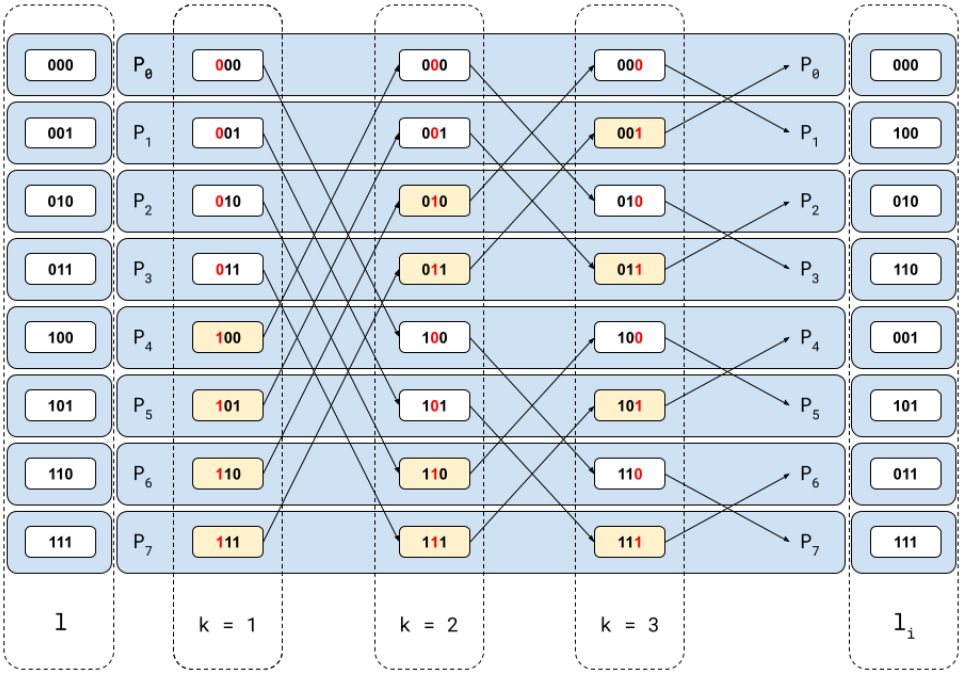


Figure 4.6.: An example of a Radix-2 butterfly communication structure for $L = 8$ time steps and $n_{\text{step}} = 8$ processors. Because the indices after the forward Fourier transform do not need rearranging, the computation proceeds with the perturbed blocks of $\mathbf{x}_{l_i}$. The counter k corresponds to the counter of Algorithm 6. The processors that exchange data with each other are the ones whose k^{th} digit differs in a binary representation of the time step they hold, marked in red. The boxes marked in yellow correspond to the `IF` statement being true in line 8 of Algorithm 6.

Let $\mathbf{r}_{\text{loc}}$ denote a corresponding chunk of $\widetilde{\mathbf{r}}_l$ stored on processor $P_{\texttt{rank}}$. The algorithm for computing the scaled FFT (lines 5, 6 in Algorithm 1) is presented in Algorithm 4 where $\mathbf{x}_{\text{loc}}$ is the corre-

sponding chunk of the output vector with the perturbed indices $\mathbf{x}_{l_i}$, now existing only on processor $P_{\texttt{rank}}$. In our implementation, the butterfly communication structure stretches across processors in the `COMM_ROW` (or `COMM_WORLD` in case $n_{\text{coll}} = 1$) subgroup, and the `rank` marks the time step index stored on $P_{\texttt{rank}}$. The communication cost is $O(\log_2(L))$ with chunks of memory sent and received being $O(MN/(n_{\text{coll}}n_{\text{space}}))$.

Algorithm 4 The parallel Radix-2 algorithm for computing the scaled Fast Fourier Transform.

Input: $\mathbf{r}_{\text{loc}}$, `COMM`
Output: $\mathbf{x}_{\text{loc}} = \mathbf{FFT}(\mathbf{r}_{\text{loc}})$

1: `rank` = `COMM`.get_rank() # rank of the processor in the group
2: $n = \log_2(L)$
3: $P = \overline{(\texttt{rank})_2} = (\overline{P_1 P_2 \dots P_n})_2$ # digits in the binary representation of rank
4: $R = \text{base10}((\overline{P_n \dots P_1})_2)$ # number in base 10 of reversed digits of P
5: $\omega = e^{2\pi i/L}$

6: **for** $k = 1, \dots, n$ **do** # stages of the butterfly
7: $\phi = 1$
8: **if** $P_k == 1$ **then**
9: $\phi = -1$
10: $r = R \,\%\, 2^k - 2^{k-1}$
11: $\mathbf{r}_{\text{loc}} = \omega^{-rL/2^k}\, \mathbf{r}_{\text{loc}}$ # redifine $\mathbf{r}_{\text{loc}}$

12: `comm_with` = $\text{base10}((\overline{P_1 \dots \text{flip}(P_k) \dots})_2)$ # base 10 with a flipped k^{th} digit
13: `req` = `COMM`.Isend($\mathbf{r}_{\text{loc}}$, destination = `comm_with`) # $P_{\texttt{rank}}$ sends $\mathbf{r}_{\text{loc}}$ to $P_{\texttt{comm\_with}}$
14: $\mathbf{v}_{\text{loc}}$ = `COMM`.Recv(source = `comm_with`) # $P_{\texttt{rank}}$ recieves $\mathbf{v}_{\text{loc}}$ from $P_{\texttt{comm\_with}}$
15: `req`.Wait()
16: $\mathbf{x}_{\text{loc}} = \mathbf{v}_{\text{loc}} + \phi \vec{\mathbf{r}}_{\text{loc}}$

17: **return** $\mathbf{x}_{\text{loc}}$

Remark 4.2
There are two fundamental ways of passing messages using MPI. Suppose processor A packs the data in a single message and sends the data to processor B via `MPI_Send`*. The message is then passed to a buffer and directed to processor B. Simultaneously, B has to indicate that it wants to receive the message with a* `MPI_Recv`*. In this scenario, both processors wait until their actions are completed: A will not continue until it can use the send buffer again, and B continues only after the receive buffer contains the newly received message. In other words, their communication is blocking.*

On the other hand, one can use non-blocking versions `MPI_Isend` *and* `MPI_Irecv`*. In these versions, the send and the receive buffers are allowed to be used again only when the message passing is successful; however, the application continues, allowing this request to be fulfilled later. In Algorithm 4 (line 13), all the processors perform a non-blocking* `MPI_Isend` *operation, allowing the application to receive the data in line 14. Success of* `req.Wait()` *in line 15 allows the application to continue because the send buffer is available again.*

If `MPI_Send` *is used in line 13, instead of a non-blocking* `MPI_Isend`*, processors A and B would be hanging in the send routine without a corresponding receive, leading to a deadlock.*

At this stage, the problem is decoupled; therefore, there is no need to rearrange the vectors back in the original order as done in the standard Radix-2 algorithm, which saves communication time. The inner system solves in (2.19b) can be carried out on these perturbed indices until the next Radix-2 for the parallel IFFT in (2.19c) (line 15 in Algorithm 1) is performed. The IFFT has an inverse communication structure, perturbing the indices back to the original state.

4.2.2. Solving decoupled problems

Solving the diagonal systems in (2.19b) requires more care. Each group or subgroup of `COMM_COL` simultaneously and independently solves its own system, storing the solutions in $\mathbf{y}_l$. Excluding the communication time, this part is expected to be the most costly one. After forming and diagonalizing the matrix $\mathbf{Q}\mathbf{G}_l^{-1}$ locally on each processor, $\mathbf{x}_l^1$ as in (2.24a) (line 10 in algorithm 1) is firstly computed as

$$(\mathbf{x}_l^1)_m = \sum_{j=1}^{M} [\mathbf{S}_l^{-1}]_{mj} (\mathbf{x}_l)_j, \quad (\mathbf{x}_l^1)_m, (\mathbf{x}_l)_j \in \mathbb{C}^N, \, \mathbf{S}_l \in \mathbb{C}^{M \times M}, \tag{4.1}$$

where $(\mathbf{x}_l)_m$ is a subvector of $\mathbf{x}_l \in \mathbb{C}^{NM}$ containing indices from mN to $(m+1)N$ which represents the corresponding implicit collocation stages. Depending on the parallelization strategy, these chunks are stored in $\mathbf{x}_{\text{loc}}$, as in the output of Algorithm 4.

When $n_{\text{coll}} = n_{\text{space}} = 1$, the sums are computed locally on each processor since all data is locally stored and no communication is needed.

When $n_{\text{coll}} > 1$, the summation is computed using `MPI_Reduce` on the `COMM_COL` level or, in case of spatial parallelization, on the `COMM_SUBCOL_ALT` level. Each processor locally computes the sum and then engages in a collective routine `MPI_Reduce`, which then computes a parallel sum on one of the processors. The communication complexity is $O(n_{\text{coll}} \log_2(n_{\text{coll}}))$. The length of the stored vector $\mathbf{x}_{\text{loc}}$ is $MN/(n_{\text{coll}} n_{\text{space}})$, therefore the chunks of memory being sent and received are $O(MN/(n_{\text{coll}} n_{\text{space}}))$.

The decoupled inner linear systems, after diagonalization across all time points (2.24b) (line 12 in algorithm 1), can be solved using three already implemented predefined solvers: a direct solver based on the perturbed LU-decomposition, a GMRES solver implemented in `scipy` or a predefined GMRES solver (without a preconditioner) from `petsc4py`. When $n_{\text{coll}} = M$, each linear system is simultaneously solved on one processor. When $n_{\text{coll}} < M$, one processor contains $M/n_{\text{coll}} > 1$ systems, and they are solved sequentially within a group. If $n_{\text{space}} > 1$, the communicator `COMM_SUBCOL_SEQ` is passed to `petsc4py` for handling the systems of size $N \times N$ in parallel. In this case, only a space-parallel solver can be used. The predefined spatial solver is user accessible and serves as an example with an intent to redefine it freely. See Section 4.3 for more details.

Equations (2.24c) and (2.23b) (lines 13 and 14 in Algorithm 1) are computed in exactly the same manner as (4.1).

4.2.3. Computing the residual

The extension to using iterative refinement heavily depends on the already discussed parts of Algorithm 1. The only thing that needs to be taken care of is the computation of the residual (line 7 in Algorithm

3). The residual vector in (3.18) can be expressed in three parts as

$$\vec{\mathbf{res}}^{(k)} = \underbrace{\begin{bmatrix} \mathbf{u}_0 + \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_L \end{bmatrix}}_{\vec{\mathbf{w}}} + \underbrace{\begin{bmatrix} 0 \\ \mathbf{H}\mathbf{u}_1^{(k)} \\ \vdots \\ \mathbf{H}\mathbf{u}_{L-1}^{(k)} \end{bmatrix}}_{\vec{\mathbf{h}}^{(k)}} + \underbrace{\begin{bmatrix} \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}(\mathbf{u}_1^{(k)}) - \mathbf{C}_{\text{coll}}\mathbf{u}_1^{(k)} \\ \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}(\mathbf{u}_2^{(k)}) - \mathbf{C}_{\text{coll}}\mathbf{u}_2^{(k)} \\ \vdots \\ \Delta T(\mathbf{Q} \otimes \mathbf{I}_N)\mathbf{f}(\mathbf{u}_L^{(k)}) - \mathbf{C}_{\text{coll}}\mathbf{u}_L^{(k)} \end{bmatrix}}_{\vec{\mathbf{d}}^{(k)}}.$$

The vector $\vec{\mathbf{w}}$ is a constant vector built just once. The chunks are then stored accordingly, depending on the parallelization strategy in use.

To assemble the vector $\vec{\mathbf{h}}^{(k)}$, each processor (or group of processors) containing the last collocation stage of $\mathbf{u}_\ell^{(k)}$ has to send it to the processor (or group of processors) containing $\mathbf{u}_{\ell+1}^{(k)}$. In case $n_{\text{step}} = L$, $n_{\text{coll}} = n_{\text{space}} = 1$, processor P_ℓ has to do one `MPI_Isend` to processor $P_{\ell+1}$, followed by a corresponding recieve. The chunk of memory being communicated is $O(N)$. This is done using the `MPI_COMM_WORLD` communicator. See Figure 4.1a for clarity.

The parallelization on the collocation level is triggered with a setup $n_{\text{step}} = L$, $n_{\text{coll}} \geq 1$, and $n_{\text{space}} = 1$. In this case, the last stage of $\mathbf{u}_\ell^{(k)}$ of length $O(N)$ is sent to the whole group of processors over which $\mathbf{u}_{\ell+1}^{(k)}$ is spread. First, the processor P_ℓ from the `COMM_ROW` group containing the last stage sends the vector to $P_{\ell+1}$ (here, the indexing of processors is within the group). After receiving the vector, the processors perform an `MPI_Bcast` within the `COMM_COL` communicator, sending the vector to each processor of that group. For example, if the parallelization strategy is as in Figure 4.2 and 4.3, the processors in the bottom row send the last stage to the processors on the right, followed by a broadcast vertically within the column.

In the case of spatial parallelization $n_{\text{space}} \geq 1$, the last stage of $\mathbf{u}_\ell^{(k)}$ is spread across multiple processors. The chunks of length $O(N/n_{\text{space}})$ are first sent within the `COMM_ROW` in the same fashion as before, but the vertical broadcast is now performed on the `COMM_SUBCOL_ALT` communicator. See Figure 4.4 for clarity.

Computing vector $\vec{\mathbf{d}}^{(k)}$ in the residual is more straightforward. The $\mathbf{C}_{\text{coll}}\mathbf{u}_\ell^{(k)} = (\mathbf{I}_{MN} - (\mathbf{Q} \otimes \mathbf{A}))\mathbf{u}_\ell^{(k)}$ contains an underlying part which is similar to equation (4.1). It can be seen as applying $\mathbf{Q} \otimes \mathbf{I}_N$ to vector containing block entries $\mathbf{A}(\mathbf{u}_\ell^{(k)})_m \in \mathbb{C}^{N \times N}$, where $(\mathbf{u}_\ell^{(k)})_m$ is a subvector containing indices from mN to $(m+1)N$. In case the parallelization strategy also parallelizes in space, the $\mathbf{A}(\mathbf{u}_\ell^{(k)})_m$ product is computed using `petsc4py` on the `COMM_SUBCOL_SEQ` level since the matrix $\mathbf{A}$ is in that case also stored in row-chunks. Vectors $\mathbf{f}(\mathbf{u}_L^{(k)})$ are computed locally and $\mathbf{Q} \otimes \mathbf{I}_N$ is applied in the same fashion as in equation (4.1).

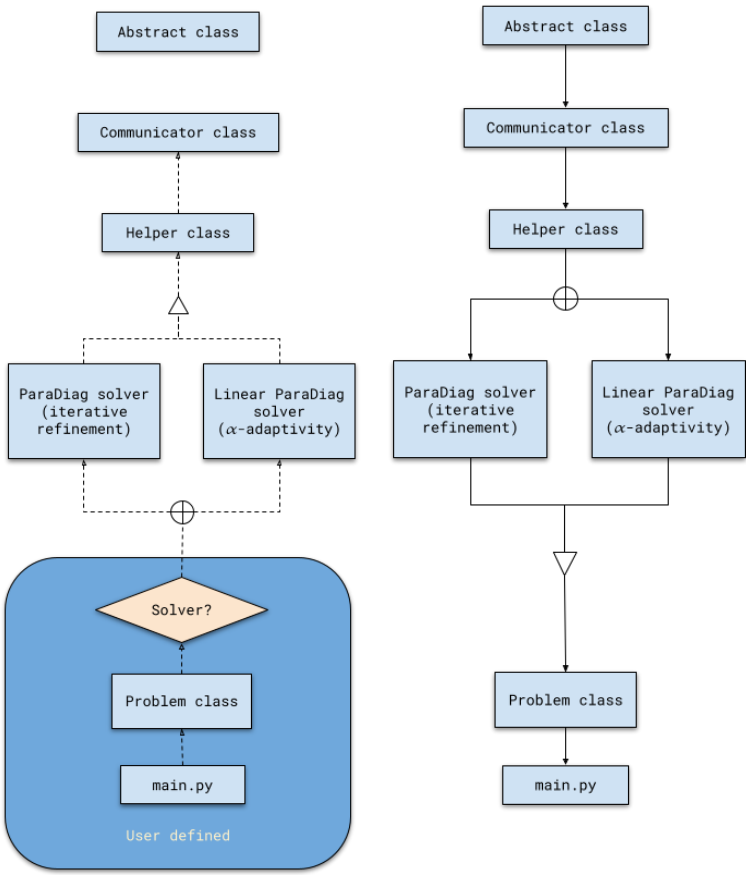
4.2.4. Computing average Jacobians

The implementation also supports the computation of the Jacobians. A simple switch variable decides whether the computation is performed or not. If so, in every iteration, a new approximation of a Jacobian is computed as an average. First, a local Jacobian $\mathbf{J}_\ell^{(k)}$ is computed on the processors that contain the last stage of $\mathbf{u}_\ell^{(k)}$. After that, the processors in the `COMM_ROW` group perform an `MPI_Allreduce` routine on the $\frac{1}{L}\mathbf{J}_\ell^{(k)}$ matrices, after which every processor storing the last stage implicit stage contains the average. The Jacobians are multiplied by $1/L$ before the summation, lowering the chances of an overflow.

In case $n_{\text{coll}} > 1$, $n_{\text{space}} = 1$, a broadcast within the `COMM_COL` is needed so that every processor gets the new approximation. In case $n_{\text{space}} > 1$, the broadcast is done within the `COM_SUBCOL_ALT` communicator.

4.3. Code structure

In this section, the structure of the implementation is explained. The design is user-friendly, where the user should just focus on defining the initial value problem in the `Problem class`, choosing a time-parallel solver class, and configuring the variables in `main.py`. The code is publicly available with a mini tutorial on GitHub [60]. The solver classes available are a linear solver with α-adaptivity and the iterative refinement-based solver. The variables in `main.py` are simple, such as choosing the number of parallel steps L, number of collocation nodes M, number of spatial points N, etc. Figure 4.7b shows the class hierarchy.



(a) Setup calls

(b) Class hierarchy

Figure 4.7.: Setup function calls and the inheritance hierarchy. The symbol $\oplus$ denotes *or*, meaning that just one of the solver classes is participating in the hierarchy. The flow is defined by the user, specifying a solver as a parent of the `Problem class`.

The `Abstract class` contains instances of variables inherited and used across every lower class in the hierarchy but become defined later on. For example, the communicator `COMM_WORLD` is instanced in the uppermost `Abstract class` but defined in the `Communicator class`. After that, it is accessible and

used within all the child classes. Another example is the definition of the **A** matrix. The variable is instanced in the `Apstract class` but defined by the user in the `Problem class`. This is possible with a call of a `setup` function within each class. The setups are performed bottom-up so that the matrix object **A** is defined when the `COMM_WORLD` is already an object, see Figure 4.7a for the `setup` function calls. In conclusion, this inheritance structure allows us to access objects across every class, whereas the bottom-up setup allows us to use the object definitions from top to bottom. It also allows us to define variables passed on on runtime instead of defining them in the `main.py`.

The `Communicator class` handles the initialization of the communicators described in section 4.1. The MPI communicating groups are defined depending on the variables set up in the `main.py` (or via runtime arguments). The only communicator a user needs is then renamed to `comm_matrix` to simplify the user experience.

`Helper class` is a collection of methods that are used as building blocks in the solver classes. Here, the time stepping method is defined through the **Q** matrix alongside every other object needed in the solver classes. The functions owned by this class handle different cases of the parallelization strategy. It is, by far, the collection of the most complicated and algorithmic methods, such as the computation of the FFT and the IFFT, computation of the residual, functions updating the right-hand side of each iteration, etc. With this, the flow in the solver classes is more straightforward, using just the inherited functions from the `Helper class`.

The two solver classes resemble the code in Algorithm 1 and Algorithm 3. An additional feature of the solver classes is that time-sequential runs are also supported, meaning that a sequential time solver can be combined with a space-parallel one. This case is carefully handled separately, ensuring that fully sequential runs have no communication overheads.

4.3.1. Setting up a problem class

The problem class consists of several vital variables. The first one is the definition of matrix **A** in (2.9) or the linear operator f_I in (3.7). The object is stored into a sparse matrix `Apar`, which can be assembled after the `super.setup()` function call. In this function call, firstly, a matrix communicator `comm_matrix` and variables `row_beg` and `row_end` are set up. After that, on the current processor, we have to store rows starting from a row indexed `row_beg` to a row indexed `row_end-1` in sparse format.

Other members of the class are functions. The function `bpar` is a definition of the function $b(t)$ in parallel, returning the rows indexed from `row_beg` to `row_end-1`. Another simple yet important function is the definition of the initial condition `u_initial`. The definition of f_E in (3.7) is a function `F`, and its first derivative is `dF`. A switch parameter `betas` determines whether `dF` is used or not. For example, if the array is defined as `betas = [0, 1]`, then in the first global iteration, $f_I = \mathbf{A}$ is used, and in every other, $f_I = \mathbf{A} + \overline{\mathbf{J}}^{(k)}$ is used. The explicit part f_E is also changed automatically to fit the splitting.

Some additional functions can be redefined or used as is. These are the `linear_solver` function and `norm` function so that the errors can be monitored in different norms. The `linear_solver` contains a `petsc4py` GMRES solver. Because of this, testing and coupling with other linear solvers is made easy.

Some pre-implemented problem classes can be found on GitHub [60]. They include different discretizations of the liner heat, advection, wave and Schrödinger equations, and nonlinear Allen-Cahn and Boltzmann equations. The parent of the class is user-defined and defines the parallel-in-time solver that will be used.

4.3.2. `main.py`

After choosing or implementing a problem class, an object of the problem class is made in `main.py`. With this, additional class attributes are defined, such as variables defining the parallelization strategy, number of time steps, number of collocation points, number of spatial points, etc. An example is included here.

```
from examples.linear.advection_2d_pbc_upwind5 import Advection
prob = Advection()

# choosing a number of points
prob.spatial_points = [700, 600]            # number of spatial points
prob.time_points = 2                        # number of collocation points

# choosing a time domain
prob.T_start = 0
prob.T_end = 0.1

# choosing the number of intervals handled in parallel
prob.time_intervals = 16                    # number of time steps in parallel
prob.rolling = 4                            # number of parallel windows

# choosing a parallelization strategy
prob.proc_col = 4                           # number of cores for the coll. problem
prob.proc_row = prob.time_intervals         # number of cores handling time steps

# choosing a solver
prob.solver = 'custom'

# setting maximum number of iterations
prob.maxiter = 5                            # maximum number of iterations
prob.smaxiter = 100                         # maximum number of inner solveri
                                           terations

# choosing a setting for the alpha sequence
prob.optimal_alphas = True

# setting tolerances
prob.tol = 1e-5                             # a stopping tolerance
prob.stol = 1e-7                            # inner solver stopping tolerance

prob.setup()                                # must be before solve()
prob.solve()                                # solve
prob.summary(details=True)                  # print details
```

A sequential solver in time steps, but parallel across the collocation nodes (or in space), is also supported and can be chosen by setting `time_intervals = 1` and `proc_col > 1`. The number of time steps is then governed by `rolling`. In general, this variable specifies how many windows of ParaDiag with will be applied in a windowed way, computing the first `time_intervals` in parallel and then using the

solution as a new initial value for the next parallel window. The variables are thoroughly explained in a mini tutorial on GitHub [60].

