Efficient Massively Space-Time-Parallel Simulations with Adaptive Spectral Deferred Correction

Vom Promotionsausschuss der
Technischen Universität Hamburg
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation (Monographie)

von
Thomas Saupe

aus
Bamberg, Deutschland

2026

1. Gutachter: Prof. Dr. Daniel Ruprecht
2. Gutachter: Prof. Dr. Matthias Bolten

Tag der mündlichen Prüfung: 3. Dezember 2025

DOI: https://doi.org/10.15480/882.16360
ORCiD: https://orcid.org/0000-0002-4676-7659

Summary

Spectral Deferred Correction (SDC) is a method for numerically integrating initial value problems. The method iteratively generates solutions to fully implicit Runge-Kutta methods with forward substitution using low order solves. This allows great flexibility, for instance in terms of splitting techniques or inexact solves. Furthermore, various parallel-in-time extensions exist that parallelize the solution of a single time-step or solve multiple steps concurrently.

We propose two adaptive step size selection algorithms that tailor the ideas behind embedded Runge-Kutta methods to SDC. Both are completely generic and work only on intermediate values within the time-integration process. We show, with a range of experiments, that computational efficiency can be boosted significantly by employing these algorithms compared to standard SDC. Furthermore, we show that parallel-in-time adaptive SDC is competitive with state-of-the-art Runge-Kutta methods for stiff partial differential equations.

We also show that adaptivity increases the resilience against soft faults in SDC. Soft faults are unanticipated alterations of the data stored in memory, brought about, for instance, by environmental radiation. Iterative or adaptive methods inherently provide an elevated level of resilience, which is well known also in the context of the embedded Runge-Kutta methods that the adaptive step size selection is based on.

We then move on to port implementations of partial differential equations within the prototyping library pySDC to GPUs and make extensive space-time-parallel scaling tests. We find that the parallel-in-time extension diagonal SDC can help extend the scaling capabilities and allowed to run a Gray-Scott example on 3584 GPUs at decent parallel efficiency. Finally, we demonstrate that findings from the previous experiments translate to practical use via space-time-parallel production runs of Gray-Scott and Rayleigh-Benard convection using adaptive SDC.

Contents

1 Introduction

Computer simulations are an indispensable tool in modern science and engineering. They are used in a variety of ways, from studying the aerodynamics of a car without building a physical prototype, to performing experiments that cannot be done in the real world, like following along supernovae.

Developing and running a computer simulation is a very complex interdisciplinary undertaking [45]. First, a suitable model of the involved physics has to be developed using pen and paper. Then, in order to prepare for the computer, the model is reshaped into a discrete representation. Making sure that the discrete representation captures the physics described by the model can be an involved mathematical exercise. Finally, the discrete representation is translated into code for the computer to execute. Doing this in a fashion that makes efficient use of the hardware falls under computer science. In this thesis, we will address the latter two steps. To be precise, we will take models describing a range of physics, tweak the discrete solution methods, and discuss code implementations for modern hardware.

The models are typically formulated as partial differential equations (PDEs). That is to say, the equation contains derivatives of a function with respect to multiple variables and the function itself is the solution to the equation. In the problems we investigate here, one of the derivatives is always with respect to time, while derivatives with respect to spatial coordinates may be additionally included. The function is then integrated numerically, where the time variable is usually treated separately from other variables. The reason is the causality principle, which applies in the time-direction and allows time-stepping. In time-stepping, the state at a later time is determined purely from the current (and possibly previous) states, but it never includes future states. A similar approach does not generally work in the treatment of space coordinates, as there is no similar directionality to space.

The mathematical methods we investigate here are concerned with time-integration. Specifically, we work with the method "spectral deferred correction" (SDC) [47]. The deferred correction approach generally entails guessing the solution, approximating the error of the guess, refining the guess by subtracting the approximate error, and iterating this refinement procedure until the error is sufficiently small [125]. SDC is a special case, which involves very accurate numerical integration schemes.

One of the advantages of SDC compared to similar methods is that the work for computing a single time step can be distributed among multiple processors [141] and it can

be used within algorithms that allow to compute multiple time steps concurrently [50]. Algorithms that allow for concurrency are needed to utilize the compute power offered by modern machines because once the speed of individual processors started to plateau, the speed of computers was mainly increased by building networks of multiple processors [99].

The notion to distribute the work among multiple compute units predates the age of digital computers, when computations were carried out by hand. For instance, Lewis Fry Richardson, one of the pioneers of numerical weather prediction, envisioned around 1920 a grand factory employing 64,000 human computers, working in parallel to forecast the weather faster than real time [133]. Richardson's idea is based on splitting the spatial domain into cells and performing computations on individual cells in parallel, a concept that proved very successful and became ubiquitous in large numerical simulation codes.

However, there are two limits to this kind of parallelism. First, the computations depend on values from neighboring cells, which requires to wait for communication in between computations. Even with the high-speed interconnect in modern supercomputers, the time spent on communication is non-negligible [60]. When the work is distributed further and further, the individual computations become faster and faster to the point that eventually the majority of time is spent in communication and no speedup can be gained by further parallelism. Second, the number of cells determines the accuracy of the spatial discretization and limits how much work can be distributed.

Regardless, the number of processors in modern supercomputers continues to grow. For instance, the first supercomputer to reach the milestone of 10^{18} floating-point operations per second (1 exaFLOPS), Frontier [44], installed in 2021, employs roughly 9 million central processing unit (CPU) cores and 37.000 graphics processing units (GPUs) to achieve this performance. CPUs are optimised to perform individual operations as quickly as possible, whereas GPUs are optimised for throughput when performing similar operations on multiple data. The latter have proven to be very effective for applications in machine learning as well as computer simulations and are providing the majority of the compute power in modern supercomputers.

In order for numerical simulation codes to utilize exaFLOPS machines, spatial parallelism has to be combined with time-parallelism [56, 124], such as offered by SDC. However, SDC appears slightly more complicated than comparable schemes and remains a niche method. In this thesis, we will equip SDC with methods for adaptive selection of the step size, which is the main parameter controlling the accuracy of the time-integration, and subsequently demonstrate the vast capabilities of adaptive SDC as a time-stepping method on modern supercomputers using a range of experiments.

In the experiments we integrate two simple non-linear toy problems for highlighting certain characteristics and two complex non-linear PDEs that are representative of real-world simulations with adaptive SDC. The toy problems are the van der Pol oscillator, which originates from the study of vacuum tubes and the Lorenz attractor, which is a simplified model that captures some properties of equations used in numerical weather prediction. The first PDE example is the Gray-Scott system, which is a system of reaction-diffusion equations. This type of equation is used to model a range of phenomena, from the spread of diseases [20], over phase-field transitions [3] to chemical reactions [62]. The second PDE benchmark is Rayleigh-Benard convection, which models the behaviour of an incompressible fluid that is heated from below and cooled from above. This setup is often used when investigating convection, for instance in the context of atmospheric flows [36]

or magma-oceans [83].

To run the experiments, we use and expand the open source code `pySDC` [142]. `pySDC` is a highly modular library built for rapid prototyping of any SDC related research questions and includes many reference implementations of time-integration methods or PDEs. This allows researchers to assemble a simulation from a mixture of pre-implemented and novel building blocks and to focus on the details relevant to the research question at hand. It is implemented in Python, which is built to streamline development of complex code by combining various modules maintained by a community of developers [157]. While code written in Python itself can typically be executed only slowly, we employ modules such as `NumPy` [75] and `CuPy` [122] for the computationally expensive parts that interface to fast code written in other languages.

Multiple ways have been proposed previously to adapt the step size at runtime and therefore to adjust the accuracy of the SDC integrator to the requirements of the model [47, 67, 68, 78, 80, 130]. However, these methods are specific to certain models or adjust the accuracy only coarsely. Here, we propose two algorithms for adaptive step-size selection in SDC that allow to finely tune the accuracy and work for a wide range of models. We then perform a series of numerical experiments in order to showcase that these methods lead to increased efficiency in the integration process with all models under consideration. In particular, we compare to state-of-the-art methods from the ubiquitous family of Runge-Kutta time integrators and show that time-parallel SDC is competitive.

Next, we perform numerical experiments showing that the adaptive step size selection algorithms increase reliability of the simulations. It is already known that adaptive step size selection algorithms have this effect in the context of Runge-Kutta methods [13]. While individual processors work rather reliably, the sheer number of components in modern supercomputers makes them susceptible to errors caused by the environment [58]. For instance, radiation can carry sufficient energy to impact the memory and change the stored state. This is also the reason behind airplane and hospital modes on consumer devices. As supercomputers continue to grow in complexity, protecting the software against unintended interference becomes increasingly necessary and of particular importance in the exaFLOPS-era [1].

Then, we move on to test the performance of space-time-parallel adaptive SDC on large modern supercomputers. We develop parallel Python implementations for both CPUs and GPUs from the ground up and show how they respond to changing the number of compute units. In particular, we show that time-parallel SDC can extend speedup beyond the limits of spatial parallelism. For the Gray-Scott model, we are able to efficiently utilize 3584 GPUs, which is almost the entire JUWELS booster [90] machine, one of the most powerful public supercomputers in Europe at the time of writing. We then finish with simulations representative of large production runs of Gray-Scott and Rayleigh-Benard convection to showcase that adaptive SDC and our implementations can be used in practical settings.

We provide background information required to interpret the results formed in later chapters in chapter 2. We then present step size adaptive extensions to SDC in chapter 3. We show experiments of adaptive SDC on computational efficiency in chapter 4 and on resilience in chapter 5. Then, we discuss implementations on GPUs and measure parallel performance in chapter 6. Finally, we present massively parallel production runs in chapter 7 and draw conclusions in chapter 8.

2 Preliminaries

We will start with a motivation and basic introduction to parallel computing, followed by a discussion of reliability of computers. Then, we move on to mathematical concepts for computer simulations, namely time integration methods, adaptive step size selection, and spectral methods for space discretization. Afterwards, we introduce the benchmark problems we use in experiments, the GPU hardware that we develop code for, the codes we use and modify, and finally the hardware we run the experiments on.

2.1 Parallel Computing

Before introducing the simulation methods we will employ later on, we start by motivating the use of parallel computing and discussing some of the basics of parallel programming.

2.1.1 Evolution of hardware towards parallel computing

In the early days of semiconductor based computers, compute power was increased between generations by increasing the number of transistors on a chip and decreasing their size. One of the founders of Intel, Gordon Moore, established Moore's law as a sort of self-fulfilling prophecy. It states that the number of transistors on a chip would double every two years [69]. The actual development over the past 50 years is shown in Figure 2.1.

Moore's law led to exponential increase in single-thread compute power over time until about 2005. After that, progress in single-thread performance slowed down severely, as it became increasingly difficult to manufacture smaller transistors and fit more on a single chip. Instead, the industry moved to multi-core architectures in order to sustain a modified version of Moore's law. Rather than doubling the number of transistors on a single chip, the number of transistors was doubled on the processor, which now included multiple cores.

This is an imperfect solution, however. While the compute power measured in floating point operations per second (FLOPS) can be easily increased this way, algorithms need to adapt to distributing the work among the multiple cores in order to harness the available compute power. Some algorithms are inherently serial and need to be altered substantially with many added operations, while other algorithms are well suited to parallelization. We will give examples of both when discussing particular algorithms in later sections. Virtually all parallel algorithms require communication between tasks and the interconnect is often

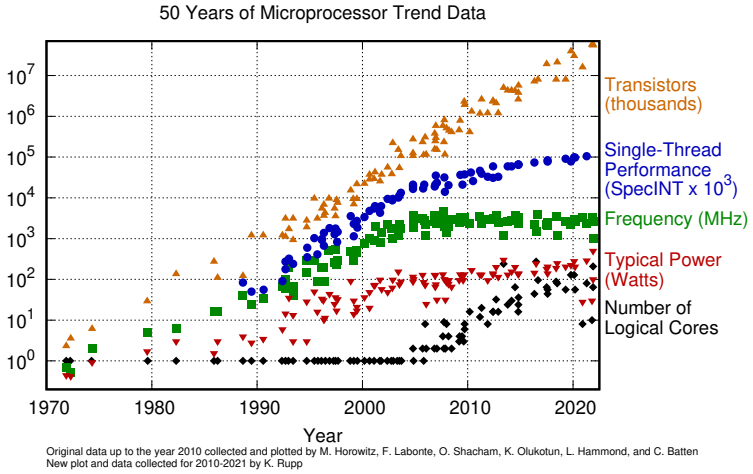


Figure 2.1: Development of compute power of microprocessors from 1970 to 2020. The number of transistors grows exponentially over time, as dictated by Moore's law. Starting around 2005, the transistors are mainly added via more cores at similar single-thread performance as the last generation. In order to translate this into faster code, parallel algorithms have become increasingly necessary. This figure is taken from [134] under CC BY 4.0 licence.

a bottleneck. The resulting sweet spot for the number of cores is specific to the algorithm and the machine and is difficult to know a priori.

2.1.2 Evaluating the performance of parallel algorithms

In order to determine how well the parallelization of an algorithm works, its strong scaling and weak scaling characteristics have to be measured on the target machine. Strong scaling refers to how the run time responds when solving the same problem with different number of processes. Here, the quantities typically recorded are the speedup S with

$$S(N) = \frac{t(1)}{t(N)}, \tag{2.1}$$

where N is the number of processes and $t(N)$ is the run time needed to solve the problem with N processes, and the parallel efficiency E with

$$E(N) = \frac{S(N)}{N}. \tag{2.2}$$

The ideal outcome of parallelization is that the work is distributed and performed perfectly simultaneously on all processors with no process waiting for another at any time. In this case, the speedup would be

$$S_{\text{ideal}}(N) = \frac{t(1)}{t(1)/N} = N \tag{2.3}$$

and the parallel efficiency would be 100 %.

Weak scaling is different from strong scaling in that the size of the problem is increased in tandem with the number of processes, such that the work per task is kept as constant

as possible. The ideal scaling in this case is that the run time is the same for any problem size.

In practice, the maximum attainable speedup is limited by operations in the algorithm that cannot be parallelized and communication between tasks. With increasing number of tasks, individual messages often have to traverse longer paths in the network, possibly at reduced bandwidth.

2.1.3 Parallel programming using the Message Passing Interface

We give a very brief introduction to parallel programming here. This can be generally grouped in two categories. The first is shared memory parallelisation, where no explicit communication is required because any task can read any part of memory, regardless of which task has written the respective data. Second is distributed memory parallelisation, where the tasks operate with distinct memory and need to explicitly communicate any data that is to be shared.

The second approach is often tackled using the Message Passing Interface (MPI). As we will later develop code in the Python language, where shared memory parallelisation is supported only in very limited fashion at the time of writing, we restrict this discussion to programming with MPI which can be efficiently integrated in Python using the `mpi4py` [43] library.

MPI is a standard that is administered by the MPI forum and has been first released in 1994 [111]. It defines an interface that programmers can use to communicate data in various patterns. Multiple implementations of MPI exist, which have to support the interface dictated by the standard, but which are free to implement communication algorithms however they like. For instance, OpenMPI is an extremely portable open source implementation, whereas Intel MPI is an implementation that claims to outperform OpenMPI specifically on Intel hardware.

MPI is built around so called communicators, which are a collection of tasks that can send messages to one another. Within each communicator, each task is assigned a unique number, called rank, which is used to identify the task relative to other tasks. All tasks receive the same code and are then instructed to do different parts of the computation purely based on the rank.

MPI supports both peer-to-peer communication, as well as collective communication. We will now list the collective communication patterns that we need in this thesis.

All-to-all. During all-to-all communication, all ranks exchange data with all other ranks, as illustrated in Figure 2.2. In MPI, there are different versions based on the complexity of the communication pattern. The simplest is ALLTOALL, which can be used when all data to be communicated between ranks are the same size and of the same datatype. The next more elaborate method is ALLTOALLV, which requires the data to be of the same type, but allows different sizes in individual communications. The most generalized version is ALLTOALLW, which allows different sizes of data and datatypes per task. Note that MPI implementations of ALLTOALLW are often less optimized than for the more commonly used ALLTOALLV and may require substantially longer time for communication.

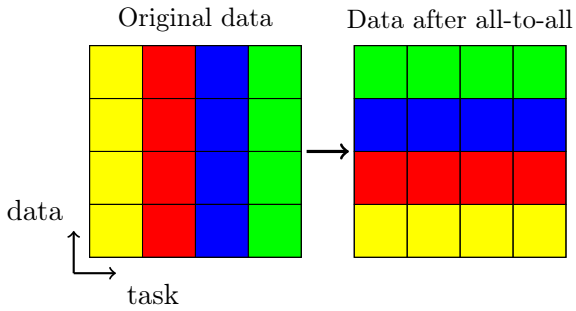


Figure 2.2: Illustration of all-to-all communication via an example with $N = 4$ tasks. Each task starts out with N packets of data, which are all of the same color in this illustration (see left panel). During all-to-all communication, each task exchanges data with every other task, such that each task ends up with data packets of every of the N colors (see right panel).