4.4. Computational complexity and speedup analysis

In this section, we explore the theoretical speedup of parallelization in time with the proposed algorithm. Speedup is a number that measures how many times is a parallel execution of a program faster compared to a serial execution. In a formula, speedup using n cores can be expressed as

$$s_n = \frac{T_1}{T_n},$$

where T_n is the time needed to execute the same algorithm on n cores. Another important measure accompanying speedup is efficiency. It measures how efficient the parallel execution of a program is and can be expressed as

$$e_n = \frac{T_1}{n\,T_n}.$$

Ideally, using n cores would be n times faster, resulting in $s_n = n$ and $e_n = 100\%$. However, this is almost never the case when communication overheads are counted in.

4.4.1. The linear case

In order to estimate speedup, we need computational complexity estimates for three different cases: an estimate for a completely sequential implementation T_1, an estimate if there are M processors available for solving the problem sequentially over time steps, but parallel across the collocation points T_M, and lastly, an estimate if we have LM processors to solve in parallel across the steps (lines 6 and 17 in Algorithm 1) and across the collocation points (lines 11–13 in Algorithm 1) T_{ML}. The last time corresponds to a setup when $n_{\text{step}} = L$ and $n_{\text{coll}} = M$. Communication times in this model are ignored, and we assume to handle algebraic operations on the same amount of memory chunks $O(N)$ in all cases. The estimates can be given as follows:

$$\begin{aligned} T_1 &= L(MT_{\text{sol}} + 2M^2) = LM(T_{\text{sol}} + 2M), \\ T_{\text{M}} &= L(T_{\text{sol}} + 2M\log_2(M)), \\ T_{\text{ML}} &= k(T_{\text{sol,par}} + 2\log_2(L) + 3M\log_2(M)). \end{aligned}$$

Quantity T_1 denotes the computational complexity when solving the collocation problem (2.6) sequentially over L time steps directly via diagonalization of $\mathbf{Q}$. For each step, the solution is obtained by solving each system on the diagonal, one by one, M times, with a solver complexity of T_{sol}. The expression $2M^2$ stands for the two matrix-vector multiplications needed for the diagonalization of $\mathbf{Q}$.

T_{M} denotes the complexity for solving the collocation problem sequentially over L time steps via diagonalization, except these diagonal systems of complexity T_{sol} are solved in parallel across M processors. The two matrix-vector products can be carried out in parallel with a computational complexity of $2M\log_2(M)$.

At last, T_{ML} represents the computational complexity when $n_{\text{step}}n_{\text{coll}} = LM$ processors. Here, k denotes the number of outer iterations, and $2\log_2(L) + 3M\log_2(M)$ is the complexity of operations when using the communication strategies as discussed in Section 4.1. The solver complexity $T_{\text{sol,par}}$ may

differ from T_{sol} since differently conditioned systems may be handled on the diagonals. Also, the algorithm requires solving complex-valued problems even for real-valued systems, which could also cause an overhead, depending on the solver at hand. Remark 4.3 comments more on the solver complexities.

Hence, the theoretical speedups for $k \in \mathbb{N}$ iterations look like:

$$\frac{T_1}{T_{\text{ML}}} = \frac{LM(T_{\text{sol}} + 2M)}{k(T_{\text{sol,par}} + 2\log_2(L) + 3M\log_2(M))}, \tag{4.2a}$$

$$\frac{T_{\text{M}}}{T_{\text{ML}}} = \frac{L(T_{\text{sol}} + 2M\log_2(M))}{k(T_{\text{sol,par}} + 2\log_2(L) + 3M\log_2(M))}. \tag{4.2b}$$

The true definition of speedup would be (4.2a); however, the baseline method in our algorithm is also a parallel method (parallel across the method, i.e. over the collocation nodes); therefore, we also need to take (4.2b) into account. For $L \geq 2$ it holds

$$2\log_2(L) + 3M\log_2(M) \geq 2M$$

which in combination with (4.2a) gives

$$\frac{T_1}{T_{\text{ML}}} \leq \frac{LM}{k}\frac{T_{\text{sol}} + 2M}{T_{\text{sol,par}} + 2M}. \tag{4.3}$$

If we combine the fact that

$$2\log_2(L) + 3M\log_2(M) \geq 2M\log_2(M)$$

is true for all M, L and (4.2b), we get

$$\frac{T_{\text{M}}}{T_{\text{ML}}} \leq \frac{L}{k}\frac{T_{\text{sol}} + 2M\log_2(M)}{T_{\text{sol,par}} + 2M\log_2(M)}. \tag{4.4}$$

The speedup estimates (4.3) and (4.4) roughly depend on the ratio between how many steps and nodes are handled in parallel and the number of outer iterations. On the other hand, the efficiency depends on the number of outer iterations $e \approx 1/k$, deteriorating fast with every iteration.

This is not a surprise when dealing with Parareal-based parallel-in-time methods, and our speedup estimates fit the usual theoretical bounds in this field, too. Most importantly, as in Parareal, minimizing the number of iterations is crucial to achieving parallel performance. This is the reason why carefully choosing the $(\alpha_k)_{k\in\mathbb{N}}$ sequence is important; it brings the number of outer iterations down. Lowering the outer iteration count for even a few iterations can provide significant speedup. This claim is not just supported theoretically but in our test runs as well.

4.4.2. Iterative refinement

An extension to the iterative refinement can be expressed using already made estimates for the purely linear case. Let the parallelization setups for estimating T_1, T_M, and T_{ML} be the same as for the previous case. Following the same logic, the estimates read:

$$\begin{aligned}
T_1 &= L\bar{k}(MT_{\text{sol}} + 2M^2 + T_{1,\text{res}}),\\
T_{\text{M}} &= L\bar{k}(T_{\text{sol}} + 2M\log_2(M) + T_{M,\text{res}}),\\
T_{\text{ML}} &= k(T_{\text{sol,par}} + 2\log_2(L) + 3M\log_2(M) + T_{ML,\text{res}}).
\end{aligned}$$

Since solving an equation on a single time step is an iterative process, $\bar{k}$ denotes the average number of iterations over L time steps, and k is the same as before, the global number of iterations when doing a fully-parallel run.

A novelty in each iteration is the computation of the residual. The dominant part in the computational complexity $T_{1,\text{res}}$ is the application of the $\mathbf{Q}$ matrix two times, once on a vector $\mathbf{A}\mathbf{u}_\ell^{(k)}$ and the second time on $\mathbf{f}(\mathbf{u}_\ell^{(k)})$. We can assume that the matrix-vector multiplication has a complexity of N since the matrices in play are usually sparse, and in total, we have M multiplications. With this, we have $T_{1,\text{res}} = 2M^2 + MN$. The complexity $T_{M,\text{res}}$ has the application of the $\mathbf{Q}$ two times, which is on M processors $2M\log_2(M)$ and a matrix-vector computation on each processor. In total, it reads $T_{M,\text{res}} = 2M\log_2(M) + N$. The same computation is carried out on ML cores, concluding $T_{ML,\text{res}} = 2M\log_2(M)+N$.

With this, the theoretical speedup estimates read

$$\frac{T_1}{T_{\text{ML}}} = \frac{LM\bar{k}(T_{\text{sol}} + 4M + N)}{k(T_{\text{sol,par}} + 2\log_2(L) + 5M\log_2(M) + N)}, \tag{4.5a}$$

$$\frac{T_{\text{M}}}{T_{\text{ML}}} = \frac{L\bar{k}(T_{\text{sol}} + 4M\log_2(M) + N)}{k(T_{\text{sol,par}} + 2\log_2(L) + 5M\log_2(M) + N)}. \tag{4.5b}$$

For $L \geq 4$ it holds

$$5M\log_2(M) + 2\log_2(L) \geq 4M$$

which in combination with (4.5a) gives

$$\frac{T_1}{T_{\text{ML}}} \leq \frac{LM\bar{k}}{k}\frac{T_{\text{sol}} + 4M + N}{T_{\text{sol,par}} + 4M + N}. \tag{4.6}$$

To get the other speedup estimate, we can use the inequality, true for every M and L

$$5M\log_2(M) + 2\log_2(L) \geq 4M\log_2(M),$$

which in combination with (4.5b) gives

$$\frac{T_{\text{M}}}{T_{\text{ML}}} \leq \frac{L\bar{k}}{k}\frac{T_{\text{sol}} + 4M\log_2(M) + N}{T_{\text{sol,par}} + 4M\log_2(M) + N}. \tag{4.7}$$

The speedup estimates (4.6) and (4.7), similarly as before, depend on the ratio between the average number of iterations per time step and the number of how many steps are handled in parallel and the number of outer iterations. The efficiency roughly depends on the ratio between the number of average iterations and outer iterations $e \approx \bar{k}/k$. In conclusion, lowering the number of outer iterations is an important part of gaining good speedup and efficiency.

Remark 4.3
It has to be kept in mind that the computational complexities of system solves should stay similar $T_{\text{sol,par}} \approx T_{\text{sol}}$ while trying to minimize the number of outer iterations. For example, we can choose a smaller parameter α to lower the convergence factor, expecting fewer outer iterations. By doing so, we would also need to balance the rounding errors by solving the inner systems to a better accuracy ε. This would balance the rounding errors, ending up with errors being approximately ε/α. However, the systems in the sequential run usually do not have to be solved to machine precision, but rather to some accuracy $\tau \gg \varepsilon$. In the end, this results in an imbalance in the system solving complexities $T_{\text{sol,par}} \gg T_{\text{sol}}$, which would again reduce speedup.

CHAPTER 5

Numerical results

The structure of this chapter is as follows. Section 5.1 comments on weak and strong scaling of time-parallel methods. Next, in Section 5.2, we discuss the challenges associated with selecting appropriate test cases for time-parallel integration methods and examine specific difficulties in evaluating performance measures that may result in overestimating the method's performance. In Section 5.2.2, we provide a conclusion on how to select appropriate benchmark setups that will be employed in the main results sections, which numerically test our ParaDiag method for linear (Section 5.3) and nonlinear (Section 5.4) equations. Throughout Section 5.2.1, a collection of example test cases and setups that lead to overperformance or faulty conclusions, we refer to future benchmarks performed in Sections 5.3 and 5.4 as already known results, even though they show up chronologically later in the manuscript. The structure is nontraditional because we first want to stress the importance of manufacturing meaningful results, especially since some overlooked misconceptions can be very easily introduced silently. Section 5.3 analyzes the ParaDiag method coupled with the collocation problem on linear equations, namely the heat and advection equation, where Section 5.4 examines the method for the nonlinear Allen–Cahn and Boltzmann equation.

All the results presented here were obtained by the implementation discussed in Chapter 4, and performed on the supercomputer JURECA-DC, a supercomputer at Forschungszentrum Jülich. The system combines a flexible Data Centric (DC) module, based on the Atos BullSequana XH2000 [77]. The configuration used consists of 480 compute nodes containing 2 sockets with AMD EPYC 7742 processors where each processor has 2 sockets with 64 cores.

5.1. Parallel scaling

The type of scaling where the problem size is fixed, but the number of processors varies is called *strong scaling*. To do a thorough case study of how well our proposed ParaDiag method behaves for a given equation, we first pick a desired tolerance ζ we want our approximation to satisfy. Then, while increasing the number of processors, we must ensure that we always solve the same equation on the same time domain. This is done by fixing the number of time steps L we want to scale for and choosing a time step

ΔT for the propagator to reach the desired accuracy. The time domain we are solving for is then fixed as $[T_0, T_0 + L\Delta T]$. Then, depending on the number of processors for the parallelization across time steps, we compute parallel moving windows of ParaDiag. For example, if we want to cover a time domain $[0, 8\Delta T]$ with $n_{\text{step}} = 4$ processors, we compute the first 4 time steps in parallel and then propagate the solution further for the next 4 time steps, in parallel (see Figure 5.1). As a result, the benchmarks always produce the approximation at the same point in time.

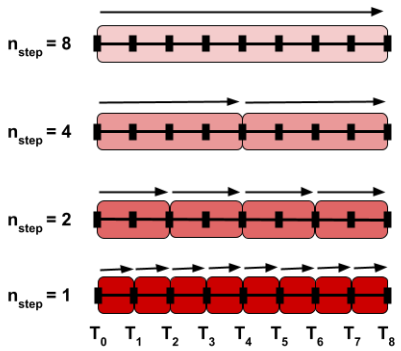


Figure 5.1.: Strong scaling with ParaDiag. The grouped intervals are solved in parallel. n_{step} processors determine the size of the moving window.

Another type of scaling is *weak scaling*. It is a benchmark where the problem size changes according to the number of given processors. Since we want to keep solving the same equation (not changing the time-space domain), weak scaling would translate to fixing the time domain $[T_0, T_0 + T]$ and changing the number of parallel time steps since each time step corresponds to one processor. Consequently, each parallel window of ParaDiag would solve the equation for $n_{\text{step}} = L = 1, 2, \ldots, L_{\max}$ time steps, with a corresponding step size $\Delta T = T/L$ on L processors. Because we would want to avoid under- or over-resolve in time, we would need to change our spatial discretization accordingly to match the time discretization error. The discretization error is now changing, therefore, to present honest speedups, we would have to solve to a different tolerance ζ, somewhere around the discretization error. Ultimately, we end up with a weak scaling plot in which each point solves the same equation to a different tolerance with a different discretization error. Solving the problem to three different thresholds $\zeta \in \{\zeta_1, \zeta_2, \zeta_3\}$ can be seen as weak scaling since the tolerance dictates the discretization order, and with this, the problem size. Because of this, strong scaling results for three thresholds $\zeta \in \{\zeta_1, \zeta_2, \zeta_3\}$ are presented since they are more valuable from a mathematical point of view.

5.2. How to compare time-parallel methods and codes

The primary objective of this section is to underscore the challenges involved in evaluating the effectiveness of parallel-in-time methods through examples. In the absence of suitable numerical experiments, performance of these methods can be overestimated, leading to faulty conclusions. It is critical to highlight some potential shortcomings to promote greater awareness among researchers. The section is written lightheartedly, drawing inspiration from the article *Twelve Ways To Fool The Masses When Giving Parallel-In-Time Results* [74] and providing additional examples that are specific to our newly derived ParaDiag method coupled with the collocation problem.

Several widely used benchmarks in the Parallel-in-Time community include the heat equation, the linear advection equation, the Allen–Cahn equation, and Burger's equation, among others. However, the examples presented frequently lack several crucial setup parameters, and manual comparison is the only viable means of evaluating different parallel-in-time methods. Additionally, varying time-stepping schemes are only supported by some implementations, further complicating the comparison. Therefore, a direct performance comparison with other methods, such as MGRIT and PFASST, is beyond the scope of this analysis and is reserved for future research.

It is worth to mention that some comparison has been made between ParaDiag, MGRIT, and Parareal in the preliminary work of [58]. The method used for comparison is backward Euler only, without the parameter adaptivity and without consideration of inexactness in the system solves. Despite these shortcomings, the approach still shows promising scaling properties, even outperforming the classical PinT methods.

5.2.1. Presenting numerical results with ParaDiag

Compare runtimes

Let us assume that the theoretical component of ParaDiag has been developed, and it is anticipated that rounding errors will pose a challenge. To address this issue, we adopt an iterative refinement approach as in (3.18) and monitor the number of iterations required to achieve convergence. We evaluate the performance of ParaDiag under different fixed parameter values $\alpha = 10^{-4}, 10^{-8}, 10^{-12}$, while solving the linear heat and linear advection equations and achieving three different accuracies for the approximation, $\zeta = 10^{-5}, 10^{-9}, 10^{-12}$. The parameter configuration for the equations is identical to that presented in Table 2.2.

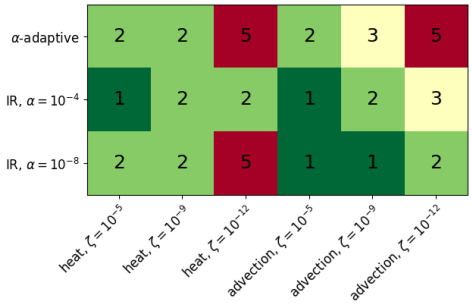
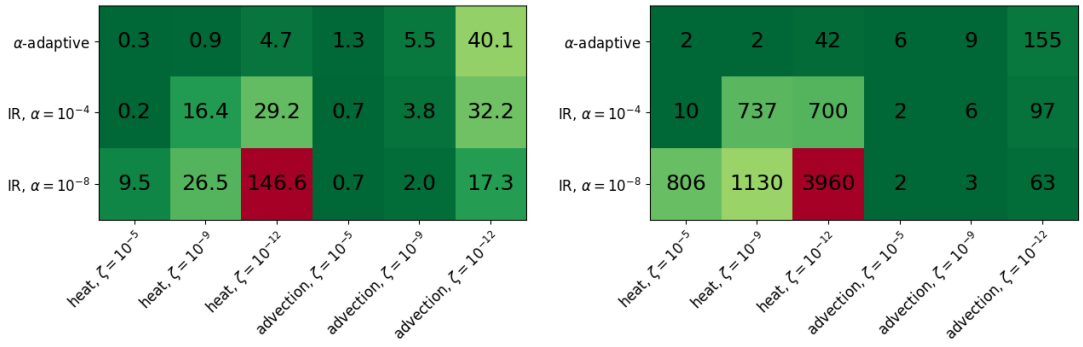


Figure 5.2.: Number of ParaDiag iterations for the linear heat and the linear advection equation, solved to three different tolerances ζ. The first row contains the number of iterations when the α-adaptive strategy is used, whereas the remaining rows contain the number of iterations when a fixed parameter α within iterative refinement is used.

The results are illustrated in Figure 5.4. We see that convergence is achieved in only 1 or 2 iterations, which is superior to the outcomes obtained in Figure 2.3. While some choices of α demonstrate that the round-off errors are still considerable, we can safely assume that values $\alpha = 10^{-4}, 10^{-8}$ are a good and robust choice. Given that our parallel-in-time method solves both parabolic and hyperbolic prototype equations, it is a highly promising approach. Moreover, efficiency estimates suggest that our parallel implementation efficiency is expected to be roughly around $e = 1/k$, where k denotes the number of outer iterations, as presented in Section 4.4. Therefore, we anticipate an efficiency of around 50% – a remarkably favorable outcome when considering parallel-in-time methods!

In order to verify the results on another numerical example, we can count the number of outer iterations. To accomplish this, we monitor the number of iterations until convergence for the setups presented in Table 5.2. The outer iterations of ParaDiag implemented with iterative refinement and ParaDiag employing the adaptive α-strategy is shown in Figure 5.2. The results show that using iterative refinement reduces the number of outer iterations, making it again a more favorable approach!



(a) Wall clock time in seconds

(b) Maximum numberof GMRES iterations.

Figure 5.3.: Wall clock times and the total number of the maximum number of inner GMRES iterations per time step for parallel runs of the linear heat and the linear advection equation, solved to three different tolerances ζ. The figures compare ParaDiag with the α-adaptive strategy to ParaDiag with a fixed parameter α using iterative refinement.

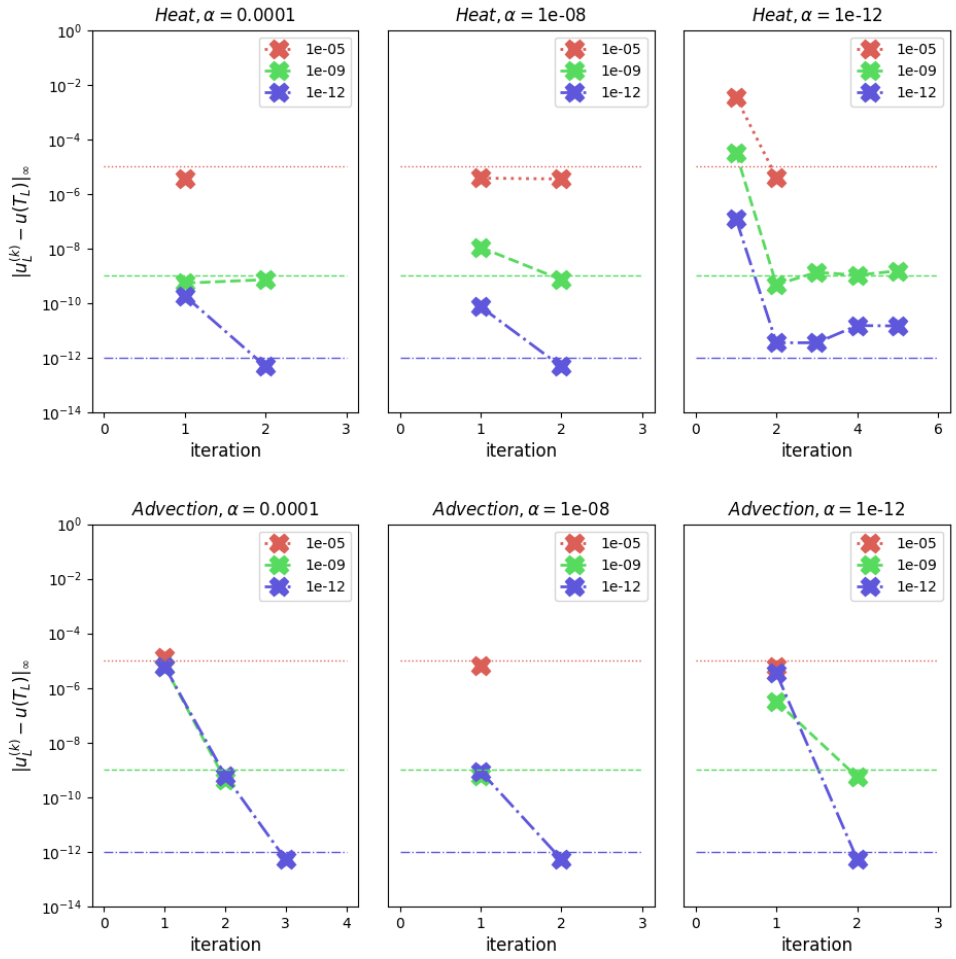


Figure 5.4.: Convergence curves for ParaDiag with iterative refinement for different tolerances ζ. The vertical lines represent the tolerances for the stopping criteria: red for the error $\|u(T_L) - u_L^{(k)}\|_\infty$ reaching under 10^{-5}, green for 10^{-9} and blue for 10^{-12}.

Despite the promising results, counting outer iterations in parallel-in-time algorithms may not necessarily be correlated with wall clock times. Figure 5.3a displays parallel runtimes for the same setups Figure 5.2 shows the number of outer iterations for. It is evident that the computation time required for solving the heat equation is significantly greater than expected. Additionally, previously deemed optimal, the parameter $\alpha = 10^{-8}$ now underperforms. The reason is that the inner systems are poorly conditioned and take a longer time to solve. This has also been observed in Subsection 5.3.1, and the smaller the value, the worse it gets. On the other hand, the iterative refinement approach may benefit the linear advection equation. However, the optimal value of α remains uncertain. The α-adaptive strategy, while not minimizing wall clock time in every case, appears to be a robust option.