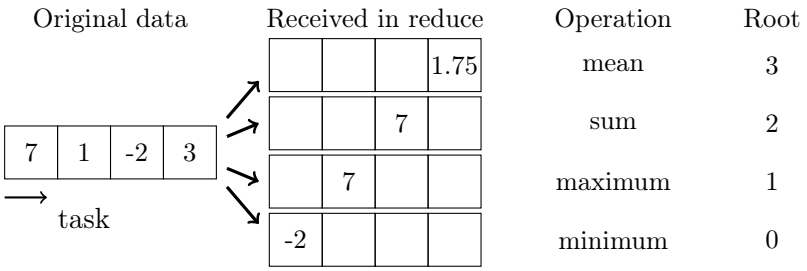


Figure 2.3: Illustration of various reduce communication operations via examples with $N = 4$ tasks. Before the communication, every task has some data, one integer in these examples. After the communication, the root task carries the result of some computation on the distributed data.

Reduce. Reduce operations perform a computation during the communication. Examples are a sum or the maximum of all elements in the communication. After the reduce operation is complete, the result of the computation is stored on a single task, which is referred to as "root" in the language of MPI. The procedure is illustrated in Figure 2.3. There is also ALLREDUCE, which communicates the result of the reduce operation to all tasks.

Broadcast. A broadcast operation is used to send data from one rank to all others. We illustrate the operation in Figure 2.4.

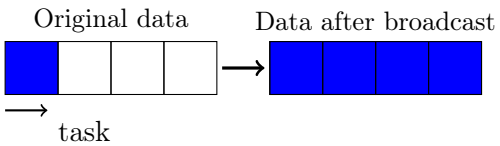


Figure 2.4: Illustration of broadcast communication operations via an example with $N = 4$ tasks. After the communication, all tasks carry data (marked in blue in this example) that resided on a single task before.

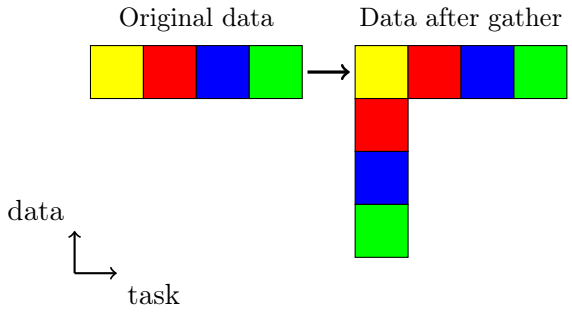


Figure 2.5: Illustration of a gather communication operation via an example with $N = 4$ tasks. Each task starts out with individual data, marked here using different colors. After the communication, one task has access to all of the previously distributed data.

Gather. In a gather operation, data that is distributed among all tasks is gathered on a single task, as illustrated in Figure 2.5. There is also ALLGATHER, where the result of the gather operation is communicated to all tasks.

2.2 Resilience

We first motivate the need for resilient computing following [12]. Then, we introduce a few common strategies for achieving resilience in practice. We group these into strategies that work on the system level and strategies that work on the application level. System level strategies are completely generic, but application level strategies allow to protect selectively which can reduce the overhead stemming from fault protection. See [1] for a more comprehensive overview on resilience.

2.2.1 A brief history of (un-)reliability in computing

While computers have the reputation of performing arithmetic without errors in the general public, this notion is rather new. A prominent example of the poor reputation of electronic computation well into the sixties features the first American to perform a full rotation around the Earth in space, John Glenn. He famously trusted crucial flight information computed by an electronic computer only after it was confirmed by one of NASA's legacy human computers equipped only with a slide rule [86].

In the early days of computing, the vacuum tubes that provided the basis for implementations of logic in hardware were prone to failure and quite unreliable [112]. In particular, they shared many problems with light bulbs, notably burning out, as well as attracting moths, requiring them to be regularly "debugged" [132].

The replacement of vacuum tubes by semiconducting transistors, following their invention in the late forties, promised to alleviate reliability concerns. However, a new problem manifested itself in the "tyranny of numbers", which was the term given to the issue that engineers found themselves unable to manufacture the complex circuits that they designed on paper. Blueprints for computers required millions of small components which had to be assembled by hand in a very error-prone process. Computer chips became reasonably reliable only following the invention of the integrated circuit in the late fifties. Rather

than assembling each component separately, the components are essentially printed on a monolithic block of silicon.

Today, as the trend in high performance computing (HPC) systems is for components to keep shrinking and their numbers to keep rising, the probability of a fault somewhere in the system is, again, increasing and is a major concern for exa-scale systems [1]. Failure of a single hardware component can cause the entire system to fail. As an example, a faulty diode in an integrated circuit in the launch vehicle of the Mariner 8 mission resulted in a failed launch. Instead of becoming the first spacecraft to orbit another planet and mapping out Mars, the probe ended up on the bottom of the ocean [87].

2.2.2 Sources of faults

Faults can originate from a variety of sources such as damage to hardware components [139] or electromagnetic radiation [97], which has lead to hospital and airplane modes on portable consumer devices. As hardware components continue to shrink, the operating voltage has to be reduced in order to mitigate heat concerns, which lowers the safety threshold and increases susceptibility to bit flips [52]. Additionally, bit flips can be brought about with malicious intent. For instance, electrical interaction between neighboring memory addresses can be exploited by accessing data repeatedly in order to corrupt the data stored in adjacent memory addresses in a so called row hammer attack [91].

The type of fault we address in this thesis is often referred to as silent data corruption. Note that in the resilience literature, this is often abbreviated as SDC, but here SDC exclusively refers to spectral deferred correction. Silent data corruption is characterised as a fault that is not detected by the system. This typically comes in the form of isolated, non-persistent bit flips. Real-world studies suggest that such faults occur multiple times per day in modern HPC centers [139]. In particular, also in caches that are insufficiently protected by hardware error correction codes [58]. We do not consider faults of adversarial nature, which may lead to a large concentration of faults in some region of memory.

2.2.3 Hardware and system level resilience strategies

Resilience is often addressed by the system outside the application. While the system has less information about what data is critical to protect than the application, this has the advantage of being generic. The main way that this is typically implemented is via error correction codes (ECCs), see e.g. [114]. These use additional bits to encode the data in a redundant fashion, such that alterations to the stored bits by a fault can be detected. Certain ECCs even allow to correct faults. Using ECCs in hardware memory is standard practice [32], and software exists, which can use even more elaborate ECCs to further protect on the system level, e.g. [52].

Beyond this, it is possible to store the entire system state to disk, which allows to continue the computation from an uncorrupted state after a fault has been detected or the system has crashed. For instance, BLCR [74] is a software that allows to store the system state with MPI-parallel codes, which makes this functionality available on HPC systems.

In very critical applications, the hardware can be replicated itself. This is common practice in space-based computing [41], but can also be employed in HPC applications, e.g. [27]. While this can give very good resilience with little downtime, it is typically prohibitively expensive in the HPC context.

2.2.4 Software resilience strategies

Resilience strategies at the software level are often referred to as algorithm based fault tolerance (ABFT). ABFT can be generally grouped into two categories. Error aware ABFT schemes operate by explicitly detecting faults and then applying a correction scheme, whereas error oblivious ABFT methods give the correct result regardless of the presence of faults, as long as the fault rate is not extremely high.

Error aware ABFT comes in various forms, for instance by targeted checksums within basic linear algebra (BLAS) routines [163] or in fast Fourier transforms (FFTs) [104]. Other schemes are developed specific to the problem, such as checking bounds of the solution of the problem which can be computed beforehand [118]. Some schemes are tailored to the method, such as Hot Rod [65], which protects time-stepping schemes via error estimates. We will discuss Hot Rod in more detail in section 5.2, where we extend the method to SDC.

Typical examples of error oblivious ABFT are iterative methods, e.g. [5, 70]. The impact of a soft fault within an iterative method is akin to starting from poor initial guess. As convergence from any initial guess is not generally guaranteed, they are typically not perfectly resilient. Nevertheless, it has been shown that SDC with adaptive iteration number, being an iterative method, has good resilience naturally [63]. The same is true for step size adaptive methods, which we will introduce in section 2.4. They refine the resolution if the error was deemed unacceptably large. The adaptive scheme is not aware whether this is due to the discretization or due to a fault. It has been shown that examples of the class of adaptive time stepping methods called embedded Runge-Kutta methods are therefore naturally resilient [13]. As we develop similar adaptivity for SDC in chapter 3, we assume this to boost resilience similarly, which we show to be the case in chapter 5.

2.3 Time integrators for initial value problems

In this section, we introduce the fundamentals of computer simulation. We will start with a brief description of the general problems we are interested in solving, which take the form of differential equations. Then, we will present methods for integrating these in time before discussing a way of translating spatial derivatives into a form that we can solve in section 2.5.

2.3.1 Differential equations

A differential equation relates derivatives of a function to some right hand side and the solution to the equation is the function itself. Differential equations are a powerful tool for the mathematical description of natural phenomena. This was first discovered by Newton, whose fundamental laws of motion are denoted as such [55, Chapter 1]. See the tome by Hairer and Wanner [72] for a thorough introduction to differential equations and how to solve them, as well as the textbooks [55, 137] for information about modelling physical phenomena using differential equations.

In order to translate a physical problem into a differential equation, one only needs to know how the problem reacts to changes in each variable and not the evolution or final state of the problem, which is then the solution of the differential equation. We give a simple example with modelling the evolution of the number of rabbits n_R in a population

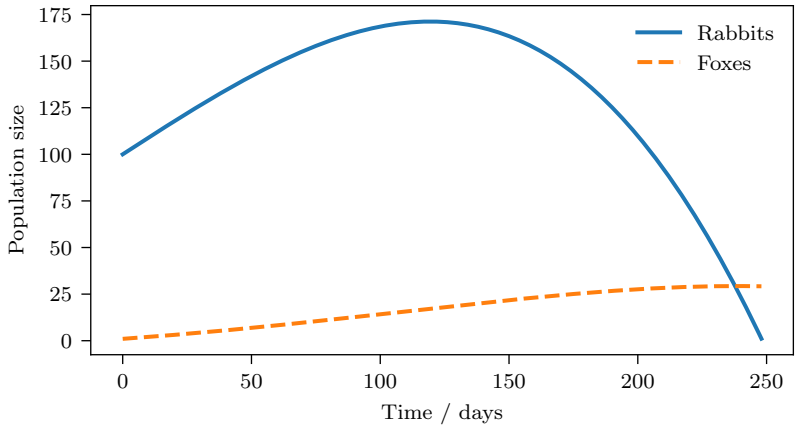


Figure 2.6: Numerical solution of the model for the evolution of rabbit and fox populations from Equation 2.5. We choose a birth rate of $\lambda = 0.1$ rabbits per rabbit and day, each fox eats one rabbit per day ($\sigma = 1$ rabbits per fox and day) and every 100 days another fox is attracted to the area per rabbit that is not eaten by another fox ($\gamma = 0.01$ foxes per rabbit and day). Starting with 100 rabbits and 1 fox, the rabbit population increases initially, but peaks at about 120 days as the fox population grows and eventually disappears entirely after 250 days.

over time, following the scheme from [55, Chapter 9]. The simplest model dictates that the number of rabbits that is birthed per unit of time is only proportional to the number of rabbits in the current population. This can be written as

$$\partial_t n_R = \lambda n_R, \tag{2.4}$$

with λ a population-specific constant. This equation is easily solved analytically with the solution $n_R(t) = n_R(t = 0) \exp(\lambda t)$.

A slightly more complex model may include a predator. We now seek to model the evolution of a fox population next to the rabbit population. The evolution of the number of rabbits is now dependent on the current number of rabbits, which generate new rabbits, and on the current number of foxes, which eat rabbits. The number of foxes n_F evolves according to the amount of surplus food available. We write this down as a coupled system of equations

$$\begin{aligned} \partial_t n_R &= \lambda n_R - \sigma n_F \\ \partial_t n_F &= \gamma(n_R - n_F), \end{aligned} \tag{2.5}$$

with σ and γ population-specific constants. This system of equations is already no longer easily solved analytically, even though the model is still extremely simplified. We show an example of a numerically obtained solution to this problem in Figure 2.6. While it is easy to write down a model as a differential equation, solving this equation can be very hard and often requires numerical methods.

Differential equations are broadly separated into ordinary differential equations (ODE), where the equation contains derivatives with respect to a single variable, and partial differential equations (PDE), which contain derivatives with respect to multiple variables, see e.g. [149]. The earlier rabbit example is an ODE because only derivatives with respect to time are present in the equation. PDEs generally describe multi-dimensional problems, like

the spatial distribution of rabbits rather than the overall population number. For instance, assuming that rabbits perform random walks, we can model the population in absence of predators as

$$\partial_t n_R = \lambda n_R^2 + \Delta n_R, \tag{2.6}$$

with Δ the Laplace operator [137]. Note that the number of births is proportional to n_R^2 now to reflect the fact that two rabbits are needed at the same position in space for generating additional rabbits.

PDEs pose significant challenges because the spatial derivatives need to be evaluated and possibly also inverted. When a computer is to be employed for finding the solution, the system must be discretized. Often, discretized versions of PDEs take the form of large systems of equations, which require immense computational resources. We will now discuss time integration of generic systems of equations, before returning to the topic of representing spatial derivatives in PDEs in section 2.5.

2.3.2 The initial value problem

Time dependent problems are typically posed as initial value problems (IVPs). An IVP is a combination of a differential equation determining the evolution in time and an initial condition

$$u_t = f(u,t), \quad u(0) = u_0, \tag{2.7}$$

where u is the solution, the subscript $(\cdot)_t$ marks the derivative with respect to time, f describes the temporal evolution and u_0 is the initial condition. The IVP has a unique solution if f is Lipshitz continuous [72, Theorem 7.3 c)], that is

$$\|f(u) - f(v)\|_\infty \leq L\|u - v\|_\infty \quad \forall u, v \in \mathbb{C}^n, \tag{2.8}$$

for some $L \geq 0$. Note that in the above definition, we have absorbed the dependence on time into the solution variables without loss of generality.

The solution of the IVP is the integral

$$u(T) = u_0 + \int_0^T f(u,t)dt. \tag{2.9}$$

Depending on the problem, a closed form solution may be unattainable and numerical integration methods need to be employed. In the remainder of this section, we introduce the numerical integration methods that we employ for solving IVPs throughout this thesis.

2.3.3 Runge-Kutta methods

We briefly sketch how Runge-Kutta methods are derived, roughly following [72]. The most simple Runge-Kutta methods are

$$u_{n+1} = u_n + \Delta t f(u_n, t) \qquad \text{(explicit Euler)}, \tag{2.10}$$

$$u_{n+1} = u_n + \Delta t f(u_{n+1}, t + \Delta t) \qquad \text{(implicit Euler)}, \tag{2.11}$$

where u_n is the solution at time t and u_{n+1} is the solution at $t+\Delta t$. This can be motivated via the limit definition of the derivative

$$\partial_t u(t) = f(u,t) = \lim_{\Delta t \to 0} \frac{u(t + \Delta t) - u(t)}{\Delta t}. \tag{2.12}$$

On the computer, we settle for finite Δt and rearrange to arrive at explicit Euler. Implicit Euler can be similarly derived by considering the backwards limit

$$f(u,t) = \lim_{\Delta t \to 0} \frac{u(t) - u(t - \Delta t)}{\Delta t} \tag{2.13}$$

with a change of variable.

Explicit Euler is typically simple to implement, with the evaluation of f the only complicated part. Implicit Euler requires to invert f, which ranges in complexity from division to solving non-linear systems, based on the problem to be integrated.

These are first order methods, which means the global error e at the end of the simulation is halved if Δt is halved. An order p integrator, on the other hand, satisfies

$$e \propto \Delta t^p. \tag{2.14}$$

Note that the error of a single time step, which is called local error, scales with

$$e_{\text{local}} \propto \Delta t^{p+1}, \tag{2.15}$$

for an order p accurate method.

Higher order Runge-Kutta methods assemble multiple stages k in a special way. The stages are evaluations of f at intermediate solutions which are obtained with modified Euler steps. The order of accuracy is computed by Taylor expansion of the IVP (Equation 2.7) in the variables t and u:

$$u(t + \Delta t) = u_n + \Delta t f(u_n, t) + \frac{\Delta t^2}{2} \left(f_t + f_u f\right)(u_n, t) + \mathcal{O}(\Delta t^3), \tag{2.16}$$

where $u_n = u(t)$, the subscripts t and u denote derivatives and we substituted $u_t = f$. We can immediately see that the explicit Euler method is first order accurate because it is precisely the Taylor expansion of the IVP to first order. Note that implicit Euler is the Taylor expansion of the solution backward in time truncated after first order.