Remark 5.1
Counting the number of inner GMRES iterations would also not predict the wall clock times very well. If that were the case, we would expect the runs with 10 *inner GMRES iterations to have a greater wall clock time than those with* 2 *inner GMRES iterations. However, this is not the case, and the run with fewer total GMRES iterations takes* $\sim$ 50% *longer.*

Choose the right time step

To showcase the exceptional performance of our parallel-in-time method, we test it for the linear advection equation, which is governed by

$$u_t + u_x + u_y = 0, \quad (t, x, y) \in [0, T] \times [0, 1]^2.$$

Figure 5.5 presents noteworthy speedup for a hyperbolic type equation, achieving a factor of up to 20 on 192 cores when parallelizing across time steps and $M = 3$ collocation nodes. Previously, a modest speedup of 7 had been reported in Figure 5.19a.

Furthermore, we also test our newly proposed method based on the composite collocation problem iterations solved via iterative refinement (3.18), where we treat the nonlinear right-hand side of the equation implicitly. The base time-stepping propagator is chosen to be implicit Euler, and we apply it to the Allen–Cahn equation

$$u_t = \Delta u + \frac{1}{\varepsilon^2} u(1 - u^2), \qquad (5.1)$$
$$(t, x) \in [0, T] \times [-8, 8]^2,$$

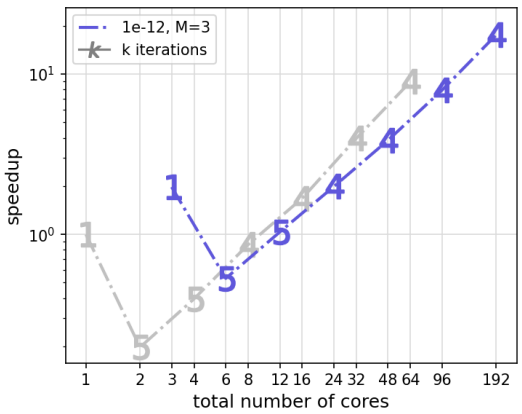
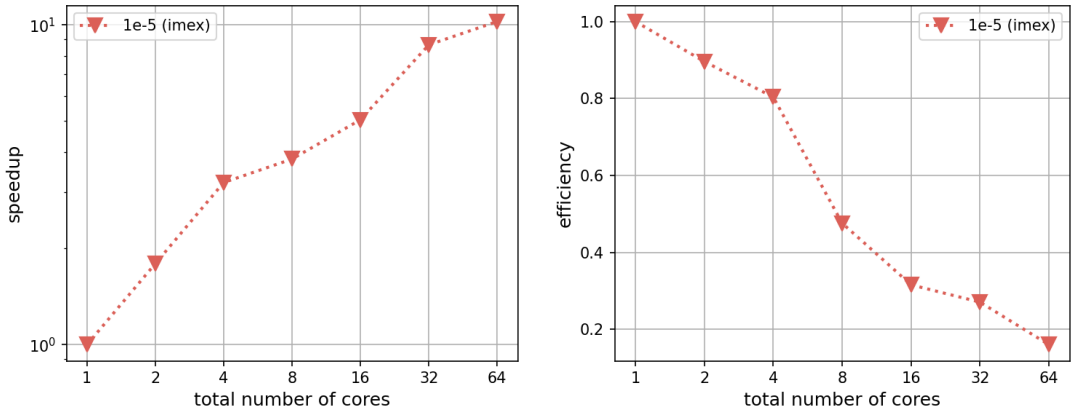


Figure 5.5.: Speedup for the advection equation, solved in parallel across time steps and the collocation nodes. The gray plots represent scaling across time steps only.

where $\varepsilon = 0.01$. The strong scaling plots are depicted in Figure 5.6, again demonstrating exceptional speedup despite the equation's Lipschitz constant of the right-hand side being approximately equal to $1/\varepsilon^2$. Our newly proposed approach is Jacobian-free and thus does not introduce communication overheads. Because of this, we expect improved performance compared to the inexact Newton's method.

Here, both results are presented in the Results section 5.4.1. The parameters for the linear advection equation are taken from Table 5.3 and the ones for the Allen–Cahn equation from Table 5.4. A slight modification in reducing the time step to $\Delta T/100$ produces the presented results.



(a) Speedup

(b) Efficiency

Figure 5.6.: Strong scaling plots for the Allen–Cahn equation with $\varepsilon = 0.01$.

For the linear advection equation, it sometimes lowers the number of global iterations from 5 to 4. This is because the desired tolerance is easier to achieve if the time step is smaller. The trade-off game played here is that, when solving the problem sequentially, the time stepper has to propagate previous solutions L times, which translates to solving a linear system in every time step, yielding a total amount of L system solves. These system solves may take less time when ΔT is smaller, but nevertheless, we still have a baseline sequential method requiring L system solves. When this setup is combined with ParaDiag, a fixed point iteration method, it needs fewer iterations since the solution does not change too much on the time domain. If we look at the convergence Theorem 3.6, where $\xi^{(k)}$ is the error in the k^{th} iteration, we see that the error is bounded as

$$\xi^{(k)} \leq \left(\frac{\alpha}{1-\alpha}\right)^k \xi^{(0)}.$$

When the initial iteration $\vec{\mathbf{u}}^{(0)}$ is filled with initial conditions $\mathbf{u}_0$, then $\xi^{(0)}$ is of order $L\Delta T$. So if instead of ΔT we plug in $\Delta T/100$, the benefits are obvious: we are simply closer to the fixed point solution! A possible bypass of the problem would be to use a relative tolerance as a stopping criterion.

In the case of the Allen–Cahn equation, ΔT plays a more significant role since it damps the explicit terms. The convergence factor from Theorem 3.6 can be expressed as

$$\xi^{(k+1)} \leq \frac{o(\Delta T)L + \alpha}{1-\alpha}\xi^{(k)}.$$

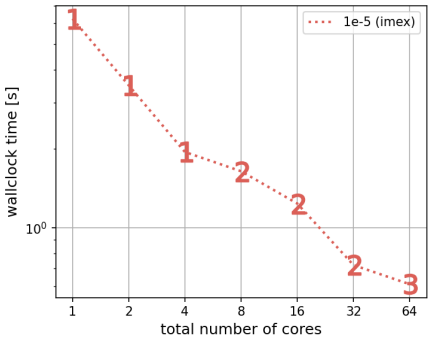


Figure 5.7.: Wall clock time for the Allenc-Cahn equation solved with implicit Euler.

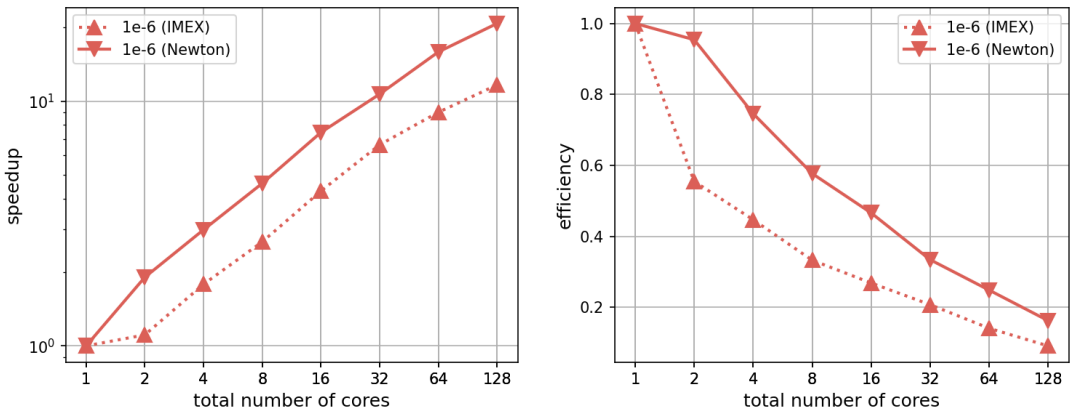
Because of this, the approach that is not even converging for the composite collocation problem iterations (**Test 1** in Subsection 5.4.1) for parallel windows coupling more than $L \geq 16$ time steps (see Figure 5.25), now exhibits significant speedup. The situation becomes more apparent if we look at Figure 5.7 representing

wall clock time with the rounded average number of iterations a parallel window needs to converge. We see that the sequential method, iteratively solving a nonlinear problem in each propagation, needs only 1 iteration to converge. The time step is so tiny that 64 time steps need just 3 iterations in total. Not only did we make the method converge, but we also inflated speedup.

In conclusion, choosing the step size of the method should always be coupled with the actual accuracy we want to reach. It can be challenging to do so since sometimes additional requirements on ΔT have to be satisfied, however, it should be chosen so that the time propagator does not over-resolve in time. Analyzing the method should be done only when the underlining sequential propagator makes sense, and using a tiny time step with a high-order implicit method rarely makes sense.

Setting the tolerance for an inner iterative solver

To analyze ParaDiag, we once again present our findings on the Allen–Cahn equation. The equation is presented in (5.1), and we set $\varepsilon = 1$. The implicit Euler time stepper is used, while the spatial and temporal discretizations are provided in Table 5.5. A solution is accepted when the norm of the residual falls below $\zeta = 10^{-6}$. Figure 5.8 depicts the strong scaling plots for ParaDiag, which was solved using inexact Newton's method and the composite collocation problem iterations.



(a) Speedup

(b) Efficiency

Figure 5.8.: Strong scaling plots for the Allen–Cahn equation with $\varepsilon = 1$.

Using inexact Newton's method results in a speedup of approximately 20, whereas using 128 cores for the composite collocation problem iterations yields a speedup of 11. This outcome represents a significant improvement over the reported results in Figure 5.28a, surpassing them by more than a factor of 2. The parallel efficiency achieved is almost 20%, which is considered to be very favorable for parallel-in-time methods!

Here, a good speedup is achieved because the inner relative tolerance for GMRES is set to $\tau = 10^{-12}$. Although a time-space discretization of order $\zeta = 10^{-6}$ is chosen, we employ over-resolution at each iteration. In a sequential computation, unnecessary precision τ is applied to each time step, resulting in an artificially and/or intentionally slower sequential method. In contrast, ParaDiag benefits from a smaller τ as it mitigates the round-off errors generated by α.

Choosing a meaningful benchmark

To test the effectiveness of ParaDiag, the heat equation is solved with implicit Euler on a time interval of $[0, 0.32]$ and $L = 64$ time steps:

$$u_t = c\Delta u, \quad u(0) = \sin(2\pi x)\cos(2\pi y).$$

The parameters are chosen such that the solution is not over-resolved in time or space, with a tolerance of $\tau = 10^{-6}$ for the inner linear solver and a stopping tolerance of $\zeta = 10^{-5}$. The parameter α is fixed at 10^{-5} for each iteration. The results, as shown in Figure 5.9, achieve a speedup of almost 15, surpassing the reported speedup of below 10 in Figure 5.13a.

The diffusion coefficient in this example is $c = \frac{100}{8\pi^2}$ and the exact solution is

$$u(t, x, y) = e^{-100t}\sin(2\pi x)\cos(2\pi y),$$

a solution that decays to 0 very fast. The algorithm probably solves inaccurately but still converges under the threshold ζ without any problems. Increasing the number of time steps that approach a nearly steady state solution leads to an increase in achievable speedup.

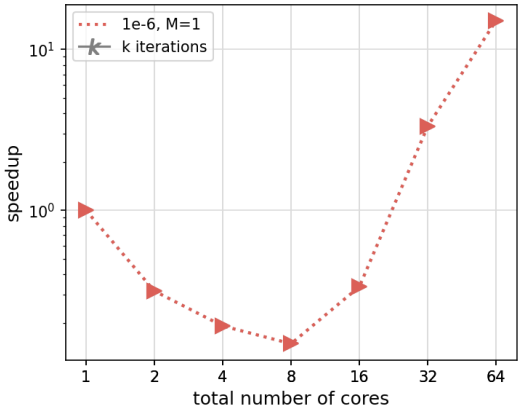


Figure 5.9.: Speedup

Even though the speedup increase is not extremely significant, the problem serves as an example to demonstrate how one has to understand the dynamics of the system in order to present meaningful results.

Optimizing the code

We examine the capabilities of the newly proposed ParaDiag on the nonlinear Boltzmann equation, where we treat the nonlinear collision term explicitly. We consider the equation of form

$$f_t + v_1 f_x = \frac{1}{\varepsilon}Q(f), \quad (t, x, v) \in [0, 1] \times [0, 1] \times [-8, 8]^3. \tag{5.2}$$

The computation of the collision term Q is the most computationally heavy part, and we make use of a `KitBase.jl` library, a `Julia` based code which contains the functions to compute the collision kernel via the Fast Spectral Method (FSM). For purposes of bridging `Python` and `Julia`, we use a `pyjulia` code wrapper which allows us to use variable definitions from both languages inside `Python`.

For the discretization part, we choose a first-order upwind scheme for the spatial discretization and implicit Euler method as a time-stepper, with a step size $\Delta T = 10^{-3}$. We set $\varepsilon = 10^{-3}$ and choose 384 points for the spatial discretization and $72 \times 36 \times 36$ points for the velocity space. The stopping criterion is $\|\vec{\mathbf{res}}^{(k)}\|_\infty \leq 10^{-4}$ and the linear solver tolerance is $\tau = 10^{-6}$.

First, we scale the problem sequentially in time but parallel in space for $L = 32$ time steps (the `petsc` line in Figure 5.10). On top of the spatial parallelization, we deploy our time-parallel IMEX-based Jacobian-free ParaDiag method for nonlinear equations. In Figure 5.10, `petsc(n)` means that a `n` cores are used for the parallelization of the spatial problem for each time step.

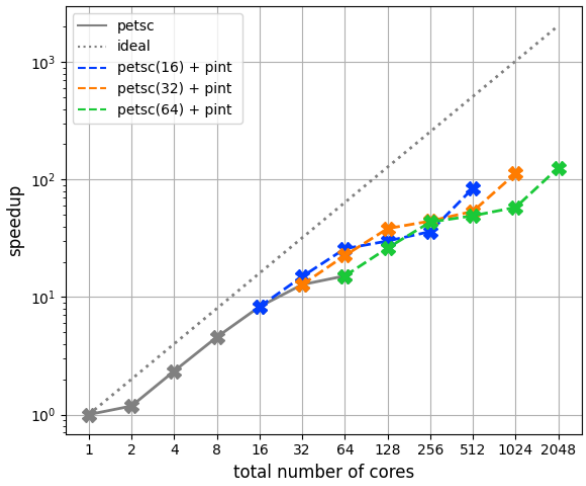


Figure 5.10.: Speedup for the Boltzmann equation using `pyjulia`.

We can see the terrific performance of our time-parallel integrator and how it continues scaling after spatial parallelism is saturated. The overall speedup achieved is approximately 126, twice the speedup obtained in the experiment shown in Figure 5.31a, an exceptional outcome when dealing with nonlinear hyperbolic equations!

What needs to be presented here is that the spatial scaling narrowed down the runtime of a simulation from 2526 seconds to around 166 seconds, and the additional parallelization in time reduced the total runtime to just 20 seconds. The experiment's setting is the same as in Subsection 5.4.2, where the execution time of the sequential run took almost 400 seconds less. Looking at the efficiency in Figure 5.11, things start to look suspicious: one can even conclude that the time-parallel integration could be more efficient than the spatial one!

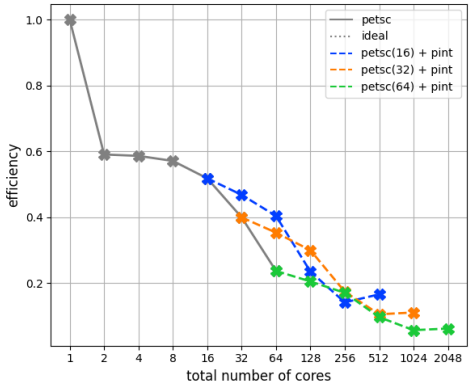


Figure 5.11.: Efficiency for the Boltzmann equation using `pyjulia`.

Unfortunately, the code wrapper does many silent things, including having a small overhead each time a `Julia` variable is used inside `Python`. The reason for this is the conversion between the variable types. The computation of the collision parameter should be an embarrassingly parallelizable process since the FSM acts on each entry of the spatial vector $\mathbf{x} = [x_1, \ldots, x_N]^\top$. Table 5.1 shows a small scaling example of the computation of the collision kernel, where the FSM method using 2 cores is called on one half of the vector $\mathbf{x}$ on each core.

We see that the computation of the collision operator, even on a smaller mesh, does not scale well. Using an inefficient code as a base plays in favor of time-parallel integration since these tiny constant overheads are accumulated during runtime and become significant. Time-parallel integration profits from these artificial overheads since they become smaller when more cores for time parallelization are used!

cores	`pyjulia`	`Julia`
1	2.5 s	1.95 s
2	1.5 s	1.03 s
efficiency on 2 cores	83%	95%

Table 5.1.: The time needed to compute the collision kernel for 100 spatial points and $48 \times 24 \times 24$ velocity points using the `pyjulia` wrapper and `Julia`.

5.2.2. Setting up the right test cases

Benchmarking parallel-in-time algorithms is often a challenging task. The heat equation, a parabolic diffusive equation that usually exhibits 'good' behavior, is a commonly used example equation. However, testing the method on different equations with various nonlinearities is crucial. The initial step in benchmarking is to have a validated parallel implementation. The number of iterations required to achieve convergence may or may not provide valuable performance estimates. Once an implementation exists, a set of example equations for test cases is selected. A problem class is created for each equation, choosing a stable space-time discretization. In some instances, additional conditions must be satisfied, further complicating the selection of a suitable test case.

Assuming that we have a valid time-space discretization that meets all our conditions, the first step in examining parallel performance is to ensure that we have the best sequential time-stepping method available. The method must solve for each propagation only to the necessary precision, which may not require an exact solver. We must then carefully select the number of discretization points. If there are too few, we will not achieve the desired precision, whereas too many will result in over-resolving in space or time. We must also establish a measure of how far we are from an exact solution of the equation. If the analytical solution is known, we can use it. Otherwise, we must compute it using a very fine discretization and consider it the 'exact' solution.

Ideally, we want to present how well our method performs depending on the accuracy we want our solution to have. Usually, we have an idea of how precise we want to solve a given equation, therefore, a stopping criterion is chosen first, following a discretization with some order that can reach a given tolerance ζ. The tolerance ζ is usually not much smaller than the wanted accuracy. It rarely makes sense to solve aGa discretization in space and time of order 10^{-3} to machine precision, and as seen in the previous section, this may lead to overperformance. However, solving to a precision $\zeta = 10^{-5}$ and solving to a precision $\zeta = 10^{-12}$ does not have to exhibit the same speedup behavior. Because of this, we choose three different tolerances $\zeta \in \{\zeta_1, \zeta_2, \zeta_3\}$ to test for.

To adapt the discretization error to the threshold ζ, we also adjust the time stepper by changing the number of collocation nodes M in the $\mathbf{Q}$ matrix. Thus, the tolerance is coupled to M, simply because there is no point in achieving low errors without deploying a higher-order method in space and time [78]. The nodes in use originate from the Gauss–Radau quadrature, with the right endpoint included. This is equivalent to using a high-order implicit Runge–Kutta scheme with an order of $2M - 2$. The underlying sequential time-stepping method solves the collocation problem via diagonalization of the $\mathbf{Q}$ matrix. In all of the test runs, ΔT is chosen so that the error of the stable time-stepping method satisfies the expected discretization order.

The linear solver used is a GMRES (without a preconditioner) implemented in `petsc4py`. The relative stopping tolerance τ is set for the linear solver. An advantage of using an iterative solver in a sequential run is that τ can be just a bit smaller than the desired threshold ζ we want to reach on our domain,

otherwise, it would unnecessarily prolong the runtime. However, this is not the case when choosing a relative tolerance $\widetilde{\tau}$ for the linear solver within our method for inner systems (see Algorithm 2, Figure 2.2 and Section 4.4). On the one hand, the method enormously benefits from having a small $\widetilde{\tau}$ in general since it plays a vital role in the convergence, visible from Lemma 2.11, but on the other hand, the linear solver needs more iterations, thus execution time $T_{\text{sol,par}}$ is higher, see Section 4.4. As a result, this is indeed a drawback when using the parallel method and has to be kept in mind.

We disabled multithreading which is usually silently triggered by `numpy`. A vector filled with initial conditions $\mathbf{u}^{(0)} = (\mathbf{u}_0, \ldots, \mathbf{u}_0)$ is chosen as guess for the initial iteration. Note that the timings do not include the startup time, the setup, nor the output times.

In the interest of reproducibility, every parameter selection for each test case is explained and justified. The study presents wall clock times, speedup, and efficiency plots as performance measures. Hopefully, some setup examples presented here will contribute to a set of standard benchmarks in the future.

5.3. ParaDiag for linear equations

The linear heat equation is one of the most standard benchmark equations for time-parallel integration. It is a parabolic-type equation describing the process of diffusion.

The second standard test case is the linear advection equation. As mentioned in the introduction (Chapter 1), it is a notoriously difficult benchmark for time-parallel integration methods since most methods rely on coarsening. Since ParaDiag coupled with the collocation method is a single-level method, it is essential to test the limits for this standard hyperbolic-type test case since single-level methods seem like a promising approach for hyperbolic-type equations.

The equations were chosen in a way that their analytical solutions are known in order to compute the difference between the approximate solution and the exact one, simultaneously checking that the chosen discretization orders are accurate. In both the linear heat and the linear advection equation, parallel runtimes of the method for three different thresholds $\zeta = 10^{-5}, 10^{-9}, 10^{-12}$ are measured. The approximation $u^{(k)}$ satisfies $\|u(T_0 + \Delta T L) - u_L^{(k)}\|_\infty \leq \zeta$, which is monitored by values m_k from Algorithm 2 in addition to monitoring the absolute error between consecutive iterates. The parallel-in-time solver used is the one with the α-adaptive strategy described in Algorithms 1 and 2. All the benchmarks are performed according to the guidelines outlined in Section 5.2.2.

5.3.1. Heat equation

The heat equation is a partial differential equation that describes how heat flows through a material over time. Our test example for the heat equation is

$$u_t = \Delta u + \sin(2\pi x)\sin(2\pi y)(8\pi^2 \cos(t) - \sin(t)),$$
$$(t, x, y) \in [\pi, \pi + T] \times [0, 1]^2,$$

with the exact solution

$$u(t, x, y) = \cos(t)\sin(2\pi x)\sin(2\pi y).$$

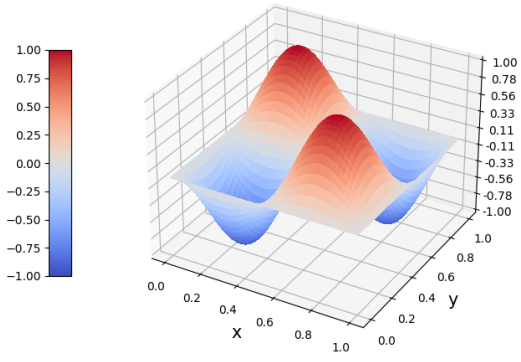


Figure: $u(\pi, x, y)$

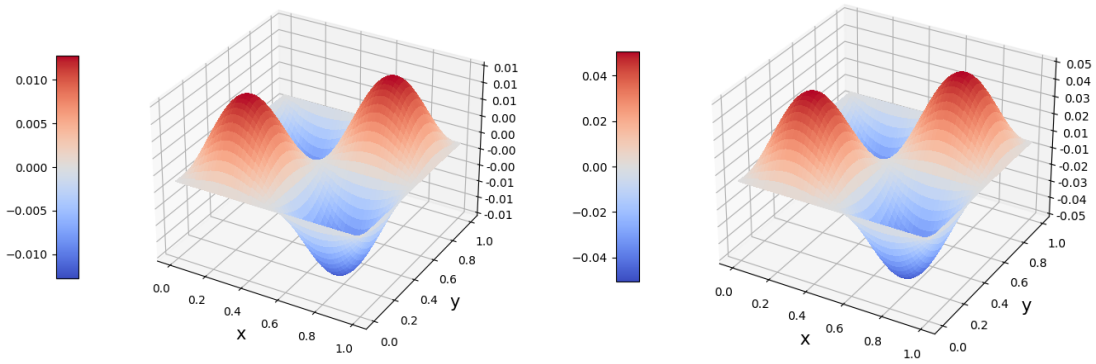
This equation has periodic boundary conditions, which were used to form the discrete periodic Laplacian with central differences bringing the equation into the generic form (2.9). The additional forcing term ensures our equation does not reach a steady state solution.

The idea is to compare the execution times of the method covering $L = 64$ time steps, solved in parallel across steps and parallel across all time points: across all collocation nodes for all time steps. The time domain is set to $[\pi, \pi + T_{64}]$, for a given threshold ζ, where the length of the interval T_{64} is chosen so that a desired accuracy can be reached with a step size $\Delta T = T_{64}/64$. Table 5.2 presents the used runtime setup. The benchmarks were performed 10 times to get a better runtime estimate since some runtimes were short.

tolerance to reach ζ	10^{-5}	10^{-9}	10^{-12}
no. of spatial points N	350	400	350
order in space κ	2	4	6
no. of collocation points M	1	2	3
time endpoint T_{64}	0.32	0.16	0.16
linear solver tolerance τ	10^{-6}	10^{-10}	10^{-13}
linear solver tolerance $\widetilde{\tau}$	10^{-6}	10^{-10}	10^{-13}

Table 5.2.: Parameter choice for the heat equation to reach an error $\|\mathbf{u}(T_{64}) - \mathbf{u}_{64}\|_\infty < \zeta$ when solving with a standard sequential approach. Here, κ denotes the discretization order in space, a centered difference scheme for the discrete Laplacian, see [5]. T_{64} represents the interval length that is needed so that the error is below ζ after 64 time steps for a given discretization.

Figures 5.12a and 5.12b plot the error $u(\pi + T_{64}, x, y) - u(\pi, x, y)$, for $T_{64} = 0.32, 0.16$, respectively. The plots show that the solution evolves in time far more than the tolerances ζ in Table 5.2 are set to, justifying the parameter choices.

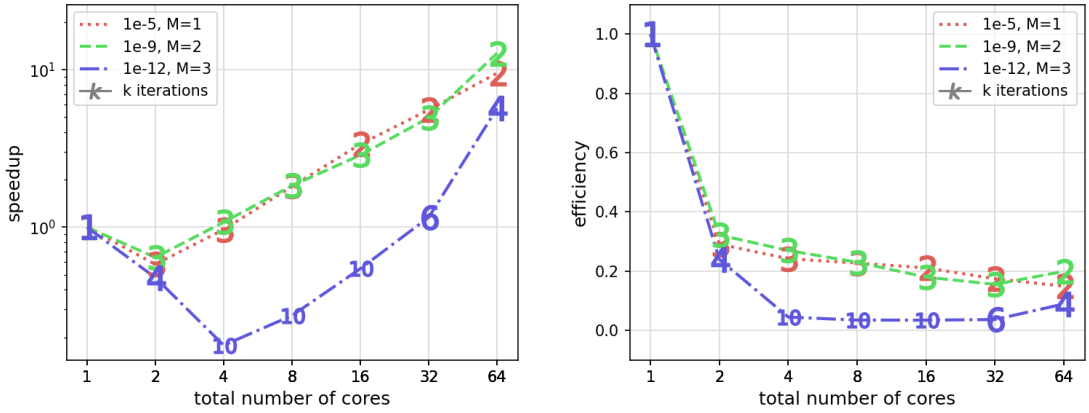


(a) $u(\pi + 0.16, x, y) - u(\pi, x, y)$

(b) $u(\pi + 0.32, x, y) - u(\pi, x, y)$

Figure 5.12.: $u(\pi + T_{64}, x, y) - u(\pi, x, y)$ for $T_{64} = 0.32,\ 0.16$.

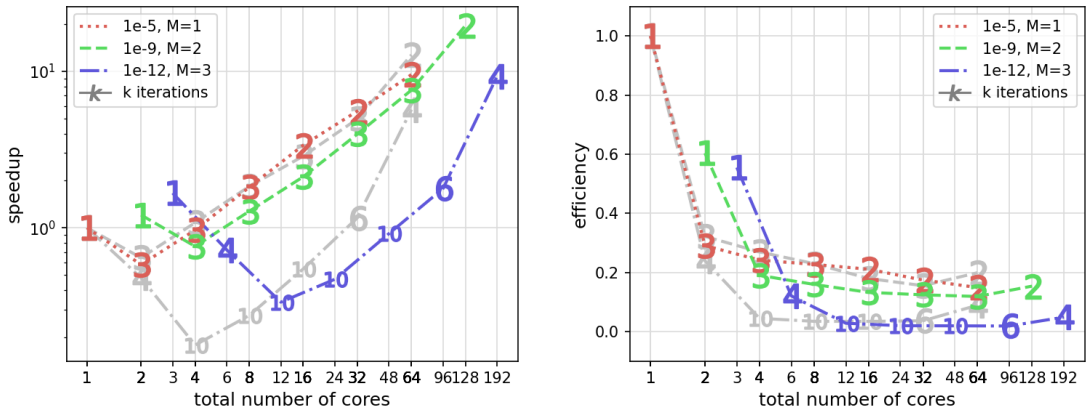
The strong scaling across time steps is presented in Figure 5.13. The equation is solved with a parallelization strategy where the total number of processors $n_{\text{proc}} = n_{\text{step}}$ is used to compute one moving window of coupled time steps. There is no other parallelization, meaning $n_{\text{coll}} = n_{\text{space}} = 1$. This is a setting where the diagonalization of the preconditioner is handled in parallel, whereas the inner systems are solved on only one core via diagonalization of $\mathbf{Q}\mathbf{G}_\ell^{-1}$. The starting value for m_0 for Algorithm 2 is $m_0 = 64T_{64}/n_{\text{step}}$.



(a) Speedup (b) Efficiency

Figure 5.13.: Strong scaling plots for the heat equation, solved in parallel accross time steps. The numbers on the curves represent the number of outer iterations ParaDiag needs to reach the given tolerance.

Given more cores, we add an additional layer of parallelism, across the collocation nodes. The parallelization strategy is then $n_{\text{space}} = 1$, $n_{\text{coll}} = M$, where n_{step} cores are used to handle the moving window, and $n_{\text{coll}} = M$ cores handle the collocation problem, yielding a total number of cores $n_{\text{proc}} = Mn_{\text{step}}$. In this setting, all the collocation nodes across all time steps are handeled in parallel and the strong scaling plots are presented in Figure 5.14.

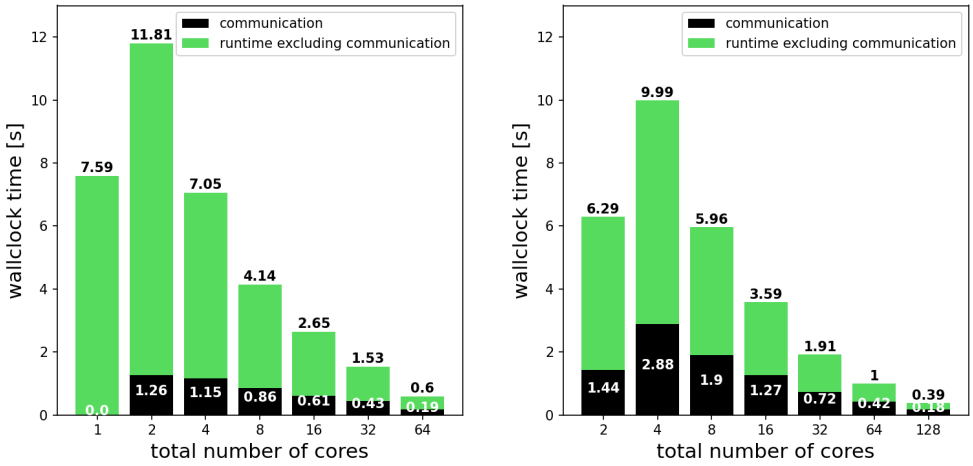


(a) Speedup (b) Efficiency

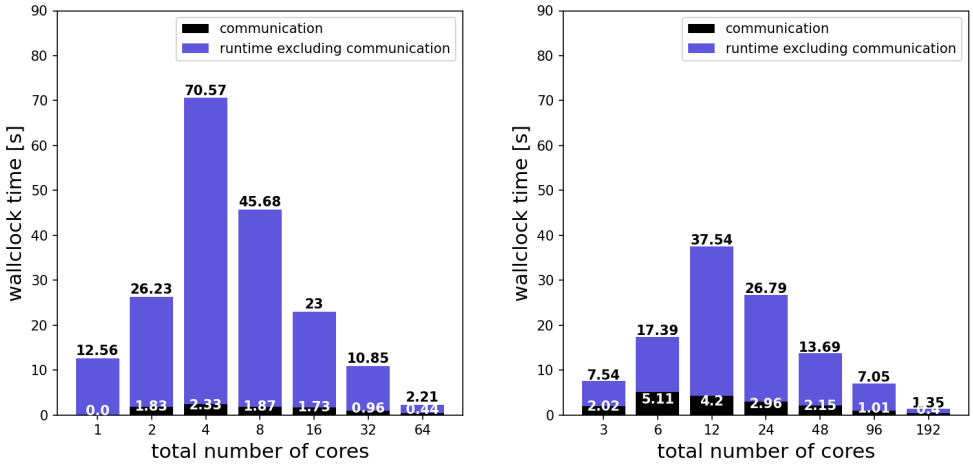
Figure 5.14.: Strong scaling plots for the heat equation, solved in parallel across time steps and the collocation nodes. The numbers on the curves represent the number of outer iterations ParaDiag needs to reach the given tolerance. The gray plots are the same as in Figure 5.13 and serve as a reference.

Overall, additional parallelism across collocation nodes gives additional speedup, at least a factor of two for the fully parallel runs. If we observe the speedup for the runs that are sequential over time steps but parallel accross collocation nodes (marked with 1 iteration for 1, 2 and 3 cores in Figure 5.14a), we see that some initial speedup is possible, however, not as efficient as one would expect. The efficiency is already reduced to $40 - 45\%$ when additional $n_{\text{coll}} = 2, 3$ cores are used. One reason is that the

system solves after diagonalizaion of the $\mathbf{Q}\mathbf{G}_\ell^{-1}$ matrix have different times-to-solution. In case $M = 3$, the slowest system solve is around $0.078s$ while the fastest one is $0.056s$, leading to a load imbalance among the cores. As a result, the cores have to wait for the core with the slowest system solve, producing overheads and, in the end, being less efficient.



(a) $n_{\text{coll}} = 1$, $M = 2$ (b) $n_{\text{coll}} = 2$, $M = 2$



(c) $n_{\text{coll}} = 1$, $M = 3$ (d) $n_{\text{coll}} = 3$, $M = 3$

Figure 5.15.: Communication overheads for different parallelization strategies for the heat equation.

When very accurate results are needed (blue curves), the performance is degrading. However, we have multiple runs with up to 10 iterations. This is because the linear system shifts that are produced by the diagonalization procedure are poorly conditioned for the particular choice of the value α_k. Manually picking this sequence with slightly different larger values can circumvent this issue to a certain degree. However, this is not presented here, because it is also important to show that the method does not always perform well when used without manually tweaking the parameters. A more in-depth study of this phenomenon is needed and left for future work. Despite the drawbacks, the scaling results are strong and additional parallelism for the collocation problem increases speedup.

The total overhead of the inner system solves can be roughly understood by plotting the total runtime where the communication overhead is also presented. In Figure 5.16 and 5.15, where the colors of the

plots correspond to the already mentioned three runs, we can see that when using parallelism accross the collocation nodes, the communication time grows. In the case when $M = 2$, in Figures 5.15a and 5.15b, the communication overhead is worth it and we see that it starts dominating as we approach a full parallel run (all time steps solved in parallel). This is not the case for the example with $M = 3$ collocation nodes, because, as already mentioned, the issue are the solver iterations that do not converge. However, when ParaDiag does converge, the communication time takes up about 20% when solving in parallel accross time steps and 30% when solving in parallel accross all time points, becoming quite significant, however, not nearly enough to bring the efficiency down to 5%. The major loss of efficiency is the price of the shifted systems.

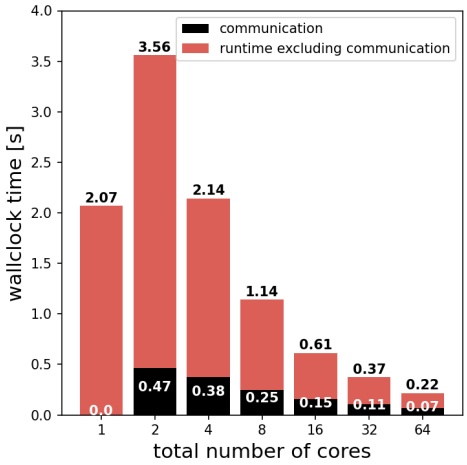


Figure 5.16.: Communication and system solving overheads in the diagonalization process across time steps with $M = 1$ collocation node (implicit Euler method) for the heat equation.

5.3.2. Advection equation

The second equation for testing the performance of ParaDiag is the linear advection equation, a hyperbolic type equation. Advection is a process that describes a movement of some quantity through a fluid or a gas. The equation is governed by

$$u_t + u_x + u_y = 0,$$
$$(t, x, y) \in [0, T] \times [0, 1]^2,$$

with an exact solution

$$u(t, x, y) = \sin(2\pi x - 2\pi t)\sin(2\pi y - 2\pi t).$$

An upwind scheme, incorporating the periodicity on the boundaries is used, bringing the equation into the generic form (2.9). The space discretization is then combined with a high order implicit time-stepping method, defined through the **Q** matrix.

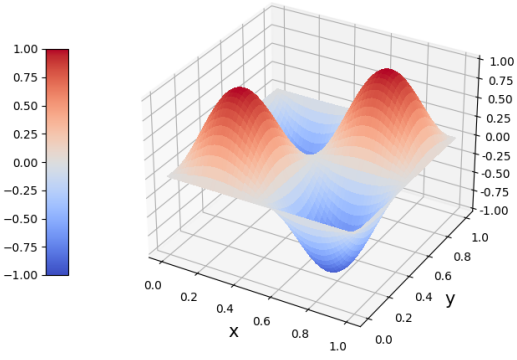


Figure: $u(0, x, y)$

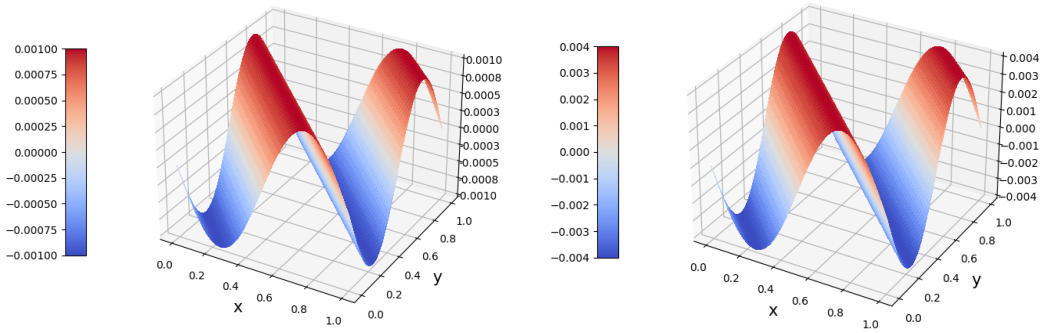
Similarly as for the heat equation, the execution times of the method covering $L = 64$ time steps are compared. Table 5.3 summarizes the parameters in use for each benchmark and Figure 5.17 plots the error $u(T_{64}, x, y) - u(0, x, y)$, for two different end times T_{64}. The solution advances enough in the chosen time domain, justifying the parameter choices.

The first strong scaling plots are given in Figure 5.18. The scaling is performed accross time steps only, corresponding to a parallelization strategy with $n_{\text{proc}} = n_{\text{step}}$ and $n_{\text{coll}} = n_{\text{space}} = 1$ for each moving window. After diagonalizing the preconditioner, the inner systems are solved again via diagonalization of $\mathbf{Q}\mathbf{G}_\ell^{-1}$, where the diagonal systems are then solved in a sequential manner on one core. The starting value for m_0 for Algorithm 2 is $m_0 = 10T_{64}/n_{\text{step}}$.

Figure 5.19 contains the scaling plots with additional parallelism across the collocation nodes. The setup in this example uses $n_{\text{proc}} = Mn_{\text{step}}$ cores for each moving window of ParaDiag, with $n_{\text{coll}} = M$ additional cores used for solving the linear systems after diagonalizing $\mathbf{Q}\mathbf{G}_\ell^{-1}$.

tolerance to reach ζ	10^{-5}	10^{-9}	10^{-12}
no. of spatial points N	800	800	700
order in space κ	1	3	5
no. of collocation points M	1	2	3
time endpoint T_{64}	0.00016	0.00064	0.0128
linear solver tolerance τ	10^{-6}	10^{-11}	10^{-14}
linear solver tolerance $\widetilde{\tau}$	10^{-9}	10^{-13}	10^{-15}

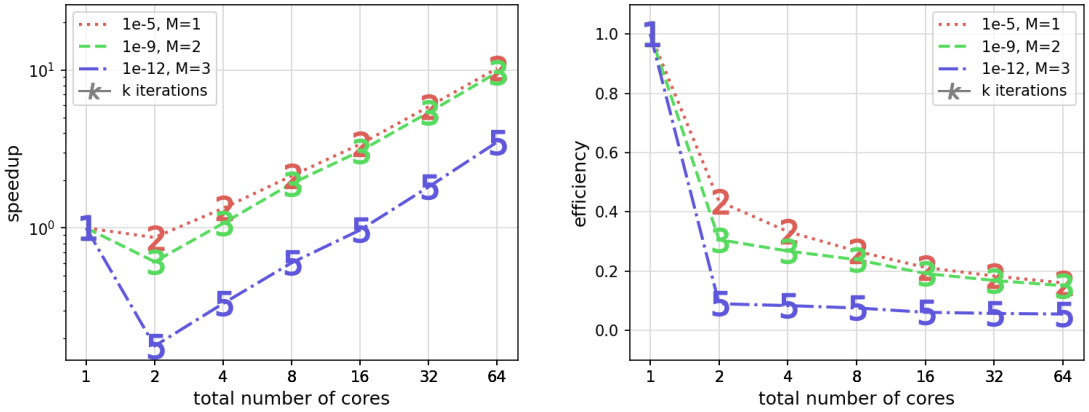
Table 5.3.: Parameter choice for the advection equation to reach an error $\|\mathbf{u}(T_{64}) - \mathbf{u}_{64}\|_\infty < \zeta$ when solving with a standard sequential approach with $L = 64$ time steps. Here, κ denotes the discretization order in space for the upwind scheme.



(a) $u(0.00016, x, y) - u(0, x, y)$

(b) $u(0.00064, x, y) - u(0, x, y)$

Figure 5.17.: Plotting $u(T_{64}, x, y) - u(0, x, y)$, for $T_{64} = 0.00016, 0.00064$.

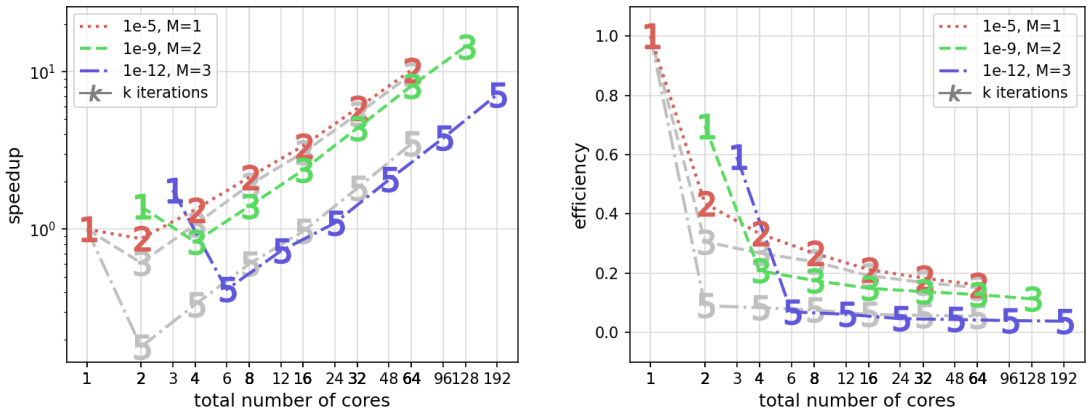


(a) Speedup

(b) Efficiency

Figure 5.18.: Strong scaling plots for the advection equation, solved in parallel accross time steps. The numbers on the curves represent the number of outer iterations ParaDiag needs to reach the given tolerance.

The additional cores provide around a factor of two additional speedup, similarly to what we have seen for the heat equation benchmarks. The efficiency cost of deploying additional cores to handle the diagonalization of the $\mathbf{Q}$ matrix (the runs with 1 iteration in Figure 5.19b for $n_{\text{proc}} = 1, 2, 3$) is around $30 - 40\%$. The method seems to be more efficient for the advection equation than for the heat equation even though the imbalances in the inner system solves are even larger, ranging from $0.25s$ to $1.44s$ (for $M = 3$). Despite that, ParaDiag is more efficient for the advection equation because the inner systems are not as poorly conditioned. Given the fact that the advection equation is of hyperbolic type, the obtained results are great for a time-parallel integrator when compared to MGRIT [33].



(a) Speedup

(b) Efficiency

Figure 5.19.: Strong scaling plots for the advection equation, solved in parallel accross time steps and the collocation nodes. The numbers on the curves represent the number of outer iterations ParaDiag needs to reach the given tolerance. The gray plots are the same as in Figure 5.18 and serve as a reference.