Consider now the following Taylor expansion

$$\begin{aligned} y_2 &= u_n + \Delta t f\left(u_n + \Delta t/2 f(u_n, t), t + \Delta t/2\right) \\ &= u_n + \Delta t f(u_n, t) + \frac{\Delta t^2}{2}(f_t + f_u f)(u_n, t) + \mathcal{O}(\Delta t^3), \end{aligned} \tag{2.17}$$

which coincides with the Taylor expansion of the IVP to second order. We can use this to construct a second order two-stage explicit Runge-Kutta method,

$$\begin{aligned} k_1 &= f(u_n, t) \\ k_2 &= f(u_n + \Delta t/2 k_1, t + \Delta t/2) \\ u_{n+1} &= u_n + \Delta t k_2. \end{aligned} \tag{2.18}$$

The method is defined by the nodes c_i, which are the times of the intermediate solutions, the weights b_i, which are the values that determine the linear combination of stages that is the solution to the step, and the Runge-Kutta matrix A, which defines the weights inside the stages. These values are usually written as a Butcher tableau [24, Section 23.2]

$$\begin{array}{c|c} c & A \\ \hline & b \end{array}$$

with the Butcher tableau for Equation 2.18 reading

$$\begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2} & \\ \hline & 0 & 1 \end{array}.$$

Note that zero-values are usually omitted in the A matrix.

The Taylor expansion can be carried out to higher order and an arbitrary number of terms can be canceled to construct an s-stage p-order Runge-Kutta method. However, construction of these methods is non-trivial due to the rapidly growing number of order conditions. In particular, there are a number of desirable properties next to the order, which further complicate the process.

Runge-Kutta methods are generally grouped into three major categories. *Explicit* methods (ERK) do not require inversion of f, as in explicit Euler. *Fully implicit* methods have dense A, meaning stages are coupled and have to be solved for simultaneously. *Diagonally implicit* methods (DIRK) are lower triangular and can be solved with forward substitution, such that f has to be inverted at only one intermediate point at a time. Implicit Euler is the only Runge-Kutta method that is both fully implicit and diagonally implicit. As ERK and DIRK methods are ubiquitous but fully implicit methods are less common, we use the acronym RKM to refer to ERK and DIRK methods only for the remainder of this thesis.

Stability. One of the fundamental considerations is stability [71, Chapter IV.2]. This is usually quantified via the Dahlquist equation

$$u_t = \lambda u, \quad \lambda \in \mathbb{C}, \quad u(t=0) = 1, \tag{2.19}$$

which has exact solution

$$u^* = \exp(\lambda t).$$

The domain of stability of a time-stepping scheme is now the set of λ with negative real part for which the absolute value of the numerical solution decreases in time. For explicit Euler, we have a small circle on the negative real axis

$$1 \leq \|u_1\| = \|1 + \Delta t \lambda\| \qquad \text{(explicit Euler)}, \tag{2.20}$$

whereas for implicit Euler, the domain of stability includes the entire left complex half-plane

$$1 \leq \|u_1\| = \frac{1}{\|1 - \Delta t \lambda\|} \qquad \text{(implicit Euler)}. \tag{2.21}$$

While higher order explicit methods often have larger stability domains than explicit Euler, none of them encompass the entire left half plane which many implicit methods do. This is the main reason to choose an implicit method. If f has large eigenvalues $\|\lambda_i\| \gg 1$, Δt must be chosen very small for all $\Delta t \lambda_i$ to fall within the domain of stability of an explicit method. The range of magnitudes of the eigenvalues is usually referred to as stiffness, i.e. a problem with both large and small eigenvalues is stiff, and a problem where all eigenvalues are similar in size is non-stiff [8].

Other desirable properties. For solving stiff problems, a very popular class of methods is called singly-diagonally implicit (SDIRK). These are DIRK methods that have the same value on the diagonal of A, which means parts of the implicit solves may be reused.

Another desirable property for RKM is called stiff accuracy, where the last line of the Runge-Kutta matrix is equal to the nodes

$$A_{sj} = b_j, \tag{2.22}$$

and which means the solution that is entered into the right hand side evaluation in the last stage is directly the solution to the step. This saves work on the one hand because the weights need not be applied separately, but can also be used to solve some problems with simple algebraic constraints (DAEs) [6].

Many problems capture multiple physical mechanisms that have different properties for time-stepping. Splitting techniques can be employed to integrate individual terms in the resulting PDE formulation separately from each other. Corresponding RKM are called additive Runge-Kutta (ARK) methods and consist of multiple coupled Butcher tableaux. We will later use methods that realize implicit-explicit (IMEX) splitting, where the problem can be separated into a stiff and a non-stiff part that are integrated with an ARK method that consists of a pair of one implicit and one explicit method.

The coupling between the tableaux introduces additional order conditions. Their number grows rapidly with the order of the scheme, particularly when the nodes and weights are allowed to be different. For instance, for a fifth order IMEX method, there are 15 additional conditions if the weights and nodes are shared between the methods, 54 if only the nodes are shared, 110 if only the weights are shared and 252 if nothing is shared [126, Table 1]. In the literature, IMEX RKM can readily be found up to order five, which share the nodes and weights [89]. However, shared nodes between a DIRK and an ERK method means not both can be stiffly accurate.

2.3.4 Spectral Deferred Correction

Spectral deferred correction (SDC) [47] is a method that iteratively solves fully implicit Runge-Kutta methods. We will now give a brief derivation of the method and then comment on the similarities and differences to the previously discussed RKM. Note that different derivations exist which focus on different properties of the method. We start with one that traces the original idea and then comment on a more modern derivation that allows greater flexibility.

We begin by integrating the initial value problem with respect to time over the interval $t = 0$ to $t = \Delta t$

$$u(t) = u_0 + \int_0^t f\left(u(s)\right) ds, \quad t \in [0, \Delta t]. \tag{2.23}$$

Note that we restrict this discussion to autonomous scalar problems, i.e. $u(t)$, u_0, $f(u)$ $\in \mathbb{R}$. The method can be easily applied to problems that do not meet these criteria, but the notation is more convoluted.

We discretize the integral using a numerical quadrature rule. To this end, we start by discretizing the interval with a set of M collocation nodes $0 \leq \tau_m \leq 1$, rescaled by Δt and 0 and use polynomial interpolation to approximate f. Interpolation is done with Lagrange

polynomials [16]

$$l_j^\tau(t) = \frac{\prod_{i=1,i\neq j}^{M}(t-\tau_i)}{\prod_{i=1,i\neq j}^{M}(\tau_j-\tau_i)} \tag{2.24}$$

which give a degree M continuous approximation together with the f evaluations at the collocation nodes

$$f(u(t)) \approx \sum_{j=1}^{M} f(u(\Delta t\tau_j))l_j^\tau(t/\Delta t). \tag{2.25}$$

The Lagrange polynomials are then integrated in order to obtain quadrature weights

$$q_{mj} = \int_0^{\tau_m} l_j^\tau(s)ds, \tag{2.26}$$

which we use to approximate the solution at the quadrature nodes by

$$u_m := u_0 + \Delta t \sum_{j=1}^{M} q_{mj} f(u(\Delta t\tau_j)) \approx u(\Delta t\tau_m). \tag{2.27}$$

We call Equation 2.27 the collocation problem and simplify the notation by using vectors $\vec{u} = (u_m)^T$, $\vec{u}_0 = (u_0, ..., u_0)^T$ and $F(\vec{u}) = (f(u_m))^T$ to

$$\vec{u} = \vec{u}_0 + \Delta t Q F(\vec{u}), \tag{2.28}$$

where $Q \in \mathbb{R}^{M\times M}$ is the quadrature matrix containing the weights q_{mj}. While the interpolation is order M accurate, the solution to the collocation problem is up to order $2M$ for certain types of nodes, such as $2M-2$ for Gauß-Lobatto, $2M-1$ for Gauß-Radau and $2M$ for Gauß-Legendre at the right boundary with only M nodes [71, Theorems 5.2, 5.3 and 5.5].

Note that the collocation nodes do not necessarily include the end point. In this case, the solution at the end of the interval has to be computed via

$$\begin{aligned} u_{\text{end}} &= u_0 + \vec{b}F(\vec{u}) \\ b_i &= \int_0^1 l_i^\tau(s)ds, \end{aligned} \tag{2.29}$$

as in RKM, which is called "collocation update". The weights b_i are the integrals over the Lagrange polynomials over the entire interval. Notice that if the nodes do include the end point, i.e. $\tau_M = 1$, we have $b_i = q_{Mi}$ in direct analogy to stiffly accurate RKM.

Equation 2.28 is a fully implicit Runge-Kutta method that is expensive to solve directly because Q is dense. We will first detail the original derivation of SDC as a method that solves an equation for the error in each iteration [47], and then present an alternative derivation that is more rooted in linear algebra and has lead to very effective schemes for stiff problems [161].

The error at iteration k is the difference between the exact solution of the collocation problem and the numerical solution that is obtained by interpolation from the solution values at iteration k

$$\delta^k(t) = u(\vec{\tau})\vec{l^\tau}(t/\Delta t) - \vec{u}^k\vec{l^\tau}(t/\Delta t). \tag{2.30}$$

Note that the collocation problem, in turn, has an error with respect to the true continuous solution of the initial value problem of

$$\delta_{\text{collocation}} = u(t) - u(\vec{\tau})\vec{l}^{\tau}(t/\Delta t), \tag{2.31}$$

but SDC iterates only towards the solution of the collocation problem. We now plug the error into Equation 2.23, to obtain

$$\delta^k(t) = u_0 + \int_0^t f\left(\vec{u}^k\vec{l}^{\tau}(s/\Delta t) + \delta^k(s)\right) ds - \vec{u}\vec{l}^{\tau}(t/\Delta t). \tag{2.32}$$

Next, we add 0, noting that the quadrature rule integrates the polynomial approximation exactly i.e. $\int_0^t F(\vec{u}^k)\vec{l}^{\tau}(s/\Delta t)ds = \Delta t(QF(\vec{u}^k))\vec{l}^{\tau}(t/\Delta t)$ to get

$$\begin{aligned}\delta^k(t) = &\int_0^t \left[f\left(\vec{u}^k\vec{l}^{\tau}(s/\Delta t) + \delta^k(s)\right) - F\left(\vec{u}^k\right)\vec{l}^{\tau}(s/\Delta t)\right] ds \\ &+ u_0 + \Delta t\left(QF\left(\vec{u}^k\right)\right)\vec{l}^{\tau}(t/\Delta t) - \vec{u}^k\vec{l}^{\tau}(t/\Delta t) \\ = &\int_0^t \left[f\left(\vec{u}^k\vec{l}^{\tau}\,(s/\Delta t) + \delta^k(s)\right) - F\left(\vec{u}^k\right)\vec{l}^{\tau}\,(s/\Delta t)\right] ds + \vec{r}^k\vec{l}^{\tau}\,(t/\Delta t)\,,\end{aligned} \tag{2.33}$$

with the residual $\vec{r}^k := \vec{u}_0 + \Delta t QF(\vec{u}^k) - \vec{u}^k$. We now obtain the SDC iteration by integrating this numerically with quadrature rule Q_Δ, which uses the same nodes τ_m as Q, and which integrates the error to low order from node to node, to get an approximation for the error which is used to update the solution

$$\vec{\delta}^k - \Delta t Q_\Delta\left(F\left(\vec{u}^k + \vec{\delta}^k\right) - F\left(\vec{u}^k\right)\right) = \vec{r}^k, \tag{2.34}$$

$$\vec{u}^{k+1} = \vec{u}^k + \vec{\delta}^k. \tag{2.35}$$

By expanding the residual and plugging in the refinement equation, we can eliminate the defect and simplify the SDC iteration to

$$\left(I_M - \Delta t Q_\Delta F\right)\left(\vec{u}^{k+1}\right) = \vec{u}_0 + \Delta t\left(Q - Q_\Delta\right)F\left(\vec{u}^k\right), \tag{2.36}$$

where I_M is the $M \times M$ identity matrix and Q_Δ is called preconditioner. Since the algorithm proceeds from one line of the system to the next, SDC iterations are often referred to as "sweeps."

Because the preconditioner integrates the error equation from node to node, it has to be lower triangular, which means each SDC iteration is solved with forward substitution. For instance, the preconditioner corresponding to implicit Euler from node to node reads

$$Q_\Delta^{\text{IE}} = \begin{pmatrix} \tau_2 - \tau_1 & 0 & 0 & \dots & 0 \\ \tau_2 - \tau_1 & \tau_3 - \tau_2 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \dots & 0 \\ \tau_2 - \tau_1 & \tau_3 - \tau_2 & \dots & \dots & \tau_M - \tau_{M-1} \end{pmatrix}. \tag{2.37}$$

Note that in SDC one may choose to perform the collocation update, even when the nodes include the end point, which can improve accuracy in some situations when the $\vec{u}^k$ are not the exact collocation solution [135].

The above derivation following the original SDC publication [47] is focused on numerical integration using quadrature rules, both in the collocation problem and in the preconditioner. SDC has since been reimagined as preconditioned Richardson iteration for the collocation problem by following a more algebraically focused derivation [161]. The standard Richardson iteration for the collocation problem reads

$$\begin{aligned} \vec{u}^{k+1} &= \vec{u}^k + \vec{r}^k \\ \vec{r}^k &:= \vec{u}_0 + \Delta t Q F(\vec{u}^k) - \vec{u}^k, \end{aligned} \tag{2.38}$$

which can be preconditioned with $(I_M - \Delta t Q_\Delta F)^{-1}$ and rearranged into Equation 2.36. The preconditioner is now essentially arbitrary and does not have to be consistent with any sort of integration rule.

Error bounds for SDC. In the original interpretation of SDC, the method approximates the current error by a first order method and subtracts it from the solution. This gives an order k accurate method after k iterations, until the order of the underlying collocation problem is reached. However, this simple explanation is no longer sufficient when preconditioners are desired that are not interpretable as quadrature rules for solving the error.

Various convergence analyses for SDC are available in the literature, e.g. [29, 73, 82, 151]. Here, we discuss error bounds for SDC with generic preconditioner via an amalgamation of [151] and [29]. This derivation merely recasts existing derivations into our notation and does not offer new insight. We start by subtracting the integrated IVP Equation 2.23 from the SDC iteration Equation 2.36. Note that throughout this paragraph, we write $\vec{l}^\tau_t = \vec{l}^\tau(t)$ in order to write the equations slightly more compactly.

$$\vec{u}^{k+1}\vec{l}^\tau_{\Delta t} - u(\Delta t) = \Delta t Q_\Delta \left[F\left(\vec{u}^{k+1}\right) - F\left(\vec{u}^k\right)\right] \vec{l}^\tau_{\Delta t} + \Delta t Q F\left(\vec{u}^k\right) \vec{l}^\tau_{\Delta t} - \int_0^{\Delta t} f(u(t))dt \tag{2.39}$$

$$\delta^{k+1} = \underbrace{\Delta t Q_\Delta \left[F\left(\vec{u}^{k+1}\right) - F\left(\vec{u}^k\right)\right] \vec{l}^\tau_{\Delta t}}_{I} + \underbrace{\int_0^{\Delta t} \left[F\left(\vec{u}^k\right) \vec{l}^\tau_t - f(u(t))\right] dt}_{II}. \tag{2.40}$$

We can pull the polynomial approximation into the integral in the second term because the quadrature rule Q integrates the polynomial approximation exactly.

We assume from now on that f is Lipshitz-continuous with Lipshitz constant L (see Equation 2.8). Since F is just a vector of f, it has the same Lipshitz constant L. We furthermore assume that $f \circ u$ is M times continuously differentiable with $C \geq \| \frac{d^M}{dt^M} f \circ u \|_\infty$.

We start with the second term II, where we proceed analogously to Lemma 2.1 from [29]. We add zero in the integral with the interpolation of the right hand side evaluated at the exact solution at the collocation nodes, which we write as $F(u(\vec{\tau}))\vec{l}^\tau_t$. Then, we use linearity of the integral, the triangle inequality and Lipshitz continuity

$$|II|_\infty = \left| \int_0^{\Delta t} \left[F\left(\vec{u}^k\right) - F(u(\vec{\tau}))\right] \vec{l}^\tau_t \, dt + \int_0^{\Delta t} \left[F(u(\vec{\tau}))\vec{l}^\tau_t - f(u(t))\right] dt \right|_\infty \tag{2.41}$$

$$\leq \left| \left[F\left(\vec{u}^k\right) - F(u(\vec{\tau})) \right] \int_0^{\Delta t} \vec{l}_t^{\tau} dt \right|_\infty + \underbrace{\left| \int_0^{\Delta t} \left[F(u(\vec{\tau}))\vec{l}_t^{\tau} - f(u(t)) \right] dt \right|_\infty}_{\text{error of collocation problem } e_{\text{coll}}} \tag{2.42}$$

$$\leq \Delta t L \left| \vec{u}^k - u(\vec{\tau}) \right|_\infty C_1 + \frac{C}{M!} \Delta t^{M+1} \tag{2.43}$$

$$\leq \Delta t L \left| \delta^k \right| C_1 + \frac{C}{M!} \Delta t^{M+1}. \tag{2.44}$$

The first term with constant $C_1 = M \left| \int_0^1 \vec{l}_t^{\tau} dt \right|_\infty$ represents the error in the integration due to the approximate solution and the second term represents the integral over the standard interpolation error. Note that using standard interpolation error as bound for the error of the collocation problem e_{coll} is often not a tight bound in practice. We will return to this later.

Next, we turn to the first term I, where we require only the Lipshitz continuity and the triangle inequality.

$$|I|_\infty \leq \Delta t L C_2 \left| \left(\vec{u}^{k+1} - \vec{u}^k \right) \right|_\infty = \Delta t L C_2 \left| \delta^k \right|_\infty,$$

with constant

$$C_2 = \sum_{m=1}^{M} \max_{1 \leq j \leq M} \left| \tilde{q}_{mj} \right|.$$

Putting the bounds on the two terms together, we get

$$\left| \delta^{k+1} \right|_\infty \leq \Delta t L (C_1 + C_2) \left| \delta^k \right|_\infty + \frac{C}{M!} \Delta t^{M+1}. \tag{2.45}$$

Note that this bound applies to arbitrary nodes, particularly also nodes that are not normalized. The implication we are interested in now, is that this is a valid bound anywhere within the interval with spectral quadrature rules. In the Runge-Kutta context, the ability to evaluate the solution to high accuracy anywhere within the interval is referred to as "dense output" [72, Sect. II.6]. SDC naturally supports dense output because the solution of the collocation problem is a polynomial approximation that can be evaluated with order M accuracy anywhere within the interval.

In practice, we are usually interested in the less general result of error bounds at the end of the interval with spectral quadrature rules. We have estimated the error of the collocation problem

$$e_{\text{coll}} = \left| \int_0^{\Delta t} \left[F(u(\vec{\tau}))\vec{l}_t^{\tau} - f(u(t)) \right] dt \right|_\infty \tag{2.46}$$

with standard interpolation error above, but for the special case of spectral quadrature rules, we get, at the end of the interval, a tighter bound

$$e_{\text{coll}} = C_3 \Delta t^{p+1}, \quad 2M - 2 \leq p \leq 2M, \tag{2.47}$$

where p is the order of the quadrature rule, provided $f \circ u$ is p times continuously differentiable. After plugging this in at Equation 2.42, we get

$$\|\delta^{k+1}\|_\infty \leq \Delta t L (C_1 + C_2) \|\delta^k\|_\infty + C_3 \Delta t^{p+1}, \tag{2.48}$$

We see that the error is multiplied by a constant times the step size between iterations, which corresponds to an increase in the order by one, if the step size is sufficiently small, i.e. $\Delta t L(C_1 + C_2) < 1$, up to the order of the collocation problem. This means the error bounds only prove the gain of one order per iteration asymptotically. However, in practice, the resulting order of accuracy is often observed numerically for larger Δt as well.

Preconditioners for SDC. After the requirement for Q_Δ to be interpretable as a quadrature rule was lifted, a popular way to generate good preconditioners emerged by optimizing the spectral radius of the SDC iteration matrix for the Dahlquist problem.

Plugging in the Dahlquist equation (Equation 2.19) into the SDC iteration (Equation 2.36), we get

$$(I_M - \Delta t Q_\Delta \lambda)\,\vec{u}^{k+1} = \vec{u}_0 + \Delta t\,(Q - Q_\Delta)\,\lambda \vec{u}^k, \tag{2.49}$$

which gives iteration matrix for the error $\vec{e}^k = \vec{u}^k - (u(\tau_1), \ldots, u(\tau_M))^T$

$$\vec{e}^{k+1} = K_{\lambda\Delta t}\vec{e}^k, \quad K_{\lambda\Delta t} = \lambda\Delta t(I_M - \lambda\Delta t Q_\Delta)^{-1}(Q - Q_\Delta), \tag{2.50}$$

see [25]. In the stiff limit $\|\lambda\Delta t\| \to \infty$, the iteration matrix reads

$$K_{\|\lambda\Delta t\|\to\infty} = I - Q_\Delta^{-1} Q. \tag{2.51}$$

If one chooses $Q_\Delta = U^T$, with $Q^T = LU$, where L has all diagonal entries one, $K_{\|\lambda\Delta t\|\to\infty}$ is nilpotent, which makes for a very effective preconditioner for stiff problems [161]. This choice of preconditioner is usually called LU trick in the literature.

When the preconditioner is diagonal, the implicit solves can be performed concurrently. Early diagonal preconditioners were derived by numerically minimising the spectral radius of the stiff and non-stiff limits of $K_{\lambda\Delta t}$ [79, 141]. The topic has been revisited later to obtain very effective diagonal preconditioners for both stiff and non-stiff problems called MIN-SR-S and MIN-SR-NS [25]. We will return to this topic when discussing parallel-in-time integration.

All preconditioners discussed so far lead to an increase in the order of accuracy by one with each iteration up to the order of the underlying collocation problem, provided they converge at all. Higher order preconditioners such as RKM can sometimes increase the order of accuracy by more than one with each iteration [29, 35]. However, lack of smoothness in the error can limit gains to one order per iteration regardless of the order of the preconditioner, particularly when non-equidistant nodes are used [34]. Note that an increase in the order of accuracy of exactly one per sweep is not always observed with LU or diagonal preconditioner [25, 93].

Inexact SDC iterations. Performing the implicit solves inside the SDC iteration to full accuracy often comes at no benefit to overall accuracy because the preconditioner itself is only approximate. Reducing tolerances or strict limits on the number of allowed iterations for the implicit solver can be used to avoid over-solving and to improve overall computational efficiency of SDC [147]. Optimal tolerances can be derived, but require realistic work and error models [162]. Still, efficiency can be gained even with sub-optimal tolerances when adjusting the tolerance of an inner solver based on the outer residual or when allowing only few iterations of an inner solver. Because SDC provides good initial guesses for the nonlinear solver, this can lead to very efficient schemes [147].

IMEX-splitting. IMEX splitting is very easy to realize in SDC [113, 135]. Recall that IMEX Euler does not have any coupling conditions, in contrast to higher order IMEX RKM [126], but can easily be employed in SDC to obtain high-order solutions. That being said, IMEX SDC is not limited to IMEX Euler preconditioners. The IMEX-SDC iteration for generic preconditioners reads

$$\begin{aligned}(1-\Delta t\tilde{q}^I_{m+1,m+1}f^I)(u^{k+1}_{m+1}) = u_0 &+ \Delta t\sum_{j=1}^{m}\left(\tilde{q}^I_{m+1,j}f^I+\tilde{q}^E_{m+1,j}f^E\right)(u^{k+1}_j)\\ &- \Delta t\sum_{j=1}^{m+1}\left(\tilde{q}^I_{m+1,j}f^I+\tilde{q}^E_{m+1,j}f^E\right)(u^{k}_j)\\ &+ \Delta t\sum_{j=1}^{M} q_{m+1,j}(f^I+f^E)(u^k_j),\end{aligned} \tag{2.52}$$

with superscripts I and E referring to the implicit and explicit part and $\tilde{q}$ being the entries of the preconditioners. Notice that the splitting is implemented simply by splitting $f = f^I + f^E$ and adding the terms with different preconditioners.

The scheme typically has the same order of accuracy as the non-split version, although order reduction may occur [64, 113]. IMEX-SDC has been shown to outperform DIRK-based IMEX Runge-Kutta methods for incompressible flow simulations in wall-time measurements [64].

2.3.5 Parallel-in-time extensions of SDC

Parallel-in-time (PinT) methods are generally grouped in two categories [23]. "Parallel-across-the-method" algorithms perform the computations required for integration of a single time step in parallel, whereas "parallel-across-the-steps" algorithms integrate multiple time steps concurrently. Here, we discuss the parallel-across-the-method extension diagonal SDC [79] and the parallel-across-the-steps extensions Block Gauß-Seidel SDC [66] and PFASST [50]. See the review by Gander [56] for a more extensive overview of the PinT landscape including non-SDC based methods.

Diagonal SDC. The idea behind diagonal SDC is rather simple: If the SDC preconditioner can be chosen arbitrarily, it can be chosen diagonal. This eliminates the coupling between the collocation nodes and allows to update them in parallel. This entails both implicit solves and right hand side evaluations, which typically form the most expensive part of the integration process.

The idea was first proposed by van der Houwen and Sommeijer [79] under the name of "parallel iterated Runge-Kutta methods" and independently rediscovered by Speck [141]. However, the efficiency crucially depends on the performance of the preconditioner. The diagonal preconditioners introduced in these publications are based on numerically minimizing the spectral radius of the SDC iteration matrix for the Dahlquist equation in the stiff or non-stiff limits (see Equation 2.50). Performance of these preconditioners lacked severely behind non-diagonal preconditioners such as LU.

Good diagonal preconditioners were later derived by Čaklović et al. [25]. For non-stiff problems, they propose "MIN-SR-NS", for which the SDC iteration matrix in the non-stiff

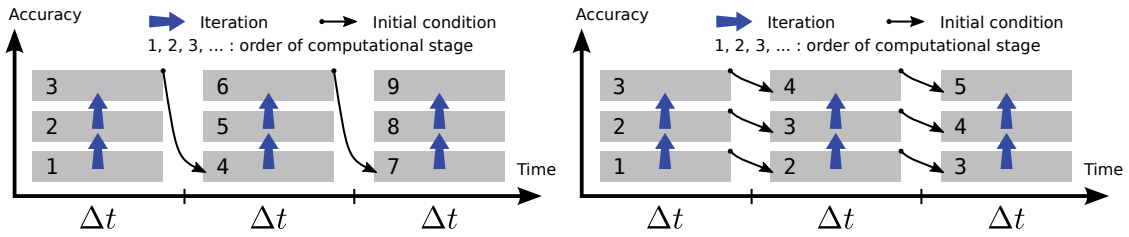


Figure 2.7: Left: Sequential SDC. Right: Block Gauß-Seidel SDC. Sequential time stepping uses the converged solution of the last step as initial conditions, whereas the parallel version receives refined initial conditions between iterations. Figure reproduced from [11, Fig. 1] under CC BY 4.0 licence.

limit is nilpotent which corresponds to analytically minimzing the spectral radius to zero. The coefficients of the preconditioner are

$$\tilde{q}^{NS}_{mm} = \frac{\tau_m}{m}, \tag{2.53}$$

where τ_m is the m^{th} collocation node. They also design a procedure for obtaining numerically optimized preconditioners for stiff problems. The resulting preconditioners are called "MIN-SR-S" and are often competitive for stiff problems to the established LU preconditioner. While this list of diagonal preconditioners is by no means exhaustive, these are the ones we found to work best in the numerical experiments we show in later sections.

Pipeline-based parallelism: Block Gauß-Seidel SDC. A relatively straightforward pipeline-based parallel-in-time variant of SDC can be constructed by solving multiple time steps simultaneously in block Gauß-Seidel fashion [66]: Instead of waiting for the previous step to converge to full accuracy before starting a new step, we begin solving a new step as soon as a single iteration has been performed on the previous step in the block and keep refining the initial conditions with the iterates from the previous step between iterations. See Figure 2.7 for an illustration of the procedure, which we refer to as BGSSDC for the remainder of the document.

Now, increasing the number of steps that are solved in parallel in this fashion changes the solution in any but the first step, as inexactness is introduced in the initial conditions. While it has been demonstrated that the convergence order is maintained when increasing the number of steps [57], obtaining the same accuracy as serial SDC often requires smaller step size. How much more work is required because of this effect is dependent on the problem and on the preconditioner, but perfect speedup is not achieved for most interesting problems.

Note that inexactness and non-optimal parallel efficiency is typical of parallel-across-the-steps algorithms. Often, they increase the workload substantially, but may reduce the time-to-solution by simultaneously increasing concurrency. For PDEs, the spatial discretization can often be parallelized with higher parallel efficiency. Parallel-across-the-steps methods are, therefore, typically added on top of spatial parallelism to extend the overall scaling capabilities, see e.g. [117, 136, 146].

PFASST The Parallel Full Approximation Scheme in Space and Time (PFASST) [50] is a parallel-across-the-steps PinT algorithm that builds on the idea of BGSSDC, but expands it significantly. Similarly to BGSSDC, multiple time steps are solved at the same time and BGSSDC is a building block itself. Another building block of PFASST is Block-Jacobi SDC (BJSDC), which is similar to BGSSDC, but all steps are started at the same time, instead of waiting for the first iteration of the previous step to use as initial conditions. While this can be significantly less accurate for a given number of iterations, the iterations are parallelizable.

Another ingredient of PFASST is multilevel SDC [145], which works on a hierarchy of levels. The levels may differ, for example, in the grid resolution, the spatial discretization or accuracy of linear solvers. The levels are coupled using an FAS correction term as in nonlinear multigrid methods [19]. In PFASST, BGSSDC is then employed to propagate information forward on the coarsest level, whereas finer levels are iterated concurrently with BJSDC.

We do not further analyse PFASST in this thesis. However, we note that BGSSDC can be viewed as single-level PFASST. Therefore, we expect adaptive time-step selection methods for BGSSDC, which we will discuss in chapter 3, to be applicable in PFASST as well.

2.4 Adaptive time-stepping

Since the step size Δt is one of the fundamental parameters controlling the accuracy of the simulation, it should be chosen with care. It has to be chosen small enough for the time-stepping to be stable, which depends on the problem and the integrator, but beyond that it is entirely up to the user to select a step size that satisfies the accuracy requirements. More specifically, the step size needs to be set relative to the time-scale the problem evolves on, in order to resolve all of the dynamics. However, for many problems the time-scale is not constant during the temporal domain that is integrated over. Since the time-scale of the problem is part of the unknown solution, it is best to adapt the step size to the time-scale during runtime. We provide an illustration of a simple problem with changing time-scale in Figure 2.8.

We will first discuss embedded RKMs, which are a class of RKM that include mechanisms for adaptive step size selection. Then we will discuss existing work in adaptive step size selection in SDC. We will later propose generic step size selection algorithms in SDC that are based on the same idea as embedded RKMs in chapter 3.

2.4.1 Embedded Runge-Kutta methods

Embedded RKM provide a generic approach to step size adaptivity that is both simple and mature. We briefly review the basic concepts here and refer to the book by Hairer and Wanner [72, Chapter II.4] for a more comprehensive discussion.

The step size selection works by estimating the local error and updating the step size to match the error estimate to the target accuracy. The idea is to use two RKM of different orders p and $q > p$. Assuming the order is observed numerically, the higher order method is sufficiently more accurate to act as an exact solution relative to the lower order method.

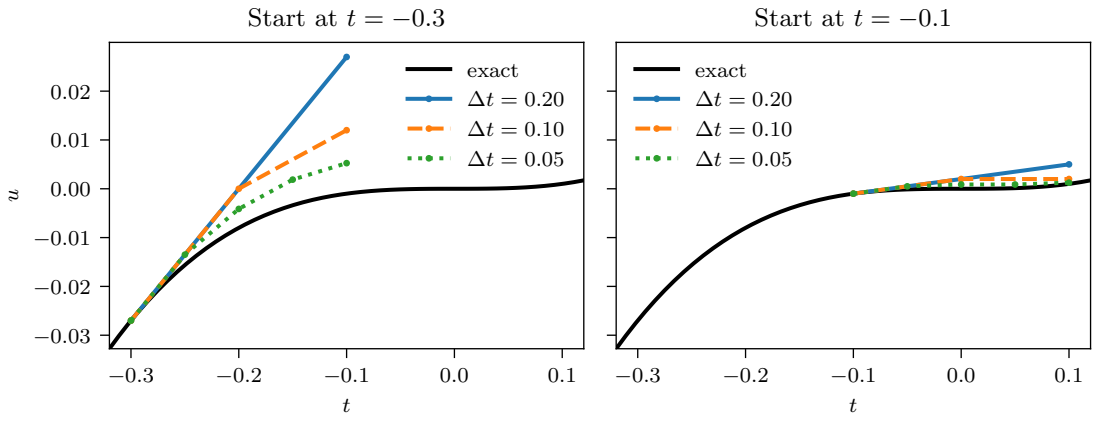


Figure 2.8: Example for a problem requiring adaptive step size selection. The ODE to be solved here is $u_t = 3t^2$, with exact solution $u^* = t^3$. The two panels show numerical solutions obtained using explicit Euler from different starting points and with different step sizes. In the left panel, we start when the evolution in time is faster than in the right panel. Consequently, much smaller step sizes are required for adequate resolution in the left panel compared to the right one. When starting at $t = -0.3$, a step size of $\Delta t = 0.05$ gives comparable error after integration for 0.2 time units as $\Delta t = 0.2$ when starting at $t = -0.1$.