We observe degrading performance when reaching lower tolerances. The overall time spent on solving linear systems can best be presented through plots in Figures 5.20 and 5.21, where the colors of the bars are linked to the corresponding setups from Table 5.3. In practice, the amount of time spent on communication grows with the number of cores, however, by the design of our strong scaling approach, all the communication times and load imbalances of all windows in all iterations are summed up, making ParaDiag worth to use as a windowed approach only when more than a certain amount of time steps are coupled. The amount of time steps that need to be coupled to gain speedup, depends on the number of ParaDiag iterations, which on the other hand, depends on the tolerance we want to achieve. This is an important result, since ParaDiag is very likely not a method that can converge with a too large number of parallel time steps. The reason behind this is that, even though the contraction factor from Theorem 2.9 guarantees convergence for linear equations,

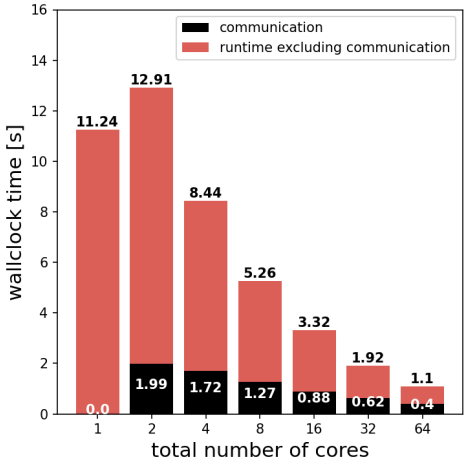
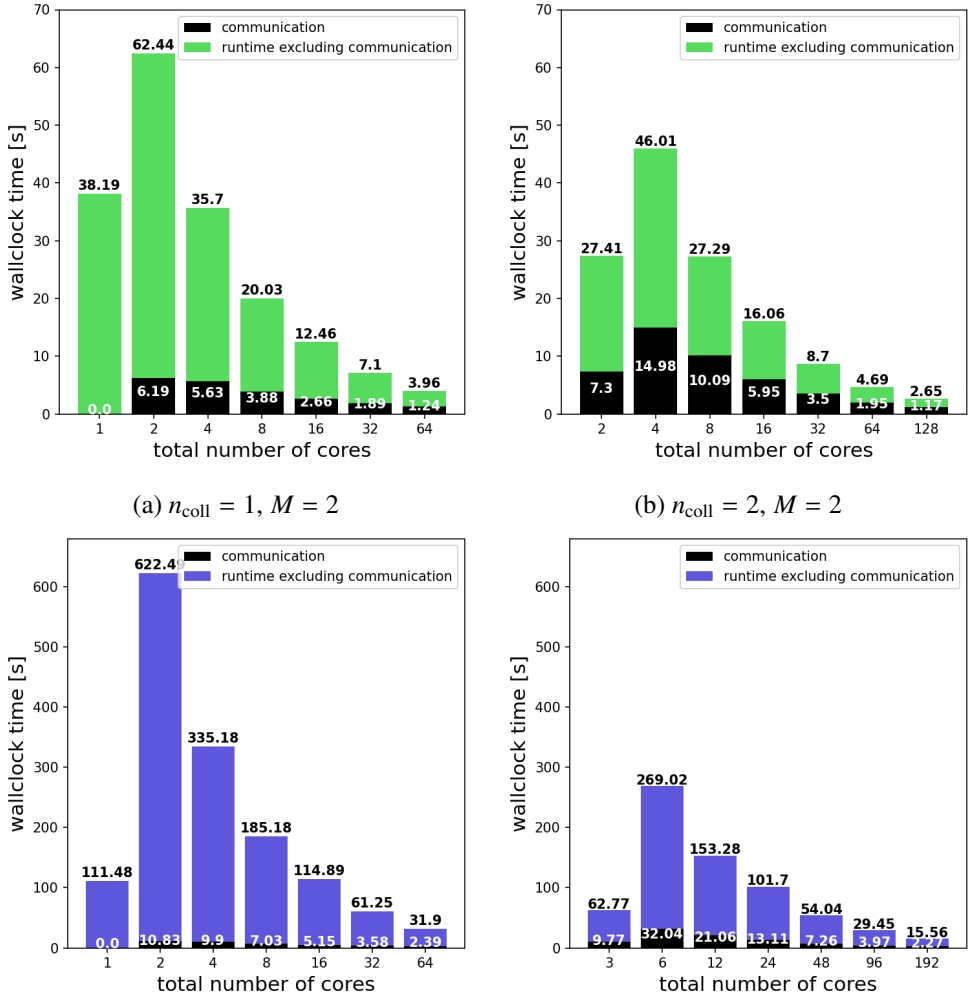


Figure 5.20.: Communication and system solving overheads in the diagonalization process across time steps with $M = 1$ collocation node (implicit Euler method) for the advection equation.

the round-off error analysis from Theorem 2.12 allows errors to grow proportionally to the number of parallel time steps. Consequently, there may exist problems that do not converge due to round-off errors, given enough time steps.



(a) $n_{\text{coll}} = 1$, $M = 2$

(b) $n_{\text{coll}} = 2$, $M = 2$

(c) $n_{\text{coll}} = 1$, $M = 3$

(d) $n_{\text{coll}} = 3$, $M = 3$

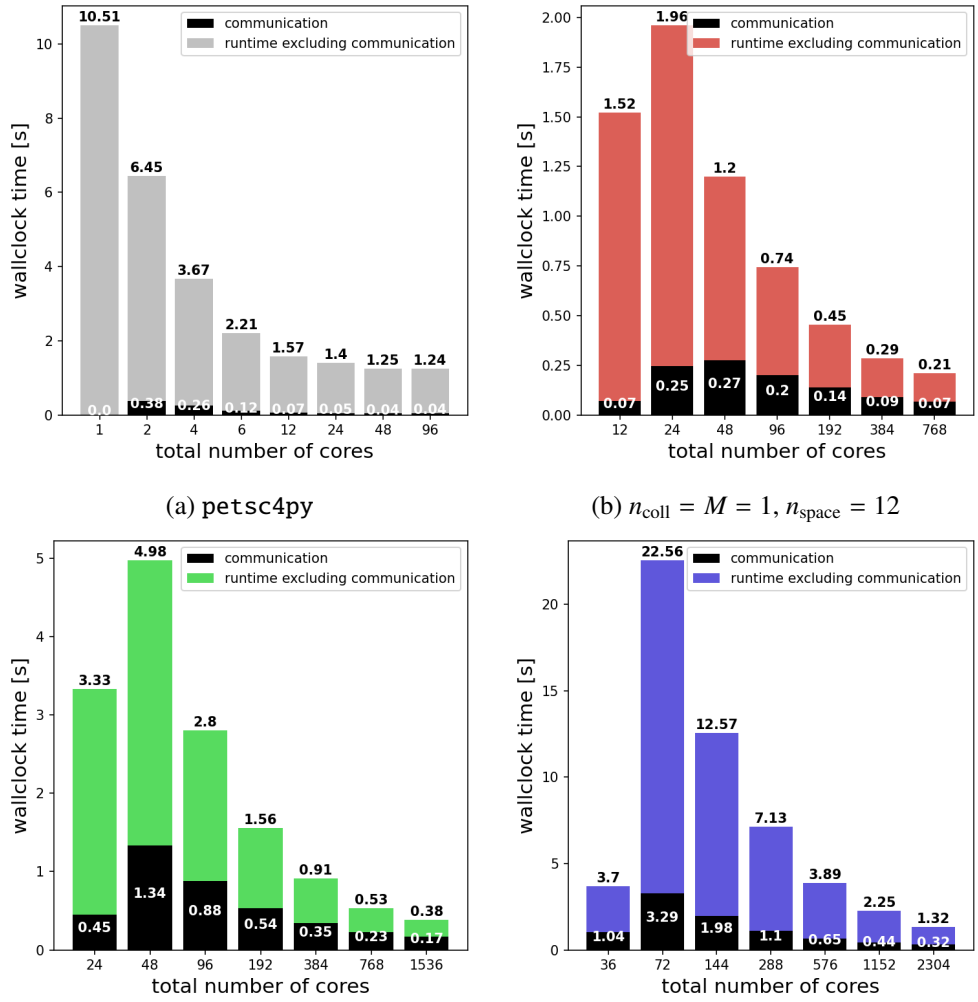
Figure 5.21.: Communication overheads for different parallelization strategies for the advection equation.

Spatial scaling

Parallel-in-time integration methods are ideally used in combination with a space-parallel algorithm, especially in the field of PDE solvers. Therefore, we test the method together with `petsc4py`'s parallel implementation of GMRES for the advection equation.

First, `petsc4py` was scaled using up to 96 cores. This was done by solving sequentially in time with implicit Euler on $L = 64$ time-steps. The number of cores $n_{\text{space}} = 12$ is chosen as the last point where `petsc4py` scaled reasonably well for this problem size. After fixing the spatial parallelization, the double time parallelization is layered on top of that. The strong scaling across all collocation nodes and time-steps is repeated for $n_{\text{step}} = 1, 2, \ldots, 64$ parallel windows for three different thresholds $\zeta = 10^{-5}, 10^{-9}, 10^{-12}$. For convenience, we changed the number of points per dimension in space to $N =$

768, 768, 702 for spatial orders $\kappa = 1, 3, 5$, respectively, so that N^2 is divisible the number of cores we scale with. Other parameters are the same as in Table 5.3 and our runs still reach below the wanted tolerance. The complete parallelization trategy is $n_{\text{proc}} = 12Mn_{\text{step}}$.



(a) `petsc4py`

(b) $n_{\text{coll}} = M = 1$, $n_{\text{space}} = 12$

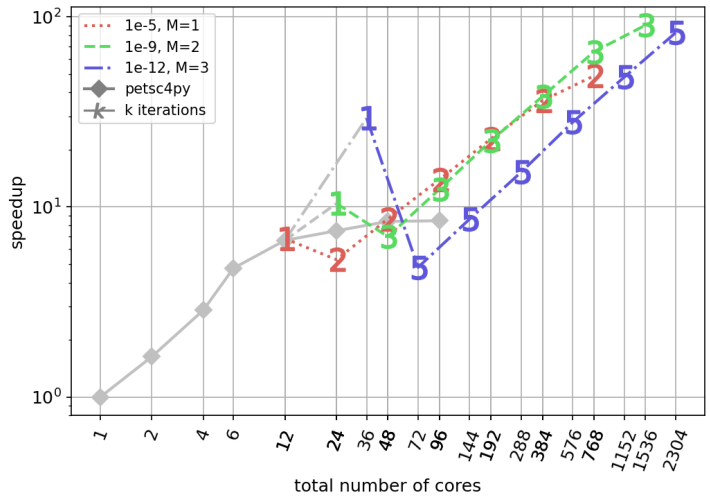
(c) $n_{\text{coll}} = M = 2$, $n_{\text{space}} = 12$

(d) $n_{\text{coll}} = M = 3$, $n_{\text{space}} = 12$

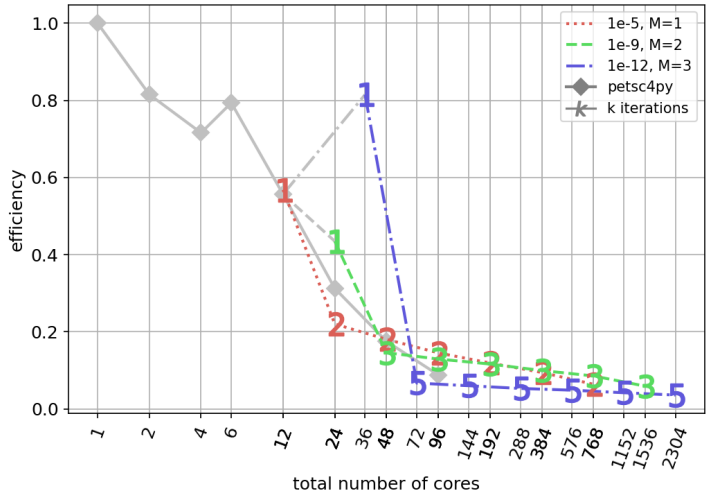
Figure 5.22.: Communication overheads when solving the advection equation parallel in space and time.

The results from Figure 5.23 clearly show that, by using ParaDiag with the collocation problem, we can get significantly higher speedups for a fixed-size problem than when using a space-parallel solver only. In the best case presented here, we obtain a speedup of about 85 over the sequential run. We would like to emphasize here that all runs are done with realistic parameters, not over-resolving in space, nor time, nor in the inner solves [74]. Thus, while the space-parallel solver gives a speedup of up to about 8, we can get a multiplicative factor of more than 10 by using a space- and doubly time-parallel method.

Figure 5.22 shows the communication overheads for the purely spatial parellel run (gray) and runs with fixed number of $n_{\text{coll}} = 12$ cores for spatial parallelization combined with a parallel-in-time run. We can see that utilizing more cores in space helps lower the communication costs even more because the chunks of memory being communicated are overall smaller.



(a) Speedup



(b) Efficiency

Figure 5.23.: Strong scaling plots for the advection equation and three thresholds. The solid gray line represents the spatial scaling with `petsc4py` for the advection equation solved sequentially with implicit Euler on 64 time steps. The curve shows the scaling of `petsc4py` for our problem is best around 12 cores, since using more cores does not increase the speedup significantly. The colored lines represent the scaling for the moving windows with $n_{\text{coll}} = M$ and $n_{\text{space}} = 12$ cores, in other words, parallelism across time-steps, across the method, and in space. The numbers on the curves represent the number of outer iterations Algorithm 1 needs in order to reach the given tolerance.

It is important to note that first the spatial parallelisation should be exhausted, then parallelization across the collocation points, and lastly, the parallelization across time steps. This is simply because the efficiency gets worse in that order. In case when we do not care about efficiency and a lot of cores are available, the parallel-in-time scaling of ParaDiag coupled with the collocation problem can provide considerable speedups for linear equations.

5.4. ParaDiag for nonlinear equations

First, we present the test result for the nonlinear Allen–Cahn equation, one of the standard benchmarks in the parallel-in-time community, solved via the inexact Newton's method and the semi-implicit iterations where the nonlinear right-hand side is handled explicitly. This test aims to compare the two methods in terms of scaling and time-to-solution in different settings.

The second example is the nonlinear Boltzmann equation, a hyperbolic-type equation. Because the right-hand side of the equation is an integral expression, the Newton's method is unfeasible. This example aims to highlight the use case of the Jacobian-free iterations with the method that shows excellent scaling results for the linear advection equation. So far, this test case has never been used since the nonlinear hyperbolic structure is known to pose difficulties to time-parallel integration.

The benchmarks are performed with a fixed value $\alpha = 10^{-8}$ within the iterative refinement iterations, defined in Algorithm 3. The global iterations (3.18) terminate when the norm of the residual is below a certain threshold, $\|\vec{\mathbf{res}}^{(k)}\|_\infty < \zeta$. For the presentation of the results for the Boltzmann equation, a `Julia`-based library `KitBase.jl` is used to generate an output file for out `Python`-based implementation. All the benchmarks are performed according to the guidelines outlined in Section 5.2.2.

5.4.1. Allen–Cahn equation

The Allen–Cahn equation is one of the well-known test examples in the PinT community. It is a reaction-diffusion equation that describes the process of phase separation:

$$u_t = \Delta u + \frac{1}{\varepsilon^2} u(1 - u^2), \quad (t, x) \in [0, T] \times \mathbb{R}^2.$$

The initial condition

$$u(0, x) = \tanh\left(\frac{R - \|x\|_2}{\sqrt{2}\varepsilon}\right)$$

describes a circle with a radius R and an interface of width $o(\varepsilon)$. The circle shrinks over time, and in the sharp interface limit $\varepsilon \to 0$, the radius can be expressed as $r(t) = \sqrt{R - 2t}$ [39]. This means that the simulation is short-lived, and the maximum simulation time for parallel-in-time algorithms should not allow $T > R/2$ [74]. For the spatial domain, we use $[-2R, 2R]$ and periodic boundary conditions.The aim is to measure the time-to-solution when solving for three different stopping tolerances. The condition $\Delta T < \varepsilon^2$ is satisfied in all of the runs [96, Proposition 3.1 and Theorem 3.3]. Following the discretization observations from paper [79], we know that the local truncation error of the implicit Euler method for the Allen–Cahn equation is $O(\Delta T^2/\varepsilon^3)$. We use this estimate to set the truncation errors in time, so that the errors in time and space are of a fixed order ζ. The accuracy of the solution is verified by comparing it to an 'exact' solution – the same equation numerically solved on a much finer mesh. The error to the fine 'exact' solution is of order of magnitude as the residual itself.

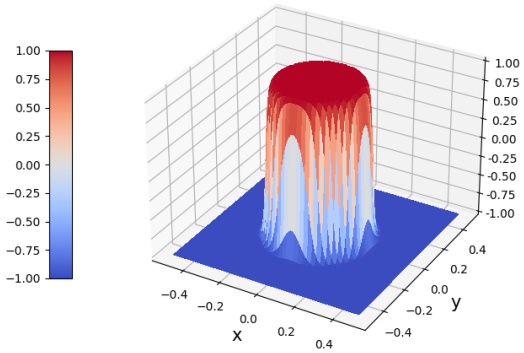


Figure: $u(0, x)$

For Paradiag, the Laplacian is treated implicitly $f_I(u) = \Delta u$. For the inexact Newton's approach, the averaged Jacobian is incorporated in the implicit part in every iteration.

Test 1: The idea of this test is to compare the inexact Newton's methods and the composite collocation problem iterations by solving the same equation on the same time-space domain for three different thresholds ζ. For this test, we choose $\varepsilon = 0.01$, $R = 0.25$ and $T = 0.003$. Even though the Lipschitz constant seems to be very large, and the estimate from the convergence Theorem 3.6 is larger than 1, the inexact Newton's approach still converges for $L = 64$[1] time steps. For each tolerance ζ, we choose a different discretization order in space κ and the number of points N for the discrete Laplacian [5]. $M = 2$ collocation points for the $\mathbf{Q}$ matrix are used, however, the scaling is performed only accross time teps. The mentioned parameters can be summarized in Table 5.4.

Figure 5.24 shows how much does the solution move over time as $u(0.003, x) - u(0, x)$. The amplitude of the difference is very steep and large in the areas the circle shrunk, resembling a cilinder. Because of this, the choice of $T = 0.003$ is reasonable even though it may at first seem as the solution would not significantly progress.

tolerance to reach ζ	10^{-5}	10^{-9}	10^{-12}
no. of spatial points N	320	180	120
order in space κ	2	4	6
linear solver tolerance τ	10^{-7}	10^{-11}	10^{-14}

Table 5.4.: Parameter choice to reach an error $\|\vec{\mathbf{res}}^{(k)}\|_\infty < \zeta$ in **Test 1**.

Runtimes for scaling accross time steps are presented in Figure 5.25. We can see that the inexact Newton's method is faster when it comes to time-to-solution, however, both methods seem to have roughly the same scaling, see Figure 5.26a. Newton's method being faster confirms our intuition from Remark 3.8, and it is heavily due to the number of iterations the parallel runs need in order to converge. We also see that the composite collocation problem iterations break down for windows with $L \geq 16$ parallel time steps.

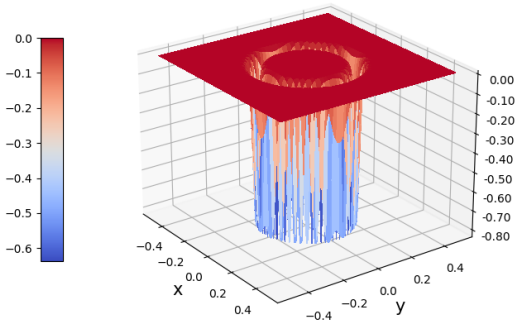


Figure 5.24.: $u(0.003, x) - u(0, x)$

Another important observation is that the number of iterations gets higher as the size of the parallel window grows. This is in contrast to when solving linear problems where the number of outer iterations did not depend directly on the window size. This is the price that has to be payed for having L terms treated explicitly. In the inexact Newton's method, these terms have a smaller amplitude, resulting in less iterations.

Test 2: The idea of this test is to compare the two approaches when the Lipshitz constant of f_E is smaller, so we choose $\varepsilon = 1$ and $R = 1$. Because ε is larger, we have more freedom of chosing the time step size and the number of collocation points. Because of this, we can compare the scaling of the methods when additionally parallelizing across the collocation points. For this, we fix the number of time steps $L = 128$ and again solve to reach three different tolerances ζ, accordingly changing time-space discretizations.

[1] Inexact Newton's method does not converge for $L = 128$ time steps.

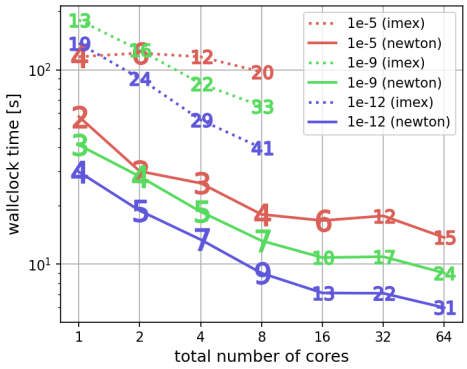
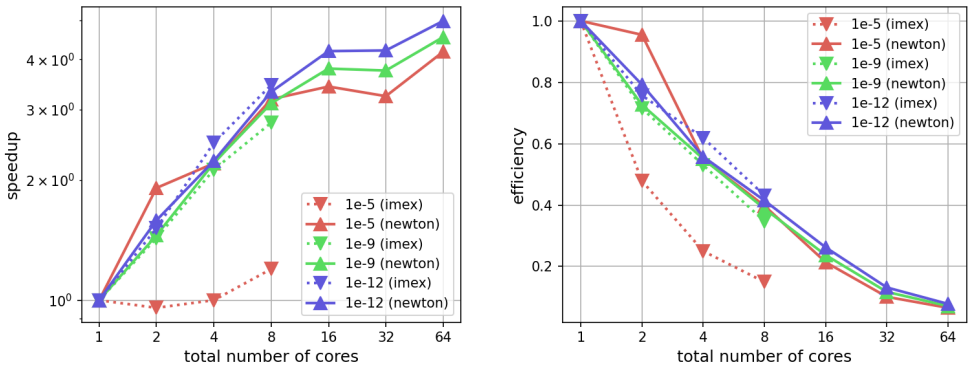


Figure 5.25.: Runtimes for $\varepsilon = 0.01$ for the Allen–Cahn equation for the inexact Newton's method and the collocation problem iterations from **Test 1**. The numbers on the graphs present the rounded average number of iterations a parallel block needs to converge.



(a) Speedup (b) Efficiency

Figure 5.26.: Strong scaling plots for the Allen–Cahn equation for **Test 1**.

Because we are adapting ΔT slightly differently, we choose the time domain as $[0, T_{128}]$ for each ζ. The difference here is that the restriction $\Delta T < \varepsilon^2$, is more easily satisfied than in the previous test. The parameter details can be found in Table 5.5.