An error estimate ϵ is then simply obtained by the difference between the two solutions

$$\begin{aligned} \epsilon &= \|u^{(p)} - u^{(q)}\|_\infty \\ &= \|(u^{(p)} - u^*) - (u^{(q)} - u^*)\|_\infty \\ &= \|\delta^{(p)} - \delta^{(q)}\|_\infty = \|\delta^{(p)}\|_\infty + \mathcal{O}(\Delta t^{q+1}), \end{aligned} \tag{2.54}$$

where $u^{(p)}$, $u^{(q)}$ are the solutions at the end of the time interval, obtained by integration schemes of order p and q with $q > p$. u^* marks the exact solution, δ denotes the local error with analogous meaning of the superscript and ϵ is the estimate of the local error. Once ϵ is known, an optimal step size

$$\Delta t_{\text{opt}} = \beta \Delta t \left(\frac{\epsilon_{\text{TOL}}}{\epsilon} \right)^{1/(p+1)} \tag{2.55}$$

can be inferred from the order of the method such that $\epsilon \approx \epsilon_{\text{TOL}}$, which is the user-defined tolerance, if the step were to be repeated with Δt_{opt} and β is a safety factor that is usually set to $\beta = 0.9$. Note the factor of $p + 1$ instead of p in the exponent because the order p refers to the global order of accuracy of the method, but the estimate is for the local error. The order of the local error is generally one higher because the local errors accumulate in the global error.

Of course, repeating the step with the optimal step size increases computational cost and should be avoided, if possible. Therefore, the step is repeated only if $\epsilon > \epsilon_{\text{TOL}}$. On the other hand, Δt_{opt} may not actually be suitable to obtain the target accuracy in the next step. The safety factor β is used to accommodate slight shortening in the problem time-scale without necessitating a restart.

Computing two solutions of different order may seem expensive, but it can be done without adding much computational cost. Recall that RKM work by computing stages,

which are low-order right hand side evaluations at intermediate times, and which are then assembled to give a high order solution at the end of the interval. In embedded RKM, the same stages are assembled twice with different weights to give the two solutions. Therefore, no additional intermediate solutions or right hand side evaluations are needed. Embedded RKM are often written as Butcher tableaux with two lines for the weights

$$\begin{array}{c|c} c & A \\ \hline & b \\ & \hat{b} \end{array}$$

where $\hat{b}$ are the weights of the lower order method.

Note that the error estimate entered into the step size update equation is for the lower order method. Because the higher order method must be at least as accurate in order for the error estimate to work, it can be used to advance in time regardless. This practice is called "local extrapolation" [71].

2.4.2 Adaptivity in SDC

There are numerous ways to implement adaptivity in SDC. First of all, if SDC is treated as an iterative scheme by using stopping criterion other than fixed number of iterations, it is inherently adaptive. However, using the iteration number for adaptivity comes with two important drawbacks. First, the iteration number is an integer, which prohibits fine tuning. Second, the maximum accuracy that can be reached here is that of the converged collocation problem, which depends on the quadrature rule and step size.

Adjusting the step size adaptively is therefore advantageous, although setting both adaptively at the same time can also yield very efficient schemes. Step size adaptivity in SDC has been proposed in a number of ways in the past, but the proposed schemes have been either tailored to specific problems, or employed sub-optimal step size update equations. We briefly review existing work here and introduce generic step size adaptivity with error estimates tailored to SDC and an efficient step size update equation in chapter 3.

In the original SDC publication [47], the authors suggest to monitor the increment between SDC iterations and double or halve the step size if the increment or residual is unsatisfactorily large. While this gives information about whether a solution to the collocation problem was obtained, it is not clear how accurate the collocation problem is itself and the simple step size update is not likely efficient in most practical cases.

The idea was picked up by van der Houwen and Sommeijer [78, 80], although not under the name of SDC, but parallel iterated Runge-Kutta (PIRK). They observe that the order of accuracy increases by one with each iteration up to the order of the underlying fully implicit Runge-Kutta method, which allows to use successive iterates as a pair of solutions of different order. Concepts from embedded RKM were therefore readily transferred to create embedded PIRK methods. We will build on this idea later on in the SDC context, which allows to leverage many of the concepts that have been developed within SDC since. The link between PIRK methods and SDC has only recently been established in the literature [25], which is why the work by van der Houwen and Sommeijer has not received a lot of attention in the SDC community thus far.

SDC with adaptive time-stepping using state-of-the-art step size update equations were successfully performed for vesicle suspension simulations [130]. The vesicles are incompress-

ible and inextensible and the authors use this to estimate the local truncation error based on the violation of these constraints. A similar approach with error estimate based on the energy derivative was shown to work well for the modified phase field crystal equation [67].

SDC has been used to obtain high order solutions for the binary fluid surfactant system, where the solution with SDC was compared to a first order method used as predictor in SDC [68]. This is very similar to the idea of embedded methods as it compared two solutions to the same problems of different order.

SDC also allows more novel versions of adaptivity. For instance, a scheme has been proposed where the increment in the SDC iteration is monitored locally in space, and convergence is declared locally as well [31]. After part of the domain has converged, fewer degrees of freedom remain to be solved in subsequent iterations.

2.4.3 Step size selection controllers

The step size selection controller sits between estimating Δt_{opt} and updating Δt. The most simple controller, which simply updates $\Delta t = \Delta t_{\text{opt}}$ aims to minimize the number of time steps needed to solve the problem to the desired accuracy. However, this is not necessarily the same as minimizing computational cost. For instance, the cost of implicit solves may increase with step size, rendering the largest possible step sizes suboptimal [49], or some parts may be reused from previous steps if the step size stays the same, in which case solutions such as PID controllers [140] can be useful.

2.5 Spectral methods for spatial discretization of partial differential equations

We have introduced SDC and RKM as methods to integrate ordinary differential equations (ODEs). The methods can be just as easily applied to systems of ODEs and a common way for solving time-dependent PDEs is the method of lines, where the PDE is discretized into a system of ODEs. We will now introduce a family of methods, which are called spectral methods, for discretizing space derivatives that follow this philosophy. We largely follow the textbooks [18] and [154].

The design principle behind spectral methods is to expand the solution in a basis of functions that are easy to deal with. When discretizing PDEs, this mainly means simple relationships between the functions and their derivatives. As an example, consider the basis of polynomials $P_n(x) = x^n,\ n \in \mathbb{N}$. The derivative of each element is $\partial_x P_n(x) = nP_{n-1}(x)$ for $n > 0$ and $\partial_x P_0(x) = 0$. When expanding a function $u(x) \approx \sum_{i=0}^{N} \hat{u}_i P_i(x)$ in the basis of polynomials up to order N, the derivative operator can then be expressed as a matrix

$$\partial_x u(x) \approx \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & N \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{u}_0 \\ \hat{u}_1 \\ \vdots \\ \hat{u}_{N-1} \\ \hat{u}_N \end{pmatrix}, \tag{2.56}$$

which allows to easily compute and invert derivatives numerically using linear algebra libraries. Note that differentiation matrices are not invertible because the integration

constant is missing. We will discuss adding the boundary conditions which define the integration constants to arrive at invertible matrices in section 2.5.10.

Using the basis of polynomials for a global expansion is instructive, but not of much use in practice. First, the elements are unbounded, which makes it very hard to decide when to truncate the expansion in practice. The second issue is obtaining the expansion. A common terminology for this is "switching from physical to spectral space." The physical representation means the values on the grid, whereas the spectral representation means the coefficients of the basis functions. The two representations are equivalent up to the truncation error in the expansion. It is easy to go back to physical space from spectral space by simply evaluating the polynomials at any x, but the other way is less straightforward to perform numerically. However, other bases exist that have similarly convenient differentiation matrices and that provide a mechanism for efficiently computing the spectral representation.

2.5.1 The Fourier base for spectral methods

The basis elements in Fourier series in a domain of length L are complex exponentials

$$W_n(x) = \exp(-2\pi i x n/L), \tag{2.57}$$

which have derivatives

$$\partial_x W_n(x) = \frac{-2\pi i n}{L} W_n. \tag{2.58}$$

The derivative matrix is diagonal, which is a very attractive property as it allows to evaluate and invert the derivative for all elements in parallel. Note that all basis elements are periodic and any linear combination will also be periodic. The Fourier base can therefore only be used to represent periodic functions.

In order to estimate the error with respect to the approximated function, one may look at the truncation error of the series expansion

$$e_{\text{trunc}} = \left| \sum_{n=0}^{\infty} \hat{u}_n \phi_n(x) - \sum_{n=0}^{N-1} \hat{u}_n \phi_n(x) \right| = \left| \sum_{n=N}^{\infty} \hat{u}_n \phi_n(x) \right|, \tag{2.59}$$

where ϕ_n are the basis functions. Since the Fourier basis functions are bounded by one, we can use the triangle inequality to obtain

$$e_{\text{trunc}} \leq \sum_{n=N}^{\infty} |\hat{u}_n| \, |\phi_n(x)| \leq \sum_{n=N}^{\infty} |\hat{u}_n| \,. \tag{2.60}$$

The coefficients in the Fourier transform decay quickly if the transformed function is smooth [154, Theorem 1]. In particular, for infinitely continuously differentiable functions, the coefficients decay asymptotically as

$$\hat{u}_n = \mathcal{O}(|n|^{-m}), \quad |n| \to \infty, \tag{2.61}$$

for any $m \geq 0$. The series therefore converges exponentially fast.

The above exponential rate of convergence is often called "spectral convergence," but it should be noted that this is not the defining property of spectral methods. In fact,

Fourier spectral methods converge only algebraically for smooth, but not infinitely smooth functions. For non-smooth functions, Darboux's principle relates the rate of convergence to singularities [18, Table 2.2].

Since the degree of smoothness of the function, or the function itself, is not generally known and the convergence rate is only known asymptotically, it can be difficult to estimate the truncation error a priori. Nevertheless, if one can expect the function to be somewhat smooth and periodic, Fourier spectral methods are often a good choice, delivering the same accuracy with fewer degrees of freedom than other discretizations.

The expansion of a periodic signal in a Fourier series can be obtained via Fourier transform, which involves an integral over the entire domain. On grids with finite spacing, we use the discrete Fourier transform (DFT), a weighted sum over all data. The DFT can efficiently be computed via an algorithm aptly named Fast Fourier Transform (FFT), which traces its history to Gauß, has been rediscovered countless times since [76] and was finally popularized by Cooley and Tukey [39].

2.5.2 Fast Fourier Transform

The DFT of a vector of data $\vec{u} = (u_0, u_1, ..., u_{N-1})$ can be written as

$$\hat{u}_m = \sum_{k=0}^{N-1} u_k e^{-2\pi i m k/N}, \quad m = 0, 1, ..., N-1, \tag{2.62}$$

where $i = \sqrt{-1}$. Note that in this section we require many indices and therefore use different notation than commonly used in the literature and outside this section, where usually n is the space index, which we call k here and k is the wave number, which we call m here. Bluntly executing these weighted sums of N u_k for each of the N $\hat{u}_m$ requires $\mathcal{O}(N^2)$ operations. The FFT algorithm instead achieves the result with $\mathcal{O}(N \log N)$ operations by a divide-and-conquer approach. The following is a brief summary from the analysis by Cooley and Tukey [39].

The domain is split into n_1 smaller domains of size n_2, such that $n_1 n_2 = N$. Indices are then split accordingly as

$$\begin{aligned} m &= m_1 n_1 + m_0, & m_0 &= 0, 1, ..., n_1 - 1, & m_1 &= 0, 1, ..., n_2 - 1, \\ k &= k_1 n_2 + k_0, & k_0 &= 0, 1, ..., n_2 - 1, & k_1 &= 0, 1, ..., n_1 - 1, \end{aligned}$$

allowing to rewrite Equation 2.62 as

$$\hat{u}_{m_1,m_0} = \sum_{k_0=0}^{n_2-1} \sum_{k_1=0}^{n_1-1} u_{k_1,k_0} e^{-2\pi i m k_1 n_2/N} e^{-2\pi i m k_0/N}. \tag{2.63}$$

Recall now that $e^{2\pi i x}$ is periodic, specifically $e^{2\pi i x} = 1$ for $x \in \mathbb{Z}$, which means

$$e^{-2\pi i m k_1 n_2/N} = \underbrace{e^{-2\pi i m_1 k_1 \overbrace{n_1 n_2/N}^{1}}}_{1} e^{-2\pi i m_0 k_1 n_2/N} = e^{-2\pi i m_0 k_1 n_2/N},$$

which can be used to simplify to

$$\hat{u}_{m_1,m_0} = \sum_{k_0=0}^{n_2-1} \left(\sum_{k_1=0}^{n_1-1} u_{k_1,k_0} e^{-2\pi i m_0 k_1 n_2/N} \right) e^{-2\pi i m k_0/N}, \tag{2.64}$$

where the inner sum depends only on m_0 and k_0. Remember that there are n_1 choices of m_0 and n_2 choices of k_0, for a combined total of N. As each of the N inner sums goes up to n_1, we need Nn_1 operations. Then, the outer sum is evaluated, similarly requiring Nn_2 operations for a total of $N(n_1 + n_2)$ operations.

Of course, this procedure can be applied successively, splitting the domain into $N = \prod_{j=1}^{J} n_j$ to compute the DFT in $N \sum_{j=1}^{J} n_j$ operations. Choosing all $n_j = n$ equal, the cost is $nN \log_n N$, as claimed. n is often called radix in this context, meaning a radix-2 FFT algorithm will accept N equals to some power of 2. In general, the algorithm requires N to be a somewhat composite number, but Bluestein has developed a method to extend this to arbitrary amounts of data [17].

2.5.3 N-dimensional discretizations

The matrices for one-dimensional discretizations are easy to adapt to more dimensions by means of tensor (Kronecker) product. For instance, a two-dimensional derivative matrix along x, $D_x^{[2]}$ is constructed from the one-dimensional one $D_x^{[1]}$ by taking the tensor product with the one-dimensional identity matrix $I_y^{[1]}$ in y-direction

$$D_x^{[2]} = D_x^{[1]} \otimes I_y^{[1]}. \tag{2.65}$$

Similarly, one would construct a three dimensional derivative matrix along y by using two Kronecker products

$$D_y^{[3]} = I_x^{[1]} \otimes D_y^{[1]} \otimes I_z^{[1]}. \tag{2.66}$$

Note that the bases in each direction need not be the same. For instance, one can use Fourier base in all directions where the boundary conditions are periodic and other bases in other directions. We will give an example of such a discretization in section 2.6.4.

2.5.4 Parallelizing Fourier-spectral methods

As mentioned in section 2.1, we need to distribute the computational work in order to cater to modern HPC systems. As the differentiation matrix is diagonal in Fourier spectral methods, we can evaluate and invert derivatives for each mode independently. We split the global differentiation matrix into local ones as

$$\begin{aligned} D_{\text{global}} &= -\frac{2\pi i}{L} \begin{pmatrix} 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \dots & N \end{pmatrix}, \\ D^i_{\text{local}} &= -\frac{2\pi i}{L} \begin{pmatrix} iN_{\text{local}} & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & (i+1)N_{\text{local}} \end{pmatrix}, \quad i = 0, ..., N_{\text{tasks}} - 1, \end{aligned} \tag{2.67}$$

where N_{local} is the number of degrees of freedom per task and N_{tasks} is the number of parallel tasks, such that $N = N_{\text{local}} N_{\text{tasks}}$.

The computationally expensive part in Fourier spectral methods are typically the transforms. When using $d > 1$ dimensions, these can be distributed as well, which is another key

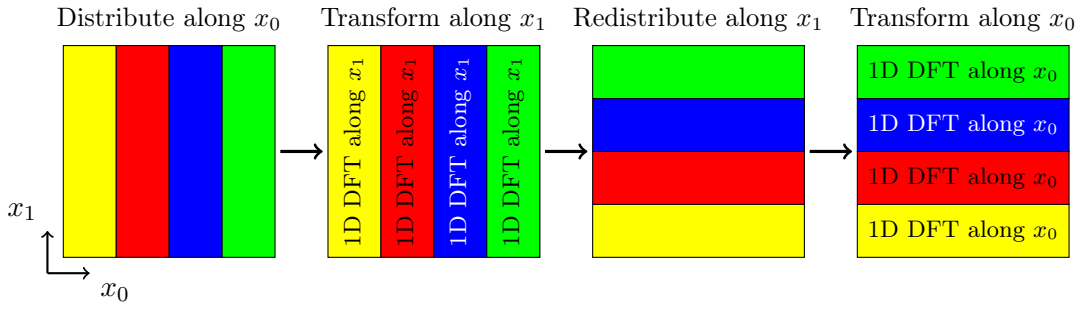


Figure 2.9: Illustration of distributed FFT via a 2-dimensional example with four processors. Each processor has access to the data of one color only. One-dimensional transforms can be performed in parallel after the data is aligned in the respective direction. In between transforms the data may need to be redistributed, requiring all-to-all communication.

reason for the popularity of FFTs. The d-dimensional FFT can be expressed as successive one-dimensional transforms

$$\hat{u}_{j_1,\ldots,j_d} = \sum_{k_1=0}^{N_1-1} \cdots \sum_{k_d=0}^{N_d-1} u_{k_1,\ldots,k_d} e^{-2\pi i j_1 k_1/N_1} \ldots e^{-2\pi i j_d k_d/N_d}, \tag{2.68}$$

where the one-dimensional transforms can be performed concurrently. In order to execute the sums in the transforms in parallel, the data to be summed must be available to a single task. In practice, this means the data is distributed along one or more axes, all transforms for non-distributed axes are performed, and then the data is redistributed along axes that are already transformed. This is iterated until the data is transformed along all axes. The procedure is illustrated with a two dimensional example in Figure 2.9.