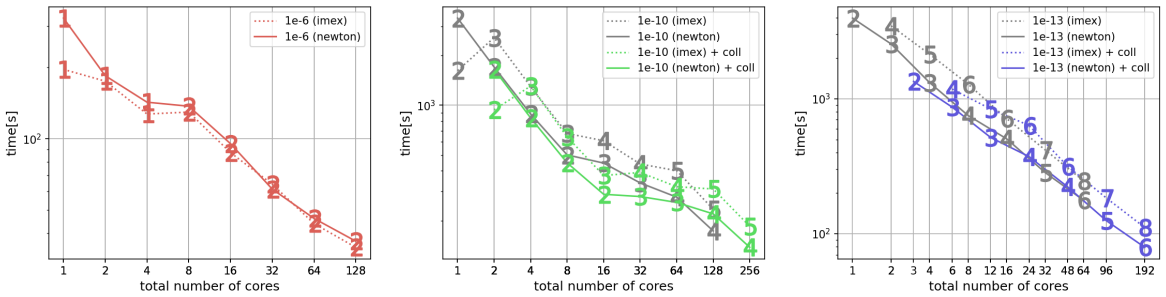
tolerance to reach ζ	10^{-6}	10^{-10}	10^{-13}
no. of spatial points N	1500	800	400
order in space κ	2	4	6
no. of collocation points M	1	2	3
time endpoint T_{128}	0.001	0.1	0.4
linear solver tolerance τ	10^{-8}	10^{-12}	10^{-15}

Table 5.5.: Parameter choice to reach an error $\|\vec{\mathbf{res}}^{(k)}\|_\infty < \zeta$ in **Test 2**.

We can see that in this case, in Figure 5.27a, the number of average iterations is closer together when comparing Newton's method and the collocation problem iterations, therefore, the scaling looks roughly the same, see Figure 5.28a. In Figures 5.27b and 5.27c[2], we can see that the collocation problem it-

[2]The fully parallel runs did not converge for $L = 128$ time steps.

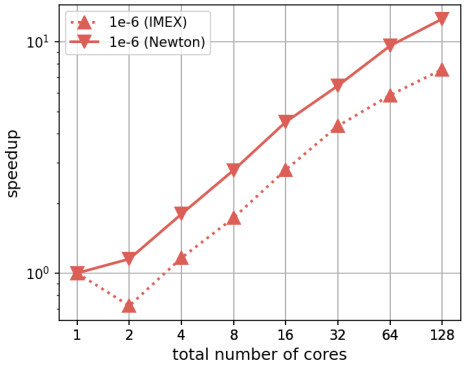
erations need more iterations compared to what the inexact Newton's method needs. This behavior is again in line with Remark 3.8, where we see that for a 'less' nonlinear f_E, both methods resemble each other more because the Lipschitz constants for these two approaches are more similar for $\varepsilon = 1$ than for $\varepsilon = 0.01$. We can see that the methods can scale further when parallelizing across all collocation nodes. The inexact Newton's method still seems to be a more favorable approach, even though in every iteration an average Jacobian is computed. It seems that when the Jacobian is a diagonal matrix (which in this case it is), the communication cost is negligible and the tradeoff between communication and the number of outer iterations is worth it.



(a) $M = 1$ (b) $M = 2$ (c) $M = 3$

Figure 5.27.: Runtimes for $\varepsilon = 1$ for the Allen–Cahn equation for the inexact Newton's method and the collocation problem iterations from **Test 2**. The numbers on the graphs present the rounded average number of iterations a parallel block needs to converge. The gray curves are runtimes with parallelism across time steps and the colored curves are parallel runs across time steps and collocation nodes.

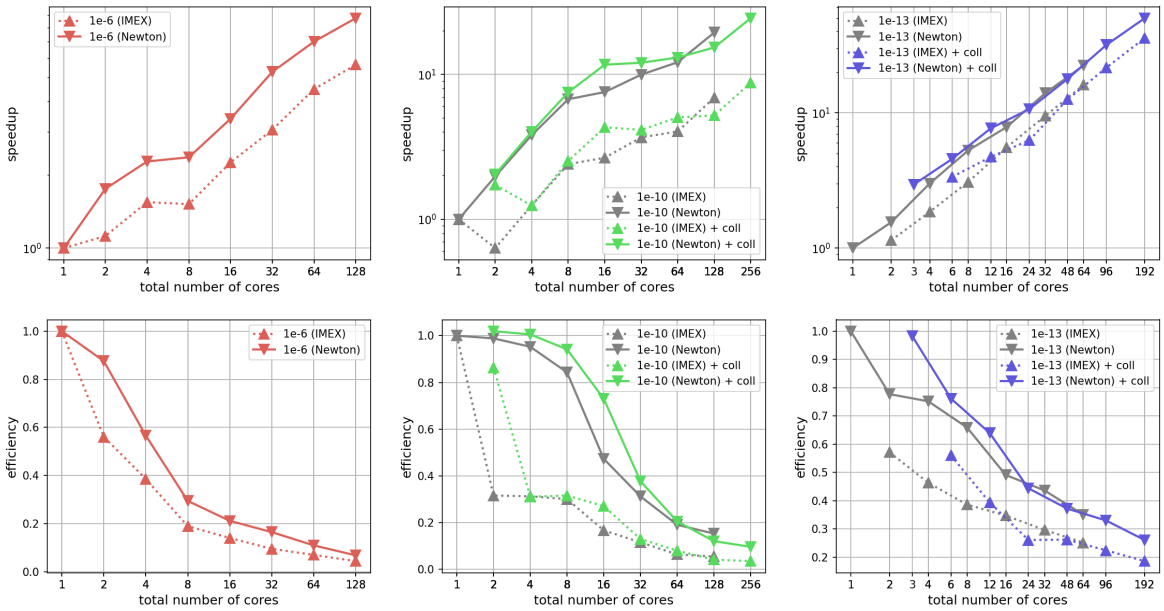
Remark 5.2



*Even with iterative refinement, ParaDiag is still susceptible to round-off errors, although they are damped with each iteration. To reduce the number of outer iterations required, solving the equation with an inner linear solver relative tolerance of $\widetilde{\tau} \ll \tau$ may be effective. Here, τ represents the tolerance necessary for the sequential approach to achieve ζ. If we select $\widetilde{\tau}$ such that $\zeta \approx \widetilde{\tau}/\alpha$, it may be possible to complete the computation in a single iteration in some cases. To demonstrate this, we can replicate the procedure of **Test 2** for $M = 1$ collocation nodes, using ParaDiag with tolerance $\widetilde{\tau} = 10^{-12}$. The plot on the right illustrates the speedup that ParaDiag achieved over the sequential run, where the inner tolerance was $\tau = 10^{-8}$. Only one iteration was required for each parallel window, resulting in a speedup of approximately 12.5. This represents an improvement over the previously reported speedup of about 8.5. This phenomenon seems to be an isolated behaviour since the speedup observed in the test with $M = 2$ collocation nodes does not improve speedup, but reduces it.*

Figure 5.28 presents the speedup and efficiency plots. We can see that ParaDiag can reach good speedups for the limited number of time-steps it can converge on, with inexact Newton's method outperforming the collocation problem iterations. In the case of implicit Euler, the reached speedup is around 9 for inexact Newton's method and around 5.5 for the collocation problem iterations.

In case when $M = 2$ collocation nodes are used, inexact Newton's method can reach a speedup of around 20 when parallel accross the time steps and 25 with additional parallelization in the colloation nodes.



(a) $M = 1$ (b) $M = 2$ (c) $M = 3$

Figure 5.28.: Strong scaling plots for the Allen–Cahn equation for **Test 2**. The gray curves represent scaling across time steps, whereas the colored curves present scaling across time steps and across collocation nodes.

The efficiencies are then around 15% and 20%[3] respectively.

In case when $M = 3$ collocation nodes are used, the maxmum speedup achieved is around 50 for the inexact Newton's method with an efficiency around 25%. When scaling across time steps, the wall clock time drops from around $4000s$ to $180s$ and is reduced even more with additional parallelism across the collocation nodes down to $80s$. These are regarded as rather good results for parallel-in-time methods. This particular run is composed of 2 parallel windows covering $L = 64$ time steps since a full parallel run covering 128 time steps did not converge. Scaling for the collocation problem iterations was compared to a sequential inexact Newton's run because the underlying iteations simply exceeded every reasonable computational time limit. This kind of comparison is still in the spirit of representing speedup and efficiency because, if possible, the fastest known sequential method should be used for comparison.

5.4.2. Boltzmann equation

The Boltzmann equation is a well-known model from the area of rarefied gas dynamics. It has a form

$$f_t + v f_x = \frac{1}{\varepsilon} Q(f), \quad (t, x, v) \in \mathbb{R}_0^+ \times \mathbb{R}^3 \times \mathbb{R}^3, \tag{5.3}$$

[3]The efficiency for the inexact Newton's method is around 101%. In theory, this should not be achievable, however, it seems that the configuration of this specific run had more favorable memory layout. Another reason could be different caching effects that take place in the two runs. A measurement error of ±3% can be expected for the runs that are not repeated multiple times.

and describes the evolution of a many-particle system with the particle distribution function $f = f(t, x, v)$, which varies with time t, position x, and velocity v. The parameter ε is a relaxation parameter proportional to gas dilution and represents the nondimensional Knudsen number. The right-hand side of the Boltzmann equation,

$$Q(f) = \int_{\mathbb{R}^3} \int_{\mathbb{S}^2} \left[f\left(v'\right) f\left(v'_*\right) - f(v) f\left(v_*\right) \right] \mathcal{B}(\cos\theta, |v - v_*|) d\Omega dv_*, \tag{5.4}$$

models two-body collisions, where $\{v, v_*\}$ denote the pre-collision velocities of two colliding particles, and $\{v', v'_*\}$ are the corresponding post-collision velocities. The nonnegative collision kernel $\mathcal{B}(\cos\theta, g)$ measures the probability of collisions, where θ is the deflection angle and $|v - v_*|$ is the magnitude of the relative pre-collision velocity. The solid angle Ω is the unit vector along the relative post-collision velocity $v' - v'_*$.

Furthermore, as $\varepsilon \to 0$, the equation becomes stiff, introducing additional numerical stability difficulties for explicit time stepping. Note that equation (5.3) describes a relaxation system, where the advection operator in the left-hand side drives the particle system towards non-equilibrium, whereas the collision operator in the right-hand side brings the solution towards the equilibrium, which follows a normal distribution in v. For an equilibrium solution, the conservation of mass and momentum leads to the relation $f(v')f(v'_*) = f(v)f(v_*)$ and the collision term vanishes. Thus, equation (5.4) is bounded as $\varepsilon \to 0$.

It is challenging to solve the Boltzmann equation due to the high dimensionality and nonlinearity. Evaluating the collision operator Q is the most costly part of the computation, and using the inexact Newton's method is not feasible. The scaling results with ParaDiag coupled with the collocation problem show excellent results for the linear advection equation, a difficult problem for time-parallel integration. Therefore, this example highlights the proposed IMEX approach's capability of treating the collision operator explicitly to solve the nonlinear Boltzmann equation.

We consider the equation of form

$$f_t + v_1 f_x = \frac{1}{\varepsilon} Q(f), \quad (t, x, v) \in [0, 1] \times [0, 1] \times [-8, 8]^3, \tag{5.5}$$

with periodic boundary conditions $f(t, 1, v) = f(t, 0, v)$ and an initial condition

$$f(0, x, v) = \rho(x) \left(\frac{1}{\pi T} \right)^{3/2} \exp\left(-\frac{(v - V)^2}{T(x)} \right),$$

where

$$\rho(x) = 1 + 0.1 \sin(2\pi x),\ V = (1, 0, 0)^T,\ T(x) = \frac{1}{\rho(x)}.$$

In this case, we adopt the hard-sphere molecule model [80, Chapter 2, p. 39] where the collision kernel takes the form

$$\mathcal{B}(|v - v_*|) = C|v - v_*|^{1/2}.$$

In the initial condition, V is a vector representing the macroscopic velocity, while ρ and T are two scalars that represent density and temperature, respectively. Equation (5.5) describes a traveling wave solution in the gas. Even though the advection operator contains only one velocity component, the example is enough to demonstrate how well time-parallel integration copes with similar problems.

To obtain the solution, we employ the upwind scheme to approximate spatial derivatives. A fast spectral method to is used to solve the quadratic integral operator Q, which converts it into a summation

of convolutions and then solves it using the discrete Fourier transform [81]. This part tends to be the most costly one when solving the Boltzmann equation, and it was treated with the help of a `Julia` library `KitBase.jl`[4]. We refer to the literature [82, 3] for details and implementation of this approach.

In the simulation, we use 384 uniform points to discretize the physical domain x and $72 \times 36 \times 36$ velocity points. The spatial resolution that we fix here is able to capture the wave structure in the initial condition of the equation (5.5). For the time discretization, we choose the implicit Euler method with $\Delta T = 10^{-3}$ and solve for $L = 32$ time steps[5], where for the spatial discretization we use a 1st order upwind scheme. For example, the time discretization is not in the stability domain of the explicit Euler method since ΔT is too large, resulting in solutions that blow up. The stopping criterion is $\|\vec{\mathbf{res}}^{(k)}\|_\infty < \zeta = 10^{-4}$, with a linear solver tolerance of $\tau = 10^{-6}$. These are reasonable tolerances since we expect the accuracy of the solution to be $\Delta T \approx \Delta x \approx \mathcal{O}(10^{-3})$. Figure 5.29 illustates the density of the solution in two different time points, assuring that the solution evolves over time. The density is defined as

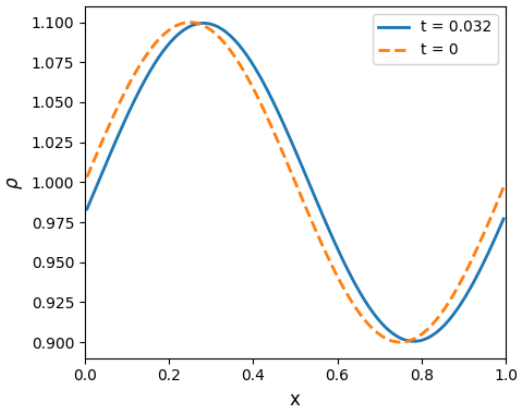


Figure 5.29.: Density in time points $t = 0, 0.032$ for a value $\varepsilon = 10^{-2}$.

$$\rho := \int_{\mathbb{R}^3} f v^\top dv.$$

Test 1: The relaxation parameter is set to $\varepsilon = 10^{-2}$. First, we scale the problem by solving it parallel-in-space, but sequentially in time. The collision term is appropriately computed embarrassingly parallel: looping over each point x_i available in the local memory of a corresponding core. Then, an additional level of parallelism is added across time steps: we solve parallel in time and space for a block of $n_{\text{step}} = 1, 2, \ldots, 32$ coupled time steps and propagate the solution in these blocks until reaching the total of 32 time steps. Figure 5.31 depicts the total runtimes when fixing the number of spatial cores to $n_{\text{space}} = 16, 32, 64$ and adding time parallelism. The number of cores n_{space} is chosen as the last number of cores where `petsc4py` scales reasonably well. We can see that the wall clock time is not reduced between 32 and 128 cores.

Even though spatial parallelism begins to be less efficient, we can continue scaling further with time parallelism. The speedup curves in Figure 5.31a differ more and more from the ideal scaling because our time-parallel integrator is an iterative method. Having a larger parallel block translates to more iterations for that block to converge. This is in line with the convergence Theorem 3.6, which states that the bound for the contraction factor of the method scales linearly with the number of parallel time steps. Table 5.6 summarizes the number of iterations per block. Following the structure of the colored curves, one can see that having a block of 4 and 8 parallel steps does not provide speedup due to the rising number of iterations. The time-parallel integration continues scaling for 16 and 32 coupled time steps.

[4]The essential parts of the algorithm, namely the discrete Fourier transforms, were recoded into `pyParaDaig` directly. The rest of the setup variables were copied from a file that was produced as a setup output from a `Julia` code. With this, one does not have to use a code wrapper combining the two languages, which proved to lead to significant overheads and long

n_{step}	2	4	8	16	32
no. of block propagations	16	8	4	2	1
total no. of iterations	16	13	12	6	8
no. of iterations per time step	1	1.625	3	3	3

Table 5.6.: The number of iterations the method needs when handling L time-steps in parallel and propagating them until covering 32 time-steps in total. We can observe that the number of iterations per time step grows proportionally to the number of parallel steps.

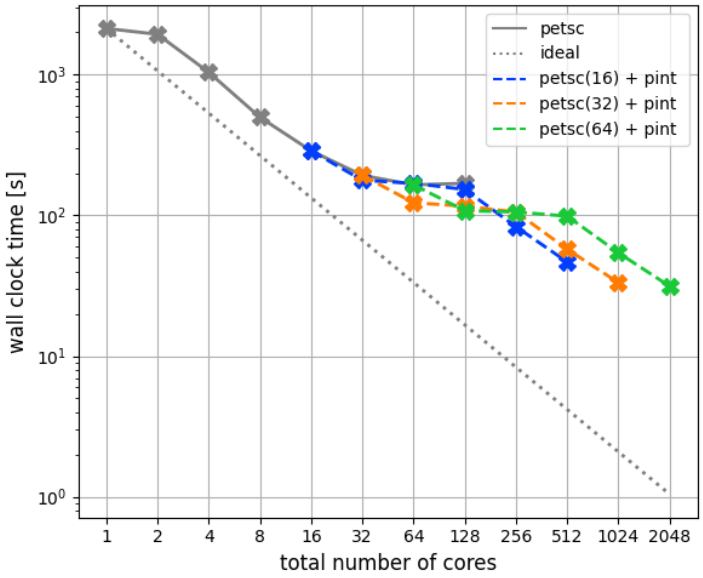


Figure 5.30.: Wall clock times for the Boltzmann equation from **Test 1**. The `petsc` line stands for the runs that solve the Boltzmann equation in space-parallel fashion, covering $L = 32$ time steps sequentially in time. `petsc(n)` means that `n` cores are used for the parallelization of the spatial problem, i.e., $n_{\text{space}} =$ `n`. On top of spatial parallelism with `n` fixed cores, parallelism across the time domain is added, covering $L = 32$ time steps.

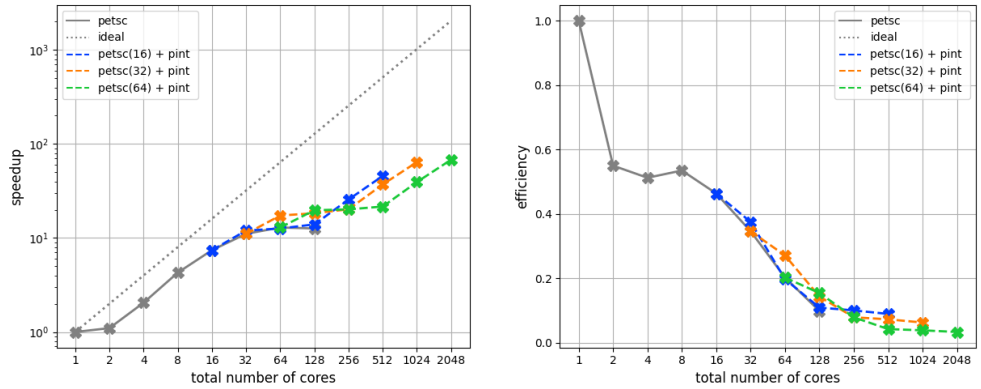
In conclusion, the simulation's runtime was significantly reduced by spatial scaling, from 2134 seconds to approximately 169 seconds, resulting in a speedup of $s_{\text{space}} = 12.6$. The addition of time parallelization further reduced the total runtime to approximately 31 seconds, utilizing 64 spatial cores. The overall speedup achieved using 2048 cores was $s_{\text{space+time}} \approx 68$. The efficiency achieved with 16 cores in space was around 10%. For time-parallel integration methods, this result is important since speedup is obtained also for a hyperbolic nonlinear problem.

Remark 5.3

The efficiency of spatial scaling decreases significantly for 2 *and* 4 *cores. See Figure 5.31b. This is due to the memory bandwidth limitations: the memory passed between cores is larger when using fewer cores since the locally stored vectors are stored in larger chunks. Because of this, the efficiency is later recovered for* 8 *or more cores.*

compilation times. The overheads were in favor of time-parallel integration.

[5]The algorithm does not converge for $L = 64$ parallel time-steps.



(a) Speedup

(b) Efficiency

Figure 5.31.: Strong scaling plots for the Boltzmann equation for **Test 1**.

Test 2: The second test examines convergence for the same setup but for varying ε values. We pick three different relaxation paramters $\varepsilon = 8 \times 10^{-3}, 10^{-2}, 5 \times 10^{-2}, 10^{-16}$. A reasonable number of cores for the spatial parallelization, concluding from **Test 1** is $n_{\text{space}} = 32$. We conduct the runs in parallel across time steps in a windowed manner, fixing the number of cores for spatial discretization and covering a total of $L = 32$ time steps with $n_{\text{step}} = 1, 2, 4, \ldots, 32$.

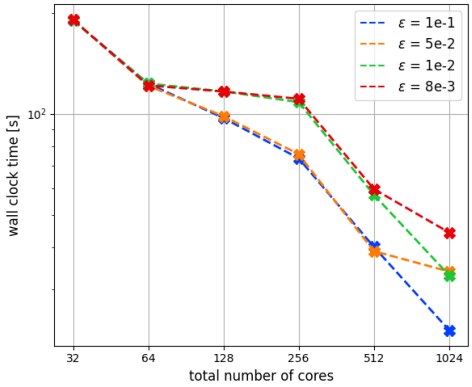


Figure 5.32.: Wall clock times for the Boltzmann equation from **Test 2** with varying ε.

Figure 5.32 compares the wall clock times of the four runs. We can see how ParaDiag needs more iterations as the relaxation parameter ε decreases. This is in line with Theorem 3.6, which states that the bound for the contraction factor increases proportionally to η, the contraction factor of the underlying single time step collocation problem iterations. On the other hand, the parameter η increases as ε decreases, increasing the total number of ParaDiag iterations. The average number of iterations per run is given in Figure 5.33. As we have already seen for the Allen–Cahn equation in **Test 1** & **2** (Figures 5.25 & 5.27), we can see how the average number of iterations per block grows proportionally with the size of a parallel block n_{step} and inversely proportionally to ε.

Overall, the results of **Test 2** align with our theoretical understanding of the method. The larger the amplitude of the explicit term in the iterations, the more difficult it becomes for the implicit left side to dampen them. It is also clear that the scaling improves after a certain amount of parallel time steps. In other words, there has to be a certain investment of cores handling the parallel block to make the scaling worth it. The reason is that all the communication times of all parallel blocks and all iterations are accumulated, degrading speedup. Still, given enough cores for parallelization across time steps, we can achieve excellent scaling for the Boltzmann equation as a time-parallel method.

[6] $\varepsilon = 5 \times 10^{-3}$ does not converge.

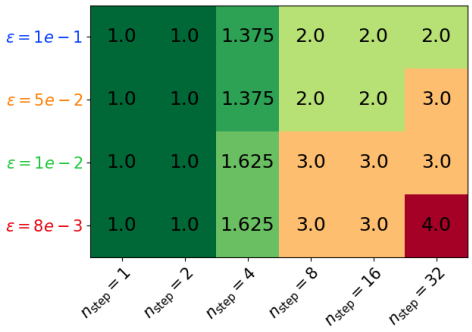


Figure 5.33.: The average number of iterations for a ParaDiag block and wallclock times depending on the relaxation parameter ε and the size of a parallel block n_{step}, covering $L = 32$ time steps. The runs correspond to **Test 2**, and the colors of ε-labels correspond to the curves in Fiure 5.32.

CHAPTER 6

Conclusion and outlook

6.1. Conclusion

In this thesis, we introduce ParDiag, a parallel-in-time integrator for linear problems that is based on diagonalization. The method relies on an α-circulant preconditioner, which is applied in an "all-at-once" fashion using a straightforward Richardson iteration. The diagonalization of this preconditioner leads to an appealing time-parallel method without the need to find a suitable coarsening strategy and a promising approach to treat hyperbolic-type equations.

We extend this idea to high-order collocation problems and analyze how to efficiently solve the local problems for each time step in parallel: across the collocation nodes of the perturbed collocation matrix, making this method doubly parallel-in-time. We propose a practical and applicable strategy to adaptively select the crucial α-parameter for each iteration based on the convergence theory and error bounds. However, depending on the size of the collocation problem, we show that some of these values α lead to non-diagonalizable inner systems, which need to be avoided. We support our claims with a benchmark and conclude that the strategy reduces the number of iterations and time-to-solution.

By design, the α-circulant integrator works only for linear problems with constant coefficients. In order to solve more complex, more realistic problems, they must be coupled to a nonlinear solver. A unifying analysis of two time-parallel integrators for nonlinear problems is presented. One originates from the inexact Newton's iterative method, while the motivation for the other lies in the implicit-explicit iteration, where we treat the nonlinear terms explicitly. In comparison, when convergent, inexact Newton's method needs fewer iterations than when the nonlinearities are treated explicitly, leading to more speedup in the first case. The fixed point iterations are carried out via iterative refinement since the α-adaptive strategy cannot be trivially extended. The case with variable space-dependent coefficients can be extended through the implicit-explicit iteration.

An open-source parallel implementation is shown, which is utilized to present actual parallel runs on a high-performance computing system to support our theory. We estimate the expected speedup and

show actual parallel runs on a high-performance computing system to support our claims. We stress the importance of choosing suitable test cases, after which we benchmark the proposed algorithm on four test cases: the linear heat and advection equation and the nonlinear Allen-Cahn and Boltzmann equation. For nonlinear equations, both preconditioners show promising scaling when convergent. The parallel benchmarks demonstrate that our proposed approaches yield a significant decrease in time-to-solution, even far beyond the point of saturation of spatial parallelization. As mentioned in the introductory Section 1.2.3, we achieve significant speedups for hyperbolic-type equations, and our results are comparable to the state-of-the-art parallel-in-time integrators such as MGRIT and PFASST, even outperforming them in some cases.

6.2. Outlook

The diagonalization process in ParaDiag may produce ill-conditioned shifted systems. It is difficult to derive a general theory investigating this issue since this occurrence is strongly problem-related. However, the analysis could be provided for the discrete Laplacian, like in the case of the heat equation where this matter arises. Even though the `petsc4py`-coupled implementation allows much flexibility with system solvers and preconditioners, it was out of the scope of this work to try and test various combinations. It is an interesting topic left for future work.

The theoretical framework for analyzing the IMEX iterations allows us to define the implicit-explicit splitting in any way. This depends on the equation being solved, however, different definitions of the explicit and implicit part may improve convergence. ParaDiag can also be used as an extension for computing iterations where the explicit part is computed in time $T_{\ell-1}$ whereas the implicit part is constructed in time T_ℓ. The implementation of this time-stepping method is currently under construction.

Another exciting line of research is to apply the method for optimal control problems. PFASST was tried out for this idea in [83], however, just for parabolic equations. ParaDiag is proposed as an approach in [84] for wave equations, preconditioning the state and the adjoint equations in a coupled way of two blocks for $\alpha = 1$. The approach seems promising and has much potential since optimal control may profit from a specific type of error control in each iteration.

Lastly, an idea of preconditioning the collocation problems as done in SDC may be performed inside ParaDiag. One approach is to solve the shifted inner systems in an iterative SDC-like way. Another approach is to directly incorporate the preconditioner for the collocation problem into the α-circulant matrix, which would then bear the preconditioning matrices for the collocation problem instead of the matrix $\mathbf{Q}$. Because of this, the inner systems do not need to be solved via diagonalization of $\mathbf{Q}\mathbf{G}_\ell^{-1}$, but instead using triangular matrices that can be solved via forward/backward substitutions. Other candidates are matrices that can easily be reduced to diagonal forms after the formation of the inner systems. So far, no tests have been conducted, and whether the tradeoff of making each iteration computationaly even cheaper provides more speedup is still being determined.

APPENDIX A

Spectral radius and the infinity norm of the iteration matrix

This segment presents findings regarding the iteration matrix's spectral radius and infinity norm. Notwithstanding being unreported, the outcomes are incorporated in the manuscript as they were obtained during the research period. In the meantime, similar results have been published: bounds on the iteration matrix norm from 2021 can be located in [57, Theorem 2.1], while the spectral radius examination from 2022 is available in [52, Theorem 2.1].

First, we need one important result.

Corollary A.1
If matrices $\mathbf{B}_1, \ldots, \mathbf{B}_n$ *all commute, then the spectral radius of the product can be bound as*

$$\rho(\mathbf{B}_1 \ldots \mathbf{B}_n) \leq \rho(\mathbf{B}_1) \ldots \rho(\mathbf{B}_n).$$

Proof: Gelfand's formula for the spectral radius is $\rho(\mathbf{B}) = \lim_{k\to\infty} \left\|\mathbf{B}^k\right\|^{\frac{1}{k}}$. Using the commuting property of the matrices and submultiplicivity of matrix norms, the result is evident. □

Theorem A.2
Let $\mathbf{R} := \mathbf{C}_\alpha^{-1}(\mathbf{C}_\alpha - \mathbf{C})$ *define the iteration matrix of preconditioned Richardson iterations* (2.18) *corresponding to the initial value problem* (2.9)

$$u_t = \mathbf{A}u + b, \quad u(T_0) = u_0 \in \mathbb{C}^N.$$

Let $\Phi_1, \ldots, \Phi_M$ *denote the* M *implicit stages obtained by solving the collocation problem of order* M *with a step size* ΔT *defined in* (2.8) *for the equation*

$$U_t = \mathbf{A}U, \quad U(T_0) = \mathbf{I}_N. \tag{A.1}$$

If $\alpha \left\|\Phi_M^L\right\| < 1$*, where* L *is the number of time steps, then*

$$\rho(\mathbf{R}) \leq \frac{\alpha \left\|\Phi_M^L\right\|}{1 - \alpha \left\|\Phi_M^L\right\|} \tag{A.2}$$

holds for any norm $\|\cdot\|$. *Furthermore, if* $\alpha \left\|\Phi_m \Phi_M^{L-1}\right\|_\infty < 1$, *for* $1 \leq m \leq M$, *then*

$$\|\mathbf{R}\|_\infty \leq \frac{\alpha \max_m \left\{\|\Phi_m\|_\infty, \|\Phi_m\|_\infty \|\Phi_M\|_\infty^{L-1}\right\}}{1 - \alpha \max_m \left\|\Phi_m \Phi_M^{L-1}\right\|_\infty}.$$

Proof: For simplicity let us define $\mathbf{T} := \mathbf{C}_{\text{coll}}^{-1}\mathbf{H}$. Then, the preconditioner can be rewritten as

$$\mathbf{C}_\alpha = (\mathbf{I}_L \otimes \mathbf{C}_{\text{coll}}) \underbrace{\begin{bmatrix} \mathbf{I}_{MN} & & & -\alpha\mathbf{T} \\ -\mathbf{T} & \mathbf{I}_{MN} & & \\ & \ddots & \ddots & \\ & & -\mathbf{T} & \mathbf{I}_{MN} \end{bmatrix}}_{=:\mathbf{T}_\alpha}.$$

The inverse of $\mathbf{T}_\alpha$ is

$$\mathbf{T}_\alpha^{-1} = \left(\mathbf{I}_L \otimes \left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1}\right) \begin{bmatrix} \mathbf{I}_{MN} & \alpha\mathbf{T}^{L-1} & \alpha\mathbf{T}^{L-2} & \dots & \alpha\mathbf{T} \\ \mathbf{T} & \mathbf{I}_{MN} & \alpha\mathbf{T}^{L-1} & \dots & \alpha\mathbf{T}^2 \\ \mathbf{T}^2 & \mathbf{T} & \mathbf{I}_{MN} & \dots & \alpha\mathbf{T}^3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{T}^{L-1} & \mathbf{T}^{L-2} & \mathbf{T}^{L-3} & \dots & \mathbf{I}_{MN} \end{bmatrix}, \tag{A.3}$$

which can be checked directly. The easiest way to derive the inverse is using the inverse of the Toeplitz matrices which has a closed form and generalizing it to block matrices. The condition $\alpha \left\|\Phi_M^L\right\| < 1$ verifies the existence of inverse of the matrix $\mathbf{I}_{MN} - \alpha\mathbf{T}^L$. To clarify this, $\mathbf{T}$ is a solution of $\mathbf{C}_{\text{coll}}\mathbf{T} = \mathbf{H}$. From here we can conclude that it has a form

$$\mathbf{T} = \begin{bmatrix} 0 & \dots & 0 & \mathbf{T}_{1M} \\ 0 & \dots & 0 & \mathbf{T}_{2M} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & \mathbf{T}_{MM} \end{bmatrix},$$

otherwise the collocation problem is not well defined. Entries $\mathbf{T}_{mM}$ are exactly the implicit stages for the equation (A.1) corresponding to one propagation of the method. Consequently, we have $\mathbf{T}_{mM} = \Phi_m$. Calculating the powers of $\mathbf{T}$ gives

$$\mathbf{T}^k = \begin{bmatrix} 0 & \dots & 0 & \Phi_1\Phi_M^{k-1} \\ 0 & \dots & 0 & \Phi_2\Phi_M^{k-1} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & \Phi_M^k \end{bmatrix},$$

from where we see that the matrix $\mathbf{I}_{MN} - \alpha\mathbf{T}^L$ is nonsingular if $\mathbf{I}_N - \alpha\Phi_M^L$ is nonsingular. This is a direct consequence of the Binet-Cauchy theorem and the condition $\alpha \left\|\Phi_M^L\right\| < 1$ is sufficient. Using the inverse $\mathbf{T}_\alpha^{-1}$ (A.3) yields

$$\mathbf{R} = \mathbf{T}_\alpha^{-1} \left(\mathbf{I}_L \otimes \mathbf{C}_{\text{coll}}^{-1}\right)(\mathbf{C}_\alpha - \mathbf{C}) = \mathbf{T}_\alpha^{-1} \begin{bmatrix} 0 & \dots & -\alpha\mathbf{T} \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix},$$

and multiplying these block matrices gives

$$\mathbf{R} = -\alpha\left(\mathbf{I}_L \otimes \left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1}\right)\begin{bmatrix} 0 & \dots & 0 & \mathbf{T} \\ 0 & \dots & 0 & \mathbf{T}^2 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & \mathbf{T}^L \end{bmatrix}. \tag{A.4}$$

Thus the nonzero eigenvalues of $\mathbf{R}$ are the same as for the matrix $-\alpha\left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1}\mathbf{T}^L$, therefore the spectral radius of the iteration matrix is

$$\rho(\mathbf{R}) = \alpha\rho\left(\left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1}\mathbf{T}^L\right).$$

Furthermore, matrices $\left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1}$ and $\mathbf{T}^L$ commute since

$$\mathbf{T}^L = \mathbf{T}^L\left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)\left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1} = \left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)\mathbf{T}^L\left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1}$$

and thus

$$\left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1}\mathbf{T}^L = \mathbf{T}^L\left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1}.$$

Note that $\mathbf{T}$ is singular and the commuting property is not obvious. The spectral radius can now be bounded as

$$\rho(\mathbf{R}) \leq \alpha\rho\left(\left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1}\right)\rho\left(\mathbf{T}^L\right)$$

due to Gelfand's corollary A.1. Here, $\rho\left(\mathbf{T}^L\right) \leq \left\|\Phi_M^L\right\|$ and since the inverse of a block triangular matrix is known, we have

$$\begin{aligned}\rho\left(\left(\mathbf{I}_{MN} - \alpha\mathbf{T}^L\right)^{-1}\right) &= \max\left\{1, \rho\left(\left(\mathbf{I}_N - \alpha\Phi_M^L\right)^{-1}\right)\right\} \\ &\leq \max\left\{1, \left\|\left(\mathbf{I}_N - \alpha\Phi_M^L\right)^{-1}\right\|\right\} \\ &\leq \frac{1}{1 - \alpha\left\|\Phi_M^L\right\|},\end{aligned}$$

where the last inequality holds beacuse $\alpha\left\|\Phi_M^L\right\| < 1$. This completes the proof regarding the bound for the spectral radius. It remains to show the norm bound. Revisiting equation (A.4) gives

$$\|\mathbf{R}\|_\infty \leq \alpha\left\|\left(\mathbf{I}_{NM} - \alpha\mathbf{T}^L\right)^{-1}\right\|_\infty \max\left\{\|\mathbf{T}\|_\infty, \dots, \left\|\mathbf{T}^L\right\|_\infty\right\}.$$

The condition $\alpha\left\|\Phi_m\Phi_M^{L-1}\right\|_\infty < 1$ is equivalent to $\alpha\left\|\mathbf{T}^L\right\|_\infty < 1$ because $\left\|\mathbf{T}^L\right\|_\infty = \max_m\left\|\Phi_m\Phi_M^{L-1}\right\|_\infty$, hence

$$\left\|(\mathbf{I}_{NM} - \alpha\mathbf{T}^L)^{-1}\right\|_\infty \leq \frac{1}{1 - \alpha\max_m\left\|\Phi_m\Phi_M^{L-1}\right\|}.$$

The second factor can be bound as

$$\begin{aligned}\max\left\{\|\mathbf{T}\|_\infty, \dots, \left\|\mathbf{T}^L\right\|_\infty\right\} &\leq \max_m\left\{\|\Phi_m\|_\infty, \|\Phi_m\|_\infty\|\Phi_M\|_\infty, \dots, \|\Phi_m\|_\infty\|\Phi_M\|_\infty^{L-1}\right\} \\ &\leq \max_m\left\{\|\Phi_m\|_\infty, \|\Phi_m\|_\infty\|\Phi_M\|_\infty^{L-1}\right\}.\end{aligned}$$

The last inequality holds because it summarizes the cases when $\|\Phi_M\|_\infty \leq 1$ or $\|\Phi_M\|_\infty > 1$. This completes the second part of the proof. □

Remark A.3
For stiff problems, if the underlying collocation method from Theorem A.2 is stable, meaning $\|\Phi_M\| < 1$, *then the condition* $\alpha \left\|\Phi_M^L\right\| < 1$ *is automatically satisfied for any* $\alpha < 1$. *Consequently,* $\rho(\mathbf{R}) \leq \frac{\alpha}{1-\alpha}$.

The proof of Theorem A.2 provides an intuition of the convergence behavior. The approximation Φ_M is essentially an approximation of $e^{\Delta T \mathbf{A}}$ of some order depending on the quadrature. The bound can be interpreted as

$$\rho(\mathbf{R}) \lesssim \frac{\alpha \left\|e^{L\Delta T \mathbf{A}}\right\|}{1 - \alpha \left\|e^{L\Delta T \mathbf{A}}\right\|}. \tag{A.5}$$

The same can be concluded for the bound $\|\mathbf{R}\|_\infty$ since for a very small step size, we get $\Phi_m \Phi_M^{L-1} \approx \Phi_M^L$. For a sufficiently small parameter α, we may say that the contraction factor of preconditioned Richardson iterations (2.18) is $o(\alpha)$.

APPENDIX B

Bounds for the norm of the collocation matrix

On function spaces, there is a natural scalar product. For integrable functions $f, g : \mathbb{R} \to \mathbb{R}$ a scalar product can be defined as

$$\langle f, g \rangle = \int_0^1 f\, g. \tag{B.1}$$

This allows us to define orthogonality on function spaces. The family of orthogonal polynomials can be used to construct high-order Gaussian-quadrature rules. Let

$$\int_0^1 p(s)ds = \sum_{i=1}^{M} w_i p(t_i) \tag{B.2}$$

define a quadrature rule, where some points t_i may be fixed while other are roots of an orthogonal polynomial. Then, the right sum computes the integral correctly for polynomials of degree $2M-2$ or less. Depending on the choice of these points, we get different quadrature rules such as Legendre, Radau, Lobatto, etc. The integration weights w_i are positive, and the integration points t_i are inside the integration interval [85]. In our case, without loss of generality, we will assume that $t_i \in [0, 1]$.

Let the collocation matrix be defined as in (2.7). In order to compute lower and upper bounds for $\|\mathbf{Q}\|_\infty$, we first have to understand what a matrix-vector product $\mathbf{Qx}$ means. The product behaves as

$$\mathbf{Qx} = \begin{bmatrix} \sum_{j=1}^{M} \mathbf{x}_j \int_0^{t_1} c_j(s)ds \\ \sum_{j=1}^{M} \mathbf{x}_j \int_0^{t_2} c_j(s)ds \\ \vdots \\ \sum_{j=1}^{M} \mathbf{x}_j \int_0^{t_M} c_j(s)ds \end{bmatrix} = \begin{bmatrix} \int_0^{t_1} \sum_{j=1}^{M} \mathbf{x}_j c_j(s)ds \\ \int_0^{t_2} \sum_{j=1}^{M} \mathbf{x}_j c_j(s)ds \\ \vdots \\ \int_0^{t_M} \sum_{j=1}^{M} \mathbf{x}_j c_j(s)ds \end{bmatrix},$$

where c_i are the Lagrange polynomils defined in (2.3). This motivates us to construct a one-to-one map from a vector $\mathbf{x} \in \mathbb{R}^M$ to a polynomial $p \in P_{M-1}$, where $p := \sum_{j=1}^{M} \mathbf{x}_j c_j$. This is uniquely defined because the polynomial p passes through M points $(t_i, \mathbf{x}_i)$ and is of degree $M-1$. Then, the sum $\int_0^{t_i} \sum_{j=1}^{M} \mathbf{x}_j c_j(s)ds$ is nothing more than $\int_0^{t_i} p(s)ds$, and we can conclude that multiplying a vector with a matrix $\mathbf{Q}$ is as computing integrals of a corresponding polynomial.

Now, we are interested in computing the norm $\|\mathbf{Q}\|_\infty$. By definition, the norm is

$$\|\mathbf{Q}\|_\infty = \max_{\|\mathbf{x}\|_\infty = 1} \|\mathbf{Q}\mathbf{x}\|_\infty, \tag{B.3}$$

but the infinity norm of a matrix can also be computed as

$$\|\mathbf{Q}\|_\infty = \max_{1 \le i \le M} \sum_{j=1}^{M} |\mathbf{Q}_{ij}| = \max_{1 \le i \le M} \sum_{j=1}^{M} \left| \int_0^{t_i} c_j(s) ds \right|,$$

which is the maximum row-sum of the absolute values of its entries. From this, we can see that

$$\begin{aligned} t_i = \left| \int_0^{t_i} 1 ds \right| &= \left| \int_0^{t_i} \sum_{j=1}^{M} c_j(s) ds \right| \\ &= \left| \sum_{j=1}^{M} \int_0^{t_i} c_j(s) ds \right| \\ &\le \sum_{j=1}^{M} \left| \int_0^{t_i} c_j(s) ds \right| \\ &\le \sum_{j=1}^{M} |\mathbf{Q}_{ij}| \le \|\mathbf{Q}\|_\infty. \end{aligned}$$

This gives a lower bound: $t_M \le \|\mathbf{Q}\|_\infty$.

Furthermore, we will redefine the definition of the matrix norm (B.3) in a way that we do not compute the maximum over all vectors $\mathbf{x}$ for which $\|x\|_\infty = 1$, but over uniquely defined polynomials p since $\mathbb{R}^M \ni \mathbf{x} \mapsto p \in P_{M-1}$ is a one-to-one map, as discussed above. Proving $\|\mathbf{Q}\|_\infty \le 1$ is then equivalent to proving that

$$\max_{1 \le i \le M} \left| \int_0^{t_i} p(s) ds \right| \le 1 \tag{B.4}$$

holds, for $p \in P_{M-1}$ and $\max_{1 \le i \le M} |p(t_i)| = 1$.

B.1. Exact quadrature for degree $2M - 2$ or higher

First, using the integral Cauchy–Bunyakovsky–Schwarz inequality on the scalar product defined as in (B.1), we can get an upper bound on $\left| \int_0^{t_i} p(s) ds \right|$ as

$$\left| \int_0^{t_i} p(s) ds \right|^2 = \left| \int_0^{t_i} p(s) \cdot 1 ds \right|^2 \le \left| \int_0^{t_i} p^2(s) ds \right| \left| \int_0^{t_i} 1^2 ds \right| \tag{B.5}$$

$$\le t_i \int_0^{t_i} p^2(s) ds \tag{B.6}$$

$$\le t_i \int_0^{1} p^2(s) ds. \tag{B.7}$$

Since $p^2 \in P_{2M-2}$, using the quadrature rule (B.2) which correctly computes integrals of polynomials of degree $2M-2$, we get

$$\int_0^1 p^2(s)ds = \sum_{i=1}^{M} w_i p^2(t_i) \leq \underbrace{\Big(\sum_{i=1}^{M} w_i\Big)}_{=1} \max_{1\leq i\leq M} p^2(t_i) = \max_{1\leq i\leq M} p^2(t_i).$$

Because we are computing (B.4) over polynomials which satisfy $\max_{1\leq i\leq M} |p(t_i)| = 1$, we know that $\max_{1\leq i\leq M} p^2(t_i) = 1$, thus

$$\int_0^1 p^2(s)ds \leq \max_{1\leq i\leq M} p^2(t_i) = 1. \tag{B.8}$$

Now (B.7) + (B.8) yield

$$\Big|\int_0^{t_i} p(s)ds\Big|^2 \leq t_i.$$

In conclusion,

$$t_i \leq \Big|\int_0^{t_i} p(s)ds\Big| \leq \sqrt{t_i}.$$

This proves that $t_M \leq \|\mathbf{Q}\|_\infty \leq \sqrt{t_M} \leq 1$ holds for nodes originating from the Gaussian quadrature that integrate polynomials of degree $2M-2$ exactly.

B.2. Gauss–Lobatto nodes

The Gauss–Lobatto quadrature integrates polynomials of degree $2M-3$ exactly, therefore, the same argument as for $\|\mathbf{Q}\|_\infty \leq 1$ does not hold. However, we will show this is still true for $M \geq 2$. The Gauss-Lobatto quadrature on $[0,1]$ is defined as

$$\int_0^1 p(s)ds = \sum_{i=1}^{M} w_i p(t_i) + R_M, \tag{B.9}$$

where $R_M = -cp^{(2M-2)}(\xi)$, $c \geq 0$, $\xi \in [0,1]$, see [85] for details. If we manage to bound $\int_0^1 p^2(s)ds \leq 1$, then using the same train of thought and arguments as in (B.7), we are done.