The redistributions require all-to-all communication, which is very costly and may increase run-time more than it is reduced by parallelism. Therefore, it usually does not make sense to choose a distribution that fully exploits the available amount of concurrency. Instead, single tasks often perform multiple 1D FFTs, where shared memory parallelization may be used for further parallelism. Note, however, that the redistribution steps are the only additional steps compared to serial computation of the entire Fourier spectral method.

The amount of concurrency is dictated by the size and distribution of the data. Common distribution styles in three dimensions are “slab” decomposition, where one axis is distributed or “pencil” decomposition where two axes are distributed [9, Fig. 2]. Slab decomposition should be used until the number of tasks exceeds the size of the smallest axis, at which point one has to switch to pencils, which offers additional concurrency at the cost of one more redistribution.

2.5.5 Real valued data.

Consider the following definition of the inverse DFT

$$u_n = \sum_{k=-\frac{N}{2}}^{k=\frac{N}{2}-1} \hat{u}_k e^{2\pi i n k/N}, \quad n = 0, 1, \ldots, N-1, \tag{2.69}$$

where we switched to the more common notation with wave number $k = -N/2, -N/2 + 1, \ldots, N/2-1$ and space index n. If we are dealing with real valued data in physical space, that is $u_n \in \mathbb{R}$, the complex part of the sum must vanish. Looking at a single summand in detail and decomposing into real and complex part

$$\begin{aligned} 2\hat{u}_k e^{2\pi i n k/N} &= (a_k + ib_k)(\cos(2\pi nk/N) + i\sin(2\pi nk/N)) \\ &= a_k \cos(2\pi nk/N) - b_k \sin(2\pi nk/N) + i\left(b_k \cos(2\pi nk/N) a_k \sin(2\pi nk/N)\right), \end{aligned}$$

we can infer that we require $a_{-k} = a_k$ and $b_{-k} = -b_k$, that is $\hat{u}_{-k}$ must be the complex conjugate of $\hat{u}_k$ due to the symmetry properties of the trigonometric functions [95].

Many FFT libraries utilize this property and return only half as many points (plus $\hat{u}_0$) in spectral space as in physical space. Note that the information content is the same as in physical space because $\hat{u}_k \in \mathbb{C}$. This is simply a way of eliminating redundant information. We stress this here, because this has an impact on distributed FFTs. The length of the axis that is distributed is halved for real data, which reduces concurrency in many FFT libraries including `mpi4py-fft`, which we discuss later in section 2.8.3.

Note that for N even, the Fourier mode with wave number $-N/2$, the so called Nyquist mode, does not have a positive counter part. When the data is real, the accompanying coefficient must vanish $\hat{u}_{-N/2} = 0$. Therefore, when dealing with real data and N even, the result is the same whether one uses N or $N-1$ degrees of freedom. In practice, it is nevertheless common to use N a power of two in order to use radix-2 FFTs.

2.5.6 The Chebychev base for spectral methods

As mentioned above, Fourier spectral methods are restricted to periodic boundary conditions. For non-periodic boundary conditions, other bases, such as Chebychev polynomials, converge faster. We now introduce this basis and discuss how to enforce boundary conditions in section 2.5.10.

Chebychev polynomials have the useful and defining property that they relate to cosine series

$$T_n(\cos\theta) = \cos(n\theta), \tag{2.70}$$

with T_n the nth Chebychev polynomial. The expansion of a function in Chebychev polynomials can therefore be expressed as a cosine expansion, when using variable transformation $x = \cos\theta$

$$u(x) = \sum_{n=0}^{\infty} \hat{u}_n T_n(x) \overset{x=\cos\theta}{=} \sum_{n=0}^{\infty} \hat{u}_n \cos(n\theta). \tag{2.71}$$

As the cosine transform is closely related to the Fourier transform, the expansion in Chebychev polynomials can be computed with similar ease and speed as Fourier based methods via the discrete cosine transform (DCT).

Because the Chebychev method is essentially a Fourier method with a change of variable, it has the same attractive convergence properties as the Fourier spectral method discussed in section 2.5.1. In particular, the basis functions are bounded by one and the expansion converges exponentially fast to infinitely smooth functions.

To understand why the Chebychev base is better suited to problems with non-periodic boundary conditions than Fourier base, consider the example $u_x = -1$ and $u(-1) = 1$ with

exact solution $u^*(x) = -x$. When solving this problem with Fourier base on the interval $[-1, 1)$, one gets a periodic "sawtooth" solution. While this function is smooth within the interval, it is discontinuous at the boundary and slow convergence is to be expected. The Chebychev base does not have this issue due to the change of variable as $u^*(\cos\theta) = -\cos\theta$ is smooth and continuous at the boundary. In this example, the Fourier series coefficients decrease slowly with $\mathcal{O}(1/n)$ [18, example one], while the Chebychev series is exact after including T_1.

The desire to use DCT defines the grid via the transformation $x_j = \cos(\theta_j)$, with $\theta_j = \frac{2\pi j}{N}$, $j = 0, \ldots, N-1$ evenly spaced. The result is a grid from -1 to 1 with points clustered at the boundaries. It is possible to use different grids by transforming to this grid before the DCT.

The derivative in the Chebychev base is dense

$$\partial_x T_n(x) = \sum_{i=0}^{n-1} \frac{2n((n-i) \mod 2)}{1+\delta_{i0}} T_i(x), \tag{2.72}$$

which makes the algebraic operations, particularly inverting derivatives, expensive. For this reason, the Chebychev base is rarely used by itself, but usually combined with other bases. For instance, the first derivative of the Chebychev polynomials of the first kind (T) is sparse in the Chebychev polynomials of the second kind (U) with the relation

$$\partial_x T_n(x) = nU_{n-1}(x). \tag{2.73}$$

Conversion from T to U is also sparse with $2T_n(x) = U_n(x) - U_{n-2}(x)$, but its inverse is dense.

Recovering the derivative in the T base requires multiplication by the dense inverse conversion operator. However, the advantage is in inverting the derivative, where the conversion operator can be used as a left preconditioner, rendering the system sparse.

2.5.7 Discrete cosine transform

Recalling Euler's equation $e^{i\phi} = (\cos(\phi) + i\sin(\phi))/2$, it is easy to see that DCT and FFT are closely related. In fact, the DCT can be computed using FFTs. We now introduce a particular example of such a method from Makhoul [109] that computes the DCT of a real-valued N-point sequence using a single N-point FFT, and which we will use later in section 6.4.

The DCT of a vector of data $\vec{u} = (u_0, u_1, \ldots, u_{N-1})$ can be written as

$$\hat{u}_k = 2\sum_{n=0}^{N-1} u_n \cos\left(\frac{\pi(2n+1)k}{2N}\right), \quad m = 0, 1, \ldots, N-1. \tag{2.74}$$

The general idea behind computing this using FFT, is to reorder the values on the grid to be periodic, rotate the data in the complex plane and then perform the FFT.

The reordered version v_n of data $u_n, n = 0, \ldots, N$ is defined by

$$v_n = \begin{cases} u_{2n}, & 0 \le n \le (N-1)/2, \\ u_{2N-2n-1}, & (N+1)/2 \le n \le N-1. \end{cases} \tag{2.75}$$

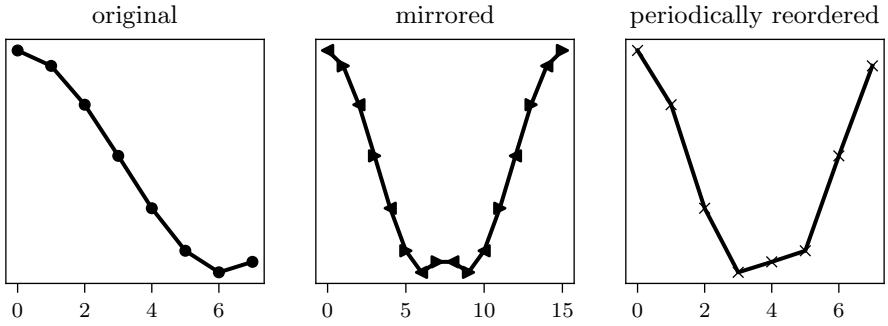


Figure 2.10: Reordering of data for computing the DCT via FFT. The left panel shows the original non-periodic data. The middle panel shows the same data, but mirrored, which makes it periodic. The right panel shows every other data point from the mirrored sequence. This is a periodic sequence of the same length as the original data set. In the middle panel, we use two different marker styles, where both the left- and the right-looking triangles contain the entire data.

This is a periodic reordered version of the same length as the original dataset, as visualized in Figure 2.10. We first write the FFT of the mirrored data set w (middle panel in Figure 2.10) of length $2N$

$$w_n = \begin{cases} u_n, & 0 \le n \le N-1, \\ u_{N-n-1}, & N-1 \le n \le 2N-1, \end{cases} \tag{2.76}$$

as

$$\mathcal{F}w_k = \sum_{n=0}^{2N-1} w_n \exp\frac{-nk2\pi i}{2N} \tag{2.77}$$

$$= \sum_{n=0}^{N-1} \left(w_n \exp\frac{-nk2\pi i}{2N} + w_{2N-n-1} \exp\frac{-(2N-n-1)k2\pi i}{2N} \right) \tag{2.78}$$

$$= \sum_{n=0}^{N-1} u_n \left(\exp\frac{-nk2\pi i}{2N} + \exp\frac{-2Nk2\pi i}{2N} \exp\frac{-(-n-1)k2\pi i}{2N} \right) \tag{2.79}$$

$$= \sum_{n=0}^{N-1} u_n \exp\frac{k\pi i}{2N} \left(\exp\frac{-(2n+1)k\pi i}{2N} + \exp\frac{+(2n+1)k\pi i}{2N} \right) \tag{2.80}$$

$$= 2\exp\frac{k\pi i}{2N} \sum_{n=0}^{N=1} u_n \cos\frac{(2n+1)k\pi}{2N} \tag{2.81}$$

$$= \exp\frac{k\pi i}{2N} \hat{u}_k, \tag{2.82}$$

where $\mathcal{F}$ denotes the FFT and $\hat{u}_k$ are the coefficients in the cosine expansion. We have computed the DCT, up to a rotation in the complex plane by an FFT of double the length of the data set.

Realising that $v_n = w_{2n}$ and $v_n = w_{2N-2n-1}$, for $n = 0, ..., N$, we get with very similar computations,

$$\mathcal{F}w_k = \sum_{n=0}^{2N-1} w_n \exp\frac{-nk2\pi i}{2N} \tag{2.83}$$

$$= \sum_{n=0}^{N-1} \left(v_n \exp \frac{-2nk2\pi i}{2N} + v_n \exp \frac{-(2N-2n-1)k2\pi i}{2N} \right) \tag{2.84}$$

$$= \sum_{n=0}^{N-1} \left(v_n \exp \frac{-nk2\pi i}{N} + v_n \exp \frac{k2\pi i}{2N} \exp \frac{nk2\pi i}{N} \right) \tag{2.85}$$

$$= \mathcal{F}v_k + \exp \frac{k2\pi i}{2N} \sum_{n=0}^{N-1} v_n \exp \frac{nk2\pi i}{N}. \tag{2.86}$$

Notice, recalling Euler's formula, that the real part of the complex exponential is symmetric. Therefore, we have

$$Re\left(\sum_{n=0}^{N-1} v_n \exp \frac{nk2\pi i}{N} \right) = Re\left(\sum_{n=0}^{N-1} v_n \exp \frac{-nk2\pi i}{N} \right) = Re(\mathcal{F}v_k), \tag{2.87}$$

where Re isolates the real part, and which we use to obtain

$$Re\left(\left(1 + \exp \frac{k2\pi i}{2N} \right) \mathcal{F}v_k \right) = Re(\mathcal{F}w_k) = Re\left(\exp \frac{k\pi i}{2N} \hat{u}_k \right), \tag{2.88}$$

which we rearrange to

$$Re(\hat{u}_k) = Re\left(2 \exp \frac{k2\pi i}{4N} \mathcal{F}v_k \right). \tag{2.89}$$

This allows to compute the DCT of N real valued data via an FFT of a reordered and rotated sequence of the same length N. However, because the complex part has to be dropped, this cannot be used on complex data. The inverse DCT can be expressed entirely analogously via the inverse FFT.

2.5.8 The ultraspherical method

The ultraspherical method [123] is an extension of the Chebychev base, for constructing sparse derivative operators for orders higher than one. It uses ultraspherical polynomials $C_n^{(\lambda)}$, which are normalized Gegenbauer polynomials. The λth derivative of the Chebychev T polynomials is sparse in $C_n^{(\lambda)}$

$$\partial_x^\lambda T_n(x) = 2^{\lambda-1}(\lambda-1)!(\lambda+n)C_{n-\lambda}^{(\lambda)}(x), \quad \lambda > 0, \tag{2.90}$$

and sparse operators exist to move from one base of ultraspherical polynomials $C_n^{(\lambda)}$ to the next $C_n^{(\lambda+1)}$. The families of polynomials are related via

$$C_n^{(\lambda)} = \frac{\lambda}{\lambda+n} C_n^{(\lambda+1)} - \frac{\lambda}{\lambda+2+n} C_{n-2}^{(\lambda+1)}, \quad \lambda > 0, \tag{2.91}$$

which can be written as a matrix S_λ, with entries on the main diagonal and one off-diagonal. S_0 is obtained via the Chebychev-T to Chebychev-U conversion described above.

The method can be understood as a preconditioned Chebychev method. Note that this implies that it has the same convergence properties as the Chebychev method. The

ultraspherical method is equivalent to constructing the matrices in Chebychev base and then left-multiplying by the operator that converts from $T = C^{(0)}$ to $C^{(\lambda)}$

$$B_\lambda = S_{\lambda-1} S_{\lambda-2} ... S_0. \tag{2.92}$$

The preconditioner is then chosen according to the highest occurring derivative in the equation. For instance, if the equation $u + \partial_x u + \partial_x^2 u = 0$ is to be solved, the system should be preconditioned with B_2. B_2 is banded with non-zero values on three diagonals. The term u will have the same structure, while the term $\partial_x^2 u$ will have only one non-zero diagonal. The term $\partial_x u$ will have non-zero values on two diagonals because S_2 has two non-zero diagonals and the term has only one after multiplication by $S_1 S_0$.

A system of equations is preconditioned using multiple B_λ. For instance, a system of equations $\partial_x^2 u + \partial_x^2 v = a$ and $\partial_x v = b$ would be preconditioned with

$$\begin{pmatrix} B_2 & 0 \\ 0 & B_1 \end{pmatrix} \begin{pmatrix} D^2 & D^2 \\ 0 & D \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} B_2 & 0 \\ 0 & B_1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix},$$

where D is the first derivative matrix in Chebychev base (Equation 2.72). The matrix that needs to be inverted to solve this system has blocks which have non-zero values on at most one diagonal.

We do not give an explicit form of the $C^{(\lambda)}$, because it is never needed in practice. The series expansion is computed in Chebychev-T and the result of the left-preconditioned matrix inversions is also in Chebychev-T. When we evaluate derivatives, we multiply by the inverse preconditioner in order to return to Chebychev-T base. While the inverse of the preconditioner is expensive to compute, we do this only once and blockwise before the simulation starts and can then use the inverse DCT to return to physical space efficiently.

2.5.9 Parallelizing Chebychev and ultraspherical spectral methods

Because the derivative matrices are not diagonal, the Chebychev and ultraspherical bases offer no way to parallelize the evaluation and inversion of derivatives. The DCT, on the other hand, can also be distributed just like the FFT.

As discussed in section 2.5.3, d-dimensional discretizations are set up via Kronecker products of one-dimensional bases. When combining Chebychev or ultraspherical basis in one direction with Fourier basis in other directions, the d-dimensional derivative matrices can be constructed from local derivative matrices in directions discretized with Fourier basis. Therefore, we retain the ability to distribute evaluation and inversion of derivatives along directions discretized with Fourier base, but we need to solve linear systems involving all points in directions discretized with Chebychev or ultraspherical base locally.

2.5.10 Boundary conditions in spectral methods

We discuss two approaches for dealing with boundary conditions in spectral methods here, although this discussion is not exhaustive. One approach is to choose a basis where any linear combination of elements satisfy the boundary conditions. This is the case with the Fourier base and periodic boundary conditions, for instance.

The second approach is usually referred to as boundary bordering or τ-method. Here, the equation is perturbed by adding a polynomial, thus ensuring a polynomial solution

exists, and the boundary condition is explicitly added to the matrix to be solved. Consider the problem

$$\begin{aligned} L\vec{u} &= \vec{a}, \quad L \in \mathbb{R}^{N\times N}, \\ B\vec{u} &= \vec{b}, \quad B \in \mathbb{R}^{n\times n}, \end{aligned} \tag{2.93}$$

with L the discretization of the PDE with N degrees of freedom and B the discretization of n boundary conditions.

In the τ-method, the equation is perturbed with so called τ-terms:

$$\begin{pmatrix} L & I_n \\ B & 0 \end{pmatrix} \begin{pmatrix} \vec{u} \\ \vec{\tau} \end{pmatrix} = \begin{pmatrix} \vec{a} \\ \vec{b} \end{pmatrix}, \tag{2.94}$$

with I_n the $n \times n$ identity.

The τ-terms are not part of the solution to the perturbed problem and it is possible to forego computing them by replacing the last n lines of L with B and the last n lines of $\vec{a}$ with $\vec{b}$:

$$\tilde{L}\vec{u} = \tilde{\vec{a}}, \quad \tilde{L} \in \mathbb{R}^{NxN}, \tag{2.95}$$

$$\tilde{L}_{ij} = \begin{cases} L_{ij}, & i < N-n, \\ B_{ij}, & i \geq N-n, \end{cases} \tag{2.96}$$

$$\tilde{a}_i = \begin{cases} a_i, & i < N-n, \\ b_i, & i \geq N-n. \end{cases} \tag{2.97}$$

The boundary condition operator B is formed by evaluating the basis elements at the boundary. For instance, the Chebychev-T polynomials satisfy

$$T_i(\pm 1) = (\pm 1)^i, \tag{2.98}$$

which easily allows to implement Dirichlet boundary conditions at $x = \pm 1$. Note that in the ultraspherical method, the boundary conditions are also implemented in the ingoing base of Chebychev-T.

The boundary conditions impact the sparsity of the matrix because they add densely populated lines. We implement a technique known as Dirichlet preconditioning or basis recombination [22], which is another change of base to Dirichlet polynomials where only the first two elements are non-zero at the boundary $x = \pm 1$. This is a right preconditioner, because it changes the incoming base. Note that this can be used as an alternative method for enforcing boundary conditions for Poisson-type problems. The homogeneous problem can be solved on all but the first two degrees of freedom, which are defined by the boundary conditions.

We give an example of the τ-method in the simple polynomial base, solving $u_x = -1$ with boundary condition $u(-1) = 1$. Since we know the solution to be $u^*(x) = -x$ with trivial expansion, we limit this example to $N = 3$. The system including τ-terms is

$$\left(\begin{array}{ccc|c} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \\ \hline 1 & -1 & 1 & 0 \end{array}\right) \begin{pmatrix} \hat{u}_0 \\ \hat{u}_1 \\ \hat{u}_2 \\ \tau \end{pmatrix} = \left(\begin{array}{c} -1 \\ 0 \\ 0 \\ \hline 1 \end{array}\right), \tag{2.99}$$

where we inserted lines for distinguishing between the differentiation matrix in the top left, the τ-term on the right and the boundary condition on the bottom left. In practice, the τ-term can be removed from the computation and the system simplified to

$$\left(\begin{array}{ccc} 0 & 1 & 0 \\ 0 & 0 & 2 \\ \hline 1 & -1 & 1 \end{array}\right) \hat{\vec{u}} = \left(\begin{array}{c} -1 \\ 0 \\ \hline 1 \end{array}\right). \tag{2.100}$$

Solving both systems gives $\hat{\vec{u}} = (0, -1, 0)^T$, which is the exact solution. Solving the first system additionally gives $\tau = 0$.

2.5.11 τ-Method and SDC

The perturbations introduced by the τ-terms are small if the problem is sufficiently resolved. For instance, in the above example, we can solve Equation 2.99 to find $\tau = 0$. However, when the resolution is too low, the τ-terms start to grow and impact the SDC residual.

To illustrate, we give another example in regular polynomials. We search for the solution of the Poisson problem $\Delta u = 12x^2$ with boundary conditions $u(-1) = -1$ and $u(1) = 1$ and $N = 5$ elements. The system we need to solve is

$$\begin{pmatrix} 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 12 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \end{pmatrix} \hat{\vec{u}}_5 = \begin{pmatrix} 0 \\ 0 \\ 12 \\ 1 \\ -1 \end{pmatrix}, \tag{2.101}$$

where the first three lines are the non-zero entries from the second derivative matrix and the bottom two lines are the boundary conditions. The exact solution is $u^* = x^4 + x - 1$, which we indeed recover from solving the system. However, consider now the case of $N = 4$ modes:

$$\begin{pmatrix} 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 6 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix} \hat{\vec{u}}_4 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ -1 \end{pmatrix}, \tag{2.102}$$

which has solution $u_4 = x$. This is the solution of the homogeneous Poisson problem because the source term has been replaced by the boundary conditions. Computing the τ-terms, we find $\tau_3 = 12$ and $\tau_4 = 0$. Clearly, four modes is not enough to resolve this problem, which will become apparent in the residual:

$$\|\Delta u_5 - 12x^2\| = \|\Delta(x^4 + x - 1) - 12x^2\| = 0 \tag{2.103}$$

$$\|\Delta u_4 - 12x^2\| = \|\Delta x - 12x^2\| = \|12x^2\|. \tag{2.104}$$

In this case, the τ-terms remain, because they are not accounted for in the residual.

The same issue appears in SDC. When the perturbation due to the τ-terms is non-negligible, the SDC residual cannot be reduced to arbitrarily small values without accounting for the τ-terms. Note that SDC will still converge to a valid, albeit under-resolved, solution to the discretized problem. If the SDC residual is to be used for stopping the

SDC iteration regardless of the spatial truncation error, the SDC residual has to be modified. Since we avoid computation of the τ-terms in practice, we did not pursue such a modification and leave this for future work.

2.5.12 Pseudo-spectral methods

Issues arise with spectral methods when the problems are non-linear. For instance, multiplication in Fourier space corresponds to convolution in real space and vice versa. Consequently, if the problem contains a multiplication, it has to be computed using convolution in spectral space, which is a very expensive operation involving all data points.

Pseudo-spectral methods avoid this by treating non-linear terms in physical space, where they are easy to evaluate, and treating only linear terms in spectral space. We illustrate the procedure in Algorithm 1 and later give an example in section 2.6.3 of a reaction-diffusion equation, where the stiff diffusion term is inverted and evaluated in spectral space and the non-linear reaction term is handled explicitly in physical space.

Algorithm 1 Evaluation of the right hand side in pseudo-spectral methods

$\hat{\vec{u}} \leftarrow$ Solution in spectral space
$\vec{u} \leftarrow$ Inverse transform of $\hat{\vec{u}}$
▷ Transform here refers to the operation for transferring from physical to spectral space, such as FFT for Fourier methods or DCT for Chebychev methods.
$\hat{\vec{f}}_{\text{linear}} \leftarrow$ Linear terms evaluated at $\hat{\vec{u}}$ in spectral space
$\vec{f}_{\text{nonlinear}} \leftarrow$ Nonlinear terms evaluated at $\vec{u}$ in physical space
$\hat{\vec{f}}_{\text{nonlinear}} \leftarrow$ Transform of $\vec{f}_{\text{nonlinear}}$
$\hat{\vec{f}} \leftarrow \hat{\vec{f}}_{\text{linear}} + \hat{\vec{f}}_{\text{nonlinear}}$

2.6 Benchmark problems

We will now introduce four non-linear test problems that we use to demonstrate various results. Two ODE toy problems are used to illustrate individual properties of the schemes. Two PDE examples will show that they can be employed in a practically useful setting.

2.6.1 Van der Pol oscillator

The van der Pol equation

$$\begin{aligned} u_{tt} - \mu\left(1 - u^2\right) u_t + u = 0, \\ u(t=0) = u_0, \quad u_t(t=0) = u_0', \end{aligned} \tag{2.105}$$

is named after a Dutch electrical engineer who used the equation to study the behavior of vacuum tubes in radios [128]. Here, μ is a parameter controlling the non-linearity and u is the solution. In our implementation, we introduce $v(t) = u_t(t)$, rewrite the van der Pol equation as a first order system and use a Newton scheme to solve the nonlinear systems within the SDC sweeps.

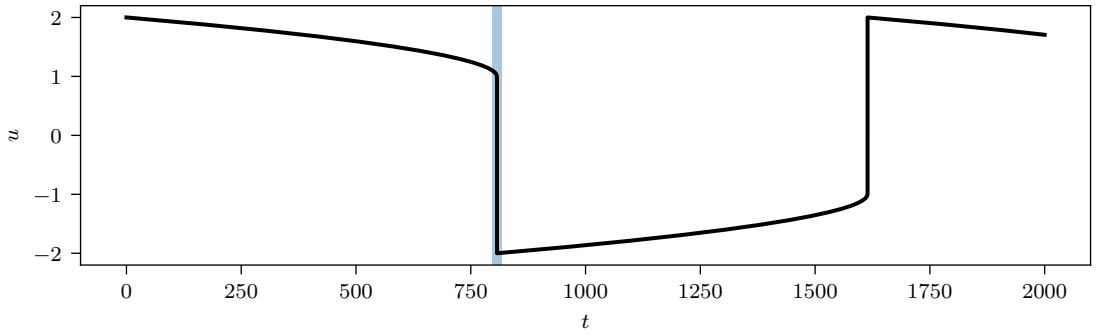


Figure 2.11: Solution of a van der Pol problem for $\mu = 1000$ over time. The solution is oscillating on two time-scales. In order not to over-resolve the slow parts, the resolution has to be adjusted during runtime. In numerical tests for computational efficiency in chapter 4, we solve only the shaded transition at $t \approx 800$ as this is already very expensive with fixed step size schemes. Note that at this high value of μ, the problem is extremely stiff. Figure reproduced from [11, Fig. 2] under CC BY 4.0 licence.

For $\mu = 0$, we recover the harmonic oscillator, but with increasing μ the problem becomes increasingly stiff. Van der Pol describes the problem with $\|\mu\| \ll 1$ as modelling free oscillations of a triode oscillator, whereas $\|\mu\| \gg 1$ models a free relaxation oscillation [129]. This is a useful test problem for adaptive step size control as the nonlinear damping introduces a second time-scale to the oscillation. See Figure 2.11 for an illustration of the solution over time with the configuration used in chapter 4. We will describe the problem configurations we use in tests in the respective chapters.

We use the `SciPy` [157] method SOLVE_IVP from the INTEGRATE package with an explicit embedded Runge-Kutta method of orders 5 and 4 (RK5(4)) [46] with tolerances close to machine precision to obtain reference solutions.

2.6.2 Lorenz attractor

Lorenz introduced this problem as a simplified system modelling phenomena that cause difficulty in numerical weather prediction [107]. Lorenz played a key role in the development of chaos theory when he attempted to demonstrate advantages of dynamical physics based weather prediction over statistics based prediction efforts.

Lorenz's experiment entailed reproducing contrived and unusual weather that the statistical methods would not be able to cope with. To his surprise, he could not reproduce the results using the dynamical model either, even though it is deterministic. He discovered that the problem lay in roundoff errors in the fourth decimal place that appeared when he fed simulation results back into his machine for starting the next simulation.

In this experiment, he discovered that numerical weather prediction exhibits chaotic behaviour, which means that small perturbations grow over time and eventually have a large impact on the solution. This makes it impossible to make accurate long time weather forecasts because the measurements that form the initial conditions have only finite accuracy. To some extent, this was already figured out by Lewis Fry Richardson, one of the pioneers in numerical weather prediction, who computed weather hindcasts by hand during the first world war. His computations were significantly off, but he correctly blamed it on the initial conditions rather than the computations themselves [48].

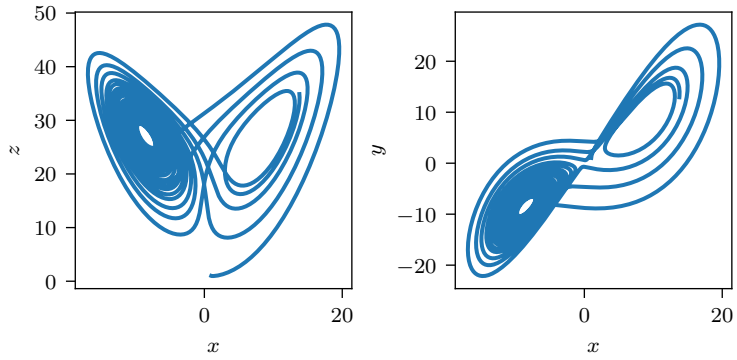


Figure 2.12: Solution of the Lorenz attractor problem over time with the parameters from Equation 2.107. Shown are projections in the x-z plane (left) and x-y plane (right). The trajectory oscillates around two attractors in a chaotic manner. Figure reproduced from [12, Fig. 9]

The Lorenz attractor is the result of Lorenz's efforts to find the simplest set of equations that exhibit the chaotic behavior found in numerical weather prediction. It is a nonlinear coupled system of ODEs, which can be written as

$$\begin{aligned} x_t &= \sigma\left(y - x\right), \\ y_t &= \rho x - y - xz, \\ z_t &= xy - \beta z. \end{aligned} \tag{2.106}$$

We use parameters, initial conditions and time domain

$$\begin{aligned} &\sigma = 10,\ \rho = 28,\ \beta = 8/3, \\ &x(t=0) = y(t=0) = z(t=0) = 1, \\ &t \in [0, 20]. \end{aligned} \tag{2.107}$$

We solve the problem implicitly with a self-constructed Newton solver and obtain reference solutions with the same SciPy implementation of explicit RK5(4) we used for the van der Pol problem.

The chaotic behavior is manifest in that the trajectory will never repeat itself. The name attractor comes from the fact that there are two positions that the solution will oscillate about, flipping between the two attractors often, but unpredictably. Examples of such attractors in weather and climate are, for instance, ice ages. The sensitivity of this problem to perturbations allows to test the robustness of the adaptive procedure as small irregularities readily propagate to the final solution. See Figure 2.12 for a visualization of the solution over time.

2.6.3 Gray-Scott

The Gray-Scott equation is a reaction-diffusion equation modelling cubic autocatalytic reactions [62]. This means one of the reactants is a catalyst for the reaction and three molecules are involved. We write the involved reactions as

$$U + 2V \to 3V, \tag{2.108}$$

$$V \to P, \tag{2.109}$$

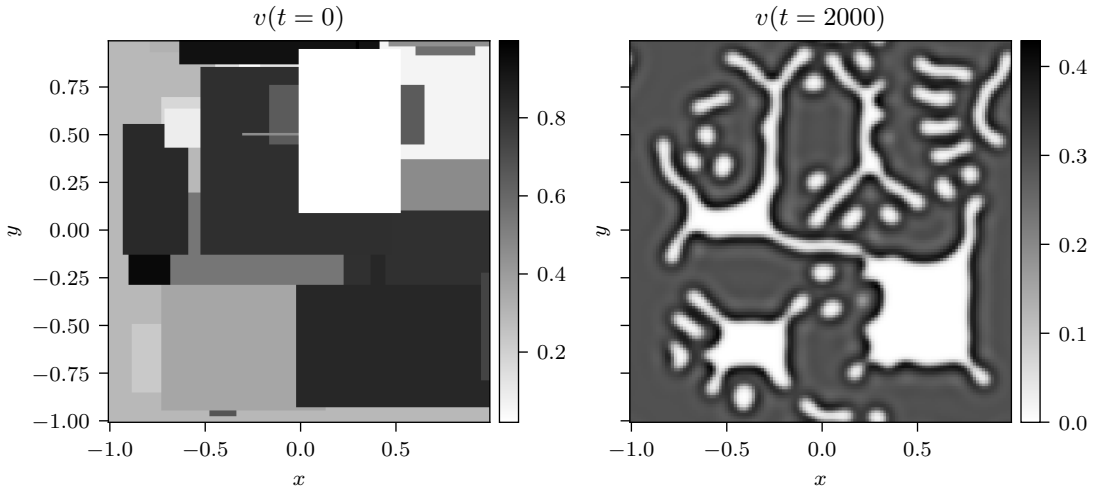


Figure 2.13: Solution of the v component in the Gray-Scott problem with "U-Skate World" [115] configuration with $F = 0.062$ and $k = 0.0609$ on a $N = 128^2$ grid. Left: random rectangles as initial conditions. Right: solution after $t = 2000$. See [10, GrayScott2D.mp4] for a video of the solution over time.

where the first reaction with the reactants U and V is the cubic autocatalytic part and the second reaction results in an inert product P [101]. The reactions can be modelled with spatial extent as a PDE

$$\begin{aligned} u_t &= \nu_u \Delta u - uv^2 + F(1-u), \\ v_t &= \nu_v \Delta v + uv^2 - (F+k)v, \end{aligned} \tag{2.110}$$

with u and v the concentration of U and V, F the feed rate and k the kill rate.