Let $p^2(s) = as^{2M-2} + \ldots$, where $a > 0$. We can write

$$\int_0^1 p^2(s)ds = \int_0^1 (p^2(s) - as^{2M-2})ds + \int_0^1 as^{2M-2}ds.$$

The polynomial $p^2(s) - as^{2M-2}$ is of degree $2M-3$ and can be computed with quadrature (B.9) as

$$\int_0^1 (p^2(s) - as^{2M-2})ds = \sum_{i=1}^{M} w_i(p^2(t_i) - at_i^{2M-2}).$$

This yields

$$\int_0^1 p^2(s)ds = \sum_{i=1}^{M} w_i(p^2(t_i) - at_i^{2M-2}) + \int_0^1 as^{2M-2}ds \tag{B.10}$$

$$\leq 1 - a\sum_{i=1}^{M} w_i t_i^{2M-2} + \int_0^1 as^{2M-2}ds, \tag{B.11}$$

where the inequality holds because $\sum_{i=1}^{M} w_i p^2(t_i) \leq 1$ for $\max_{1\leq i\leq M} |p(t_i)| \leq 1$. Now using the quadrature (B.9) on a polynomial t^{2M-2} yields

$$\int_0^1 s^{2M-2} = \sum_{i=1}^{M} w_i t_i^{2M-2} - c(2M-2)!$$

which in combination with (B.11) gives

$$\int_0^1 p^2(s)ds \leq 1 - ac(2M-2)! \leq 1,$$

since $ac \geq 0$. This proves that $t_M \leq \|\mathbf{Q}\|_\infty \leq \sqrt{t_M} \leq 1$ is also true for the Gauss-Lobatto quadrature.

B.3. Conclusion

This shows that $t_M \leq \|\mathbf{Q}\|_\infty \leq \sqrt{t_M} \leq 1$ holds for nodes originating from the Gaussian quadrature that integrates polynomials of degree $2M-2$ exactly, and it is also true for the Gauss-Lobatto quadrature.

Bibliography

[1] J. Nievergelt, "Parallel methods for integrating ordinary differential equations," *Commun. ACM*, vol. 7(12), p. 731–733, 1964.

[2] G. Čaklović, R. Speck, and M. Frank, "A parallel-in-time collocation method using diagonalization: theory and implementation for linear problems," 2021. [Online]. Available: https://arxiv.org/abs/2103.12571v1

[3] T. Xiao, "Kinetic. jl: A portable finite volume toolbox for scientific and neural computing," *Journal of Open Source Software*, vol. 6, no. 62, p. 3060, 2021.

[4] G. Čaklović, "The infinity norm bounds and characteristic polynomial for high order RK matrices," 2022. [Online]. Available: https://arxiv.org/abs/2203.04086

[5] B. Fornberg, "Generation of finite difference formulas on arbitrarily spaced grids," *Math. Comp.*, vol. 51, pp. 699–706, 1988.

[6] M. J. Gander, "50 Years of Time Parallel Time Integration," in *Multiple Shooting and Time Domain Decomposition Methods*, ser. Contributions in Mathematical and Computational Sciences, T. Carraro, M. Geiger, S. Körkel, and R. Rannacher, Eds. Springer, Cham, 2015, vol. 9.

[7] B. W. Ong and J. B. Schroder, "Applications of time parallelization," *Computing and Visualization in Science*, vol. 23, no. 1-4, Sep. 2020.

[8] M. Schreiber, N. Schaeffer, and R. Loft, "Exponential integrators with parallel-in-time rational approximations for the shallow-water equations on the rotating sphere," *Parallel Computing*, vol. 85, pp. 56–65, 2019.

[9] W. C. Agboh, D. Ruprecht, and M. R. Dogar, "Combining Coarse and Fine Physics for Manipulation using Parallel-in-Time Integration," in *International Symposium on Robotics Research*, 2019. [Online]. Available: https://arxiv.org/abs/1903.08470

[10] J. Steiner, D. Ruprecht, R. Speck, and R. Krause, "Convergence of Parareal for the Navier-Stokes equations depending on the Reynolds number," in *Numerical Mathematics and Advanced Applications - ENUMATH 2013, Lecture Notes in Computational Science and Engineering*, A. Abdulle, S. Deparis, D. Kressner, F. Nobile, and M. Picasso, Eds., vol. 103. Springer International Publishing, 2015, p. 195–202.

[11] A. T. Clarke, C. J. Davies, D. Ruprecht, and S. M. Tobias, "Parallel-in-time integration of kinematic dynamos," *Journal of Computational Physics: X*, vol. 7, 2020.

[12] S. Friedhoff, J. Hahne, and S. Schöps, "Multigrid-reduction-in-time for Eddy Current problems," in *Proceeding in Applied Mathematics and Mechanics*, 2019.

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

[14] J. L. Lions and Y. Maday and G. Turinici, " A "parareal" in time discretization of PDE's," *Comptes Rendus de l'Académie des Sciences, Série I*, vol. 332 (7), p. 661–668, 2001.

[15] K. Burrage, "Parallel methods for ODEs," *Advances in Computational Mathematics*, vol. 7, pp. 1–3, 1997.

[16] M. J. Gander and S. Vandewalle, "Analysis of the Parareal Time-Parallel Time-Integration Method," *SIAM Journal on Scientific Computing*, vol. 29, no. 2, pp. 556–578, 2007.

[17] M. J. Gander and E. Hairer, "Nonlinear Convergence Analysis for the Parareal Algorithm," in *Domain Decomposition Methods in Science and Engineering XVII*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 45–56.

[18] M. J. Gander, "Analysis of the parareal algorithm applied to hyperbolic problems using characteristics," *SeMA Journal: Boletín de la Sociedad Española de Matemática Aplicada*, no. 42, pp. 21–36, 2008.

[19] D. Ruprecht, "Wave propagation characteristics of Parareal," *Computing and Visualization in Science*, vol. 19(1), pp. 1–17, 2018.

[20] X. Dai and Y. Maday, "Stable Parareal in Time Method for First and Second-Order Hyperbolic Systems," *SIAM Journal on Scientific Computing*, vol. 5(1), p. A52–A78, 2013.

[21] M. J. Gander and M. Petcu, "Analysis of a Krylov Subspace Enhanced Parareal Algorithm for Linear Problem," *ESAIM*, vol. 25, p. 114–129, 2008.

[22] D. Ruprecht and R. Krause, "Explicit parallel-in-time integration of a linear acoustic-advection system," *Computers & Fluids*, vol. 59(0), p. 72–83, 2012.

[23] J. Angel, S. Götschel, and D. Ruprecht, "Impact of spatial coarsening on Parareal convergence," 2021. [Online]. Available: https://arxiv.org/abs/2111.10228

[24] T. M. Masthay and S. Perugini, "Parareal algorithm implementation and simulation in Julia," 2017. [Online]. Available: https://github.com/saverioperugini/Parareal-Implementation-and-Simulation-in-Julia

[25] LLNL, "XBraid: Parallel multigrid in time," 2014. [Online]. Available: https://computing.llnl.gov/projects/parallel-time-integration-multigrid

[26] "LibPFASST," version v1.1. [Online]. Available: https://github.com/libpfasst/LibPFASST

[27] R. Falgout, S. Friedhoff, T. Kolev, S. MacLachlan, and J. Schroder, "Parallel Time Integration with Multigrid," *SIAM Journal on Scientific Computing*, vol. 36 (6), p. C635–C661, 2014.

[28] J. Hahne and S. Friedhoff, "PyMGRIT: Multigrid-reduction-in-time in Python v1.0," 2020, release 1.0. [Online]. Available: https://github.com/pymgrit/pymgrit

[29] A. J. Howse, H. D. Sterck, R. D. Falgout, S. MacLachlan, and J. Schroder, "Parallel-In-Time Multigrid with Adaptive Spatial Coarsening for The Linear Advection and Inviscid Burgers Equations," *SIAM Journal on Scientific Computing*, vol. 41, no. 1, pp. A538–A565, 2019.

[30] O. A. Krzysik, H. D. Sterck, S. P. MacLachlan, and S. Friedhoff, "On selecting coarse-grid operators for Parareal and MGRIT applied to linear advection," 2019. [Online]. Available: https://arxiv.org/abs/1902.07757

[31] H. D. Sterck, S. Friedhoff, O. A. Krzysik, and S. P. MacLachlan, "Multigrid reduction-in-time convergence for advection problems: A Fourier analysis perspective," 2022. [Online]. Available: https://arxiv.org/abs/2208.01526v1

[32] H. D. Sterck, R. D. Falgout, O. A. Krzysik, and J. B. Schroder, "Efficient multigrid reduction-in-time for method-of-lines discretizations of linear advection," 2023. [Online]. Available: https://arxiv.org/abs/2209.06916v2

[33] H. D. Sterck, R. D. Falgout, and O. A. Krzysik, "Fast multigrid reduction-in-time for advection via modified semi-Lagrangian coarse-grid operators," 2022. [Online]. Available: https://arxiv.org/abs/2203.13382v1

[34] M. Emmett and M. Minion, "Toward an efficient parallel in time method for partial differential equations," *Communications in Applied Mathematics and Computational Science*, vol. 7, no. 1, pp. 105–132, 2012. [Online]. Available: https://doi.org/10.2140/camcos.2012.7.105

[35] A. Dutt, L. Greengard, and V. Rokhlin, "Spectral Deferred Correction Methods for Ordinary Differential Equations," *BIT Numerical Mathematics*, vol. 40, p. 241–266, 2000.

[36] M. Bolten, D. Moser, and R. Speck, "A multigrid perspective on the parallel full approximation scheme in space and time," *Numerical Linear Algebra with Applications*, vol. 24, 2077.

[37] M. Bolten, D. Moser, and R. Speck., "Asymptotic convergence of the parallel full approximation scheme in space and time for linear problems," *Numerical Linear Algebra with Applications*, vol. 25(6), p. e2208, 2018.

[38] P. Benedusi, M. Minion, and R. Krause, "An experimental comparison of a space-time multigrid method with PFASST for a reaction-diffusion problem," 2021. [Online]. Available: https://arxiv.org/abs/2006.12883

[39] R. Speck, "Algorithm 997: pysdc - prototyping spectral deferred corrections," *ACM Transactions on Mathematical Software*, vol. 45, no. 3, pp. 1–23, 2019.

[40] "PFASST++," version v0.5.0. [Online]. Available: https://github.com/Parallel-in-Time/PFASST

[41] M. J. Gander and S. Güttel, "PARAEXP: A Parallel Integrator for Linear Initial-Value Problems," *SIAM Journal on Scientific Computing*, vol. 35(2), p. C123–C142, 2013.

[42] T. S. Haut, T. Babb, P. G. Martinsson, and B. A. Wingate, "A high-order time-parallel scheme for solving wave propagation problems via the direct construction of an approximate time-evolution operator," *IMA Journal of Numerical Analysis*, vol. 36(2), p. 688–716, 2016.

[43] P. J. van der Houwen and B. P. Sommeijer, "Iterated Runge–Kutta Methods on Parallel Computers," *SIAM Journal on Scientific and Statistical Computing*, vol. 12, no. 5, pp. 1000–1028, 1991. [Online]. Available: https://doi.org/10.1137/0912054

[44] ——, "Parallel iteration of high-order Runge-Kutta methods with stepsize control," *Journal of Computational and Applied Mathematics*, vol. 29, no. 1, pp. 111–127, 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/037704279090200J

[45] A. Iserles and S. Nørsett, "On the Theory of Parallel Runge—Kutta Methods," *IMA Journal of Numerical Analysis*, vol. 10, no. 4, pp. 463–488, 1990. [Online]. Available: https://doi.org/10.1093/imanum/10.4.463

[46] R. Speck, "Parallelizing spectral deferred corrections across the method," *Computing and Visualization in Science*, vol. 19, p. 75–83, 2018. [Online]. Available: https://doi.org/10.1007/s00791-018-0298-x

[47] Y. Maday and E. M. Rønquist, "Parallelization in time through tensor-product space-time solvers," 2007.

[48] M. J. Gander, L. Halpern, J. Rannou, and J. Ryan, "A direct time parallel solver by diagonalization for the wave equation," *SIAM Journal on Scientific Computing*, vol. 41, no. 1, pp. A220–A245, 2019.

[49] D. Bertaccini and M. Ng, "Block ω-circulant preconditioners for the systems of differential equations," *CALCOLO*, vol. 40, p. 71–90, 2003.

[50] X. L. Lin and M. K. Ng, "An all-at-once preconditioner for evolutionary partial differential equations," 2020. [Online]. Available: https://arxiv.org/abs/2002.01108

[51] E. McDonald, J. Pestana, and A. Wathen, "Preconditioning and Iterative Solution of All-at-Once Systems for Evolutionary Partial Differential Equations," *SIAM Journal on Scientific Computing*, vol. 40, no. 2, pp. A1012–A1033, 2018.

[52] S. L. Wu, T. Zhou, and Z. Zhou, "Spectral Analysis for a Preconditioned All-at-Once System from First-Order and Second-Order Evolutionary Problems," *SIMAX*, vol. 43, no. 3, pp. 1331–1353, 2022.

[53] S. L. Wu, "Toward parallel coarse grid correction for the parareal algorithm," *SIAM Journal on Scientific Computing*, vol. 40, no. 3, pp. A1446–A1472, 2018.

[54] M. J. Gander and S. L. Wu, "Convergence analysis of a periodic-like waveform relaxation method for initial-value problems via the diagonalization technique." *Numerische Mathematik*, vol. 143, p. 489–527, 2019.

[55] ——, "A Diagonalization-Based Parareal Algorithm for Dissipative and Wave Propagation Problems," *SIAM J. Numer. Anal.*, vol. 58(5), p. 2981–3009, 2020.

[56] J. Liu and S. L. Wu, "A Fast Block α-Circulant Preconditioner for All-at-Once Systems From Wave Equations," *SIMAX*, vol. 41(4), pp. 1912–1943, 2020.

[57] S. L. Wu, T. Zhou, and Z. Zhou, "Stability implies robust convergence of a class of preconditioned parallel-in-time iterative algorithms," 2021. [Online]. Available: https://arxiv.org/abs/2102.04646

[58] M. J. Gander, J. Liu, S. L. Wu, X. Yue, and T. Zhou, "ParaDiag: parallel-in-time algorithms based on the diagonalization technique." [Online]. Available: https://arxiv.org/abs/2005.09158

[59] "Diagonalization-based Parallel-in-Time Methods." [Online]. Available: https://parallel-in-time.org/methods/paradiag.html

[60] G. Čaklović, "pyParaDiag," 2021. [Online]. Available: https://github.com/caklovicka/pyParaDiag

[61] R. Schöbel and R. Speck, "PFASST-ER: combining the parallel full approximation scheme in space and time with parallelization across the method," *Computing and Visualization in Science*, vol. 23, no. 1-4, Sep. 2020.

[62] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, ser. Springer Series in Computational Mathematics. Springer, Berlin, Heidelberg, 1996.

[63] H. Brunner, *Volterra Integral Equations An Introduction to Theory and Applications*, ser. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2017.

[64] E. McDonald, "All-at-once solution of time-dependent PDE problems," Ph.D. dissertation, University of Oxford, 2016.

[65] E. McDonald, J. Pestana, and A. Wathen, "Preconditioning and Iterative Solution of All-at-Once Systems for Evolutionary Partial Differential Equations," *SIAM Journal on Scientific Computing*, vol. 40(2), pp. A1012–A1033, 2018.

[66] R. E. Cline, R. J. Plemmons, and G. Worm, "Generalized inverses of certain Toeplitz matrices," *Linear Algebra and its Applications*, vol. 8(1), pp. 25–33, 1974.

[67] F. B. Hildebrand, *Introduction to numerical analysis*. New York : McGraw-Hill, 1956.

[68] M. J. Gander, L. Halpern, J. Ryan, and T. T. B. Tran, "A direct solver for time parallelization," in *Domain Decomposition Methods in Science and Engineering XXII*. Cham: Springer International Publishing, 2016, pp. 491–499.

[69] M. J. Gander, L. Halpern, J. Rannou, and J. Ryan, "A Direct Time-parallel Solver by Diagonalization for the Wave Equation," *SIAM J. Sci. Comput.*, vol. 41(1), pp. A220–A245, 2019.

[70] M. L. Minion, "Semi-Implicit Spectral Deferred Correction Methods For Ordinary Differential Equations," *Communications in Mathematical Sciences*, vol. 1, no. 3, pp. 471–500, 2003.

[71] D. Ruprecht and R. Speck, "Spectral Deferred Corrections with Fast-wave Slow-wave Splitting," *SIAM Journal on Scientific Computing*, vol. 38, no. 4, 2016.

[72] J. Huang, J. Jia, and M. Minion, "Accelerating the convergence of spectral deferred correction methods," *Journal of Computational Physics*, vol. 214, no. 2, pp. 633–656, 2006.

[73] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Englewood Cliffs, N.J. : Prentice-Hall, 1963.

[74] S. Götschel, M. Minion, D. Ruprecht, and R. Speck, "Twelve Ways to Fool the Masses When Giving Parallel-in-Time Results," in *Parallel-in-Time Integration Methods*, B. Ong, J. Schroder, J. Shipton, and S. Friedhoff, Eds. Cham: Springer International Publishing, 2021, pp. 81–94.

[75] L. Dalcin, P. Kler, R. Paz, and A. Cosimo, "Parallel Distributed Computing using Python," *Advances in Water Resources*, vol. 34, pp. 1124–1139, 2011.

[76] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. C. McInnes, R. Mills, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Users Manual," 2020. [Online]. Available: http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf

[77] P. Thörnig, "Jureca: Data centric and booster modules implementing the modular supercomputing architecture at jülich supercomputing centre," *Journal of large-scale research facilities JLSRF*, vol. 7, 10 2021.

[78] M. L. Minion, R. Speck, M. Bolten, M. Emmett, and D. Ruprecht, "Interweaving PFASST and parallel multigrid," *SIAM Journal on Scientific Computing*, vol. 37, pp. S244–S263, 2015. [Online]. Available: 10.1137/14097536X

[79] J. Zhang and Q. Du, "Numerical Studies of Discrete Approximations to the Allen–Cahn Equation in the Sharp Interface Limit," *SIAM Journal on Scientific Computing*, vol. 31, no. 4, pp. 3042–3063, 2009.

[80] G. A. Bird, *Molecular Gas Dynamics and the Direct Simulation of Gas Flows*. Oxford University Press, 1972.

[81] C. Mouhot and L. Pareschi, "Fast algorithms for computing the Boltzmann collision operator," *Mathematics of computation*, vol. 75, no. 256, pp. 1833–1852, 2006.

[82] T. Xiao, "A flux reconstruction kinetic scheme for the Boltzmann equation," *Journal of Computational Physics*, vol. 447, p. 110689, 2021.

[83] S. Götschel and M. L. Minion, "Parallel-in-Time for Parabolic Optimal Control Problems Using PFASST," in *Domain Decomposition Methods in Science and Engineering XXIV*, ser. DD 2017., 2018, pp. 363 – 371.

[84] S. L. Wu and J. Liu, "A Parallel-In-Time Block-Circulant Preconditioner for Optimal Control of Wave Equations," *SIAM Journal on Scientific Computing*, vol. 42, no. 3, pp. A1510–A1540, 2020. [Online]. Available: https://doi.org/10.1137/19M1289613

[85] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical Integration*. Springer, Berlin, Heidelberg, 2006, pp. 379–422.

[86] Dalcin, Lisandro and Mortensen, Mikael and Keyes, David E, "Fast parallel multidimensional FFT using advanced MPI," *Journal of Parallel and Distributed Computing*, 2019.

[87] G. A. Staff and E. M. Rønquist, "Stability of the parareal algorithm," in *Science and Engineering, Lecture Notes in Computational Science and Engineering*, e. a. R. Kornhuber, Ed., vol. 40. Springer, Berlin, 2005, p. 449–456.

[88] M. J. Gander and S. Vandewalle, "On the Superlinear and Linear Convergence of the Parareal Algorithm," in *Domain Decomposition Methods in Science and Engineering, Lecture Notes in Computational Science and Engineering*, O. Widlund and D. Keyes, Eds., vol. 55. Springer Berlin Heidelberg, 2007, p. 291–298.

[89] J. Hahne, S. Friedhoff, and M. Bolten, "Pymgrit: A python package for the parallel-in-time method mgrit," 2020. [Online]. Available: http://arxiv.org/abs/2008.05172v1

[90] G. Xie and Y. Li, "Parallel Computing for the Radix-2 Fast Fourier Transform," in *2014 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, 2014, pp. 133–137.

[91] G. HortonStefan and S. Vandewalle, "A Space-Time Multigrid Method for Parabolic Partial Differential Equations," *SIAM Journal on Scientific Computing*, vol. 16(4), p. 848–864, 1995.

[92] Jülich Supercomputing Centre, "JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing Centre," *Journal of large-scale research facilities*, vol. 5, no. A135, 2019.

[93] Y. Saad and M. H. Schultz, "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems," *SIAM J. Sci. Stat. Comput.*, vol. 7, no. 3, p. 856–869, 1986.

[94] B. W. Ong, R. D. Haynes, and K. Ladd, "Algorithm 965: RIDC Methods: A Family of Parallel Time Integrators," *ACM Trans. Math. Softw.*, vol. 43, no. 1, 2016.

[95] M. Emmett and M. L. Minion, "Toward an Efficient Parallel in Time Method for Partial Differential Equations," *Communications in Applied Mathematics and Computational Science*, vol. 7, pp. 105–132, 2012.

[96] C. Gräser, R. Kornhuber, and U. Sack, "Time discretizations of anisotropic Allen–Cahn equations," *IMA Journal of Numerical Analysis*, vol. 33, no. 4, pp. 1226–1244, 03 2013. [Online]. Available: https://doi.org/10.1093/imanum/drs043

[97] LLBL, "Website for PFASST codes," 2018. [Online]. Available: https://pfasst.lbl.gov/codes

[98] "References." [Online]. Available: https://parallel-in-time.org/references/

[99] J. C. Butcher, *Runge–Kutta Methods*. John Wiley & Sons, Ltd, 2016, ch. 3, pp. 143–331. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119121534.ch3

[100] ——, "On the implementation of implicit Runge-Kutta methods," *BIT Numerical Mathematics*, vol. 16, p. 237–240, 1976.

[101] M. J. Gander and S. Vandewalle, "Analysis of the Parareal Time-Parallel Time-Integration Method," *SIAM Journal on Scientific Computing*, vol. 29, no. 2, pp. 556–578, 2007.

[102] L. Xuelei, M. K. Ng, and H. Sun, "A Separable Preconditioner for Time-Space Fractional Caputo-Riesz Diffusion Equations," *Numerical Mathematics: Theory, Methods and Applications*, vol. 11, no. 4, pp. 827–853, 2018.