The Laplacians model diffusion of the concentrations in space. The terms $\pm uv^2$ model the cubic autocatalytic reaction in Equation 2.108. The term $+F(1-u)$ adds U based on the current concentration and the term $-(F+k)v$ models the reaction from V to the inert product in Equation 2.109.

Like reaction diffusion equations can be used to model various natural phenomena ranging from solid-gas interactions [3] to the spread of diseases [20]. This particular equation is known for producing highly complex patterns for particular choices of F and k [127] due to so called Turing instability [156].

We discretize the equation using a Fourier pseudo-spectral method with periodic boundary conditions and IMEX splitting. We evaluate and invert the Laplacians in spectral space and evaluate the nonlinear reaction terms explicitly in physical space. See Figure 2.13 of a two-dimensional example similar to the one used in chapter 4 and chapter 5. We compute reference solutions with our own implementation in `pySDC` (see section 2.8.2) by choosing very tight $\epsilon_{\text{TOL}} = 10^{-8}$ with Δt-adaptivity (see section 3.1).

2.6.4 Rayleigh-Benard convection

Rayleigh-Benard convection (RBC) describes the dynamics that occur in a fluid that is heated from below, cooled from above and subject to gravity. The governing equations

can be written as

$$\begin{aligned} u_t - \nu(u_{xx} + u_{zz}) + p_x &= -uu_x - vu_z, \\ v_t - \nu(v_{xx} + v_{zz}) + p_z - T &= -uv_x - vv_z, \\ T_t - \kappa(T_{xx} + T_{zz}) &= -uT_x - vT_z, \\ u_x + v_z &= 0, \end{aligned} \tag{2.111}$$

with u the velocity in x-direction, v the velocity in z-direction, T the temperature, p the pressure and κ the thermal diffusivity and ν the kinematic viscosity [106, Equations (1)-(3)].

The temperature profile generates buoyancy through thermal expansion of the fluid that is heated near the bottom. The upward force due to the reduced density is proportional to the temperature gradient ΔT, the thermal expansion coefficient of the fluid α and the gravitational acceleration g. However, friction in the form of fluid viscosity and stabilizing thermal conduction act against the movement, which are proportional to the kinematic viscosity, the thermal diffusivity and the separation between the plates L_z. The Rayleigh number

$$Ra = \frac{\alpha g \Delta T L_z^3}{\kappa \nu} \tag{2.112}$$

is a non-dimensionalized quantity describing the balance between between the forces. At low Rayleigh number the flow is dominated by diffusion, whereas convection becomes more and more prominent with increasing Rayleigh number. At a Rayleigh number greater than approximately 1708, turbulence develops [106]. The Prandtl number

$$Pr = \frac{\nu}{\kappa} \tag{2.113}$$

is an inherent property of the fluid and the flow patterns are further determined by the aspect ratio of the domain L_x/L_z [37].

In Equation 2.111, the third equation describes the evolution of the temperature distribution. The term $\kappa(T_{xx} + T_{zz})$ describes diffusion or thermal conduction, while the term uT_x models horizontal convection with the fluid velocity and vT_z models vertical convection. The first two equations model the evolution of the horizontal and vertical fluid velocities. Due to the viscous damping, there is again a diffusion term and there are convection terms just like in the temperature component. There is, however, an additional term influencing the fluid velocity based on the pressure gradient. Note that the absolute pressure never appears in the equations, which means a constant offset is to be chosen which has no impact on the simulation. This is called the pressure gauge and is completely arbitrary. In the second equation, the vertical velocity has a term proportional to the temperature, which is absent in the horizontal velocity because the direction of gravity is defined in z-direction. Finally, the fourth equation $u_x + v_z = 0$ is the incompressibility constraint. Intuitively, it means that if there is vertical inflow into a cell in space, there has to be a compensating horizontal outflow and vice versa.

In all simulations, we use $Pr = 1$ and a spatial domain of size $\Omega = [0, 8) \times (-1, +1)$ ($L_x/L_z = 4$), with periodic boundary conditions in x-direction and the following Dirichlet boundary conditions in z-direction:

$$u(z = -1) = u(z = 1) = v(z = -1) = v(z = 1) = T(z = 1) = 0 \tag{2.114}$$

$$T(z=-1)=2. \tag{2.115}$$

We choose $\int_\Omega p = 0$ for the pressure gauge.

The incompressibility constraint makes RBC a differential algebraic equation (DAE). Applying SDC to DAEs [77] is its own research topic by itself, but the simple nature of this algebraic constraint allows to use stiffly accurate RKM or quadrature rules where the nodes include the right end-point without modification [6]. This is because the solution to the step in these methods is not obtained by linear combination of intermediate solutions, but is one of the intermediate solutions itself. In methods where this is not the case, the pressure is not correctly computed in the collocation update and instead needs to be computed in a separate step [33]. We do not implement such schemes here and, therefore, restrict to Gauß-Radau nodes or stiffly accurate RKM.

We use a pseudo-spectral method for discretization, where we use Fourier base horizontally and ultraspherical base vertically and IMEX splitting. We already put the linear terms on the left hand side of Equation 2.111, which we will treat implicitly in spectral space while treating the convection terms on the right hand side explicitly in physical space. We rewrite the equation as

$$\begin{aligned}
M\vec{u}_t + L\vec{u} &= f_{\text{nonlin}}(\vec{u}) \\
\vec{u} &= (u, v, T, p)^T \\
M &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\
L &= \begin{bmatrix} -\partial_x^2 - \partial_z^2 & 0 & 0 & \partial_x \\ 0 & -\partial_x^2 - \partial_z^2 & -1 & \partial_z \\ 0 & 0 & -\partial_x^2 - \partial_z^2 & 0 \\ \partial_x & \partial_z & 0 & 0 \end{bmatrix} \\
f_{\text{nonlin}}(\vec{u}) &= -(uu_x + vu_z, uv_x + vv_z, uT_x + vT_z, 0)^T.
\end{aligned} \tag{2.116}$$

The resulting IMEX Euler solver that can be used in the SDC iterations is

$$(M + \Delta t L)\,\vec{u} = M\vec{u}_0 + \Delta t f_{\text{nonlin}}(\vec{u}_0). \tag{2.117}$$

Note that while the mass matrix M is not invertible, as is characteristic of DAEs, $M + \Delta t L$ is invertible, for nonzero step size.

We insert $2 \times n_x$ τ terms for the vertical boundary conditions on u, v and T in the equations for u_t, v_t and T_t. Note that incompressibility requires the z-derivative of v to vanish on the 0 mode in x because the x-derivative of the constant 0 mode in x is zero. Therefore, we need to remove one of the boundary conditions of v in this mode in order not to overdetermine v. We insert the pressure gauge in the constant horizontal and last vertical mode in the incompressibility equation, which is required to close the system in this discretization.

We show an example of a simulation in Figure 2.14. Similarly as in the Gray-Scott example, we compute reference solutions with our implementation within `pySDC` by choosing very tight $\epsilon_{\text{TOL}} = 10^{-8}$ with Δt-adaptivity. We verified the reference solution by confirming the order of accuracy of the converged collocation problem up to order five.

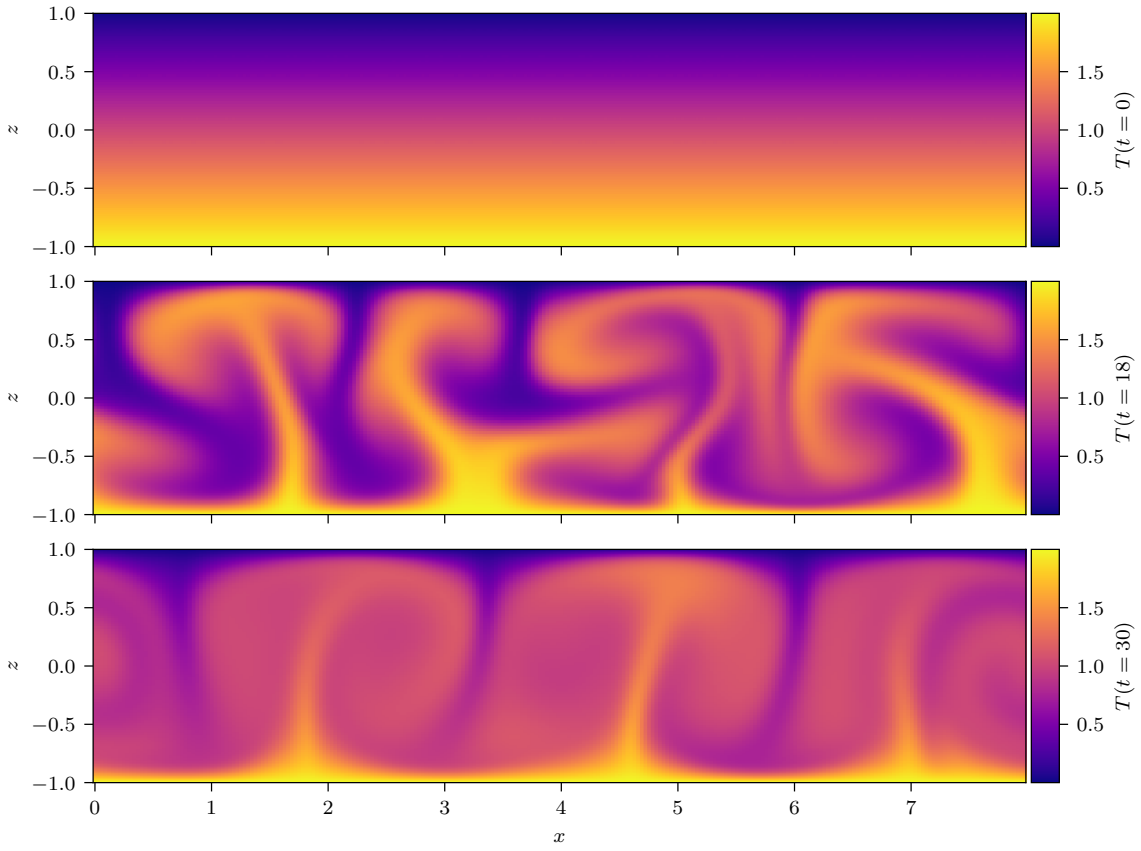


Figure 2.14: Example of an RBC simulation with $Ra = 2 \times 10^4$ and $N = 256 \times 128$. Shown is the temperature profile at different times. Top panel: Initial temperature profile, consisting of a linear gradient and perturbations on the the order of 10^{-3}. Middle panel: Buoyancy has led to upwelling, but when the plumes reach the top, they cannot keep rising. Instead, they are pushed to the side and back down by continued upwelling. Bottom panel: After a while, the situation has settled into multiple pseudo-stationary circular flow patterns. See [10, RBCLowRayleigh.mp4] for a video of the solution over time.

Recall from section 2.5.11 that insufficient resolution in combination with τ-methods can lead to the SDC residual not reaching small values even if SDC converges to the collocation solution. Since very large gradients appear at the boundaries in RBC, quite large resolution is needed to achieve small SDC residuals.

2.7 GPUs

Most modern HPC systems are made up of a combination of central processing units (CPUs) and graphics processing units (GPUs). The majority of the compute power increasingly comes from the GPU part, however. We give a brief summary of the design differences between the two following [158]. CPUs have few but powerful cores that are designed to execute individual operations as quickly as possible. GPUs, on the other hand, have many cores, which are individually less powerful, but can result in very high throughput for operations that can be spread across the different cores. GPUs, therefore, are built around concurrency on device. Another difference is that CPUs try to keep memory latency to a minimum, whereas GPUs aim to hide memory latency by overlapping with computation.

We now discuss how parallelism is implemented on NVIDIA GPUs as we will employ in chapter 6 and chapter 7 following [121]. The cores on a GPU are referred to as streaming multiprocessors (SMs). Each SM can concurrently execute blocks of threads, which, in turn, can execute individual threads in parallel. This is managed with the so called single-instruction, multiple-thread (SIMT) architecture, where threads are scheduled in groups of 32, called warps. This allows for very fine-grained concurrency of tasks with many independent operations [81]. As an example, consider two-dimensional FFTs. As discussed in section 2.5.2, these entail one-dimensional FFTs that can be executed concurrently. When doing so on a single GPU, they can be distributed among the SMs and then further parallelised on each SM by scheduling individual operations within FFTs in individual threads.

2.7.1 CUDA-specific programming

Harnessing the compute power of GPUs requires a different programming model compared to CPUs. We will give a brief overview here of the central concepts we rely on when porting code to GPU in chapter 6. As we specialize on NVIDIA GPUs, which use the CUDA API, this section is limited to CUDA and not GPUs in general. See the CUDA programming guide [121] for more extensive information.

Kernels. Kernels are similar to functions, but add the interface for thread parallelism. A kernel can be launched N times simultaneously on N threads. Within the kernel, the thread index can be used to execute different commands. For instance, when the same operation is to be performed on an array of length N, a suitable kernel would be launched with N tasks, and the thread index would be used as an index for the array within the kernel.

Launching a kernel comes with its own cost on the order of micro seconds [61], independently of what the kernel does. For very small kernels, the launch cost can therefore dominate the overall time taken by the kernel.

Streams. The mode of operation of GPUs is fundamentally different from CPUs. On CPUs, when the interpreter encounters an instruction, it is executed, the interpreter waits for completion, and then moves on to the next instruction. GPUs, on the other hand, rely on their host CPUs to submit operations to queues, called "streams" on the devices. When the interpreter on the CPU encounters a GPU instruction, it is launched on the GPU and typically returns right after, meaning the interpreter on the CPU moves on to the next instruction while the GPU executes it asynchronously from the host.

Concurrency within single GPUs can be facilitated by submitting independent operations to separate streams, which then proceed asynchronously and are scheduled concurrently. Arbitrarily complex patterns of concurrency can be designed in a simple fashion by multiple streams and the use of so called CUDA events to synchronize between the streams. The CUDA runtime will automatically translate the concurrency to parallelism by distributing the operations among the available SM, provided the resources suffice.

For using GPUs efficiently, the queue should always be filled in order to avoid idleness. Hence, care should be taken to avoid synchronization between CPU and GPU, since once the GPU has caught up to the CPU, the CPU needs time to submit new operations. Synchronization happens implicitly during memory operations copying between the two, for instance, and needs to be done explicitly when the CPU accesses data residing on GPU.

CUDA graphs. A CUDA graph is a collection of nodes and edges, with the nodes representing operations and the edges representing dependencies between the operations. After a graph is created, all encompassed operations are launched as a single kernel with the CUDA system automatically scheduling independent operations concurrently on a low level.

This enables efficient use of the device even when using many very small kernels. As mentioned, launching kernels incurs overhead by itself and launching the graph as a single kernel reduces this overhead compared to individually launching the kernels that make up the graph.

CUDA graphs provide an interface for the programmer to optimize concurrency with very little effort, simply by recording the operations issued to a stream. After the recording process, the graph is created during its first execution, which incurs some overhead. Subsequent executions, on the other hand, are very efficient.

2.7.2 NCCL

The NVIDIA Collective Communication Library (NCCL) is specifically developed for communication across multiple NVIDIA GPUs. While there are implementations of the established MPI standard that work with GPUs, there are a couple of shortcomings of these libraries that are addressed by NCCL. We will first detail these differences in functionality and then briefly mention differences in the interface.

CUDA-aware MPI. The aforementioned MPI implementations that are compatible with NVIDIA GPUs are called CUDA-aware and are capable of communicating data directly between GPUs with the fastest available interconnect. However, the implementations are held back by a philosophical decision in the MPI standard.

Algorithm 2 Using CUDA-aware MPI. Before data residing on GPU can be communicated, the host needs to be synchronized to the GPU in order to make sure all operations on the data have completed.

$x \leftarrow$ some data on CPU
$y \leftarrow$ some data on GPU
Perform some operations on x
Submit some operations on y to stream s on GPU
$x_2 \leftarrow$ Communicate x with MPI
Synchronize host to stream s on GPU
$y_2 \leftarrow$ Communicate y with MPI

The MPI standard was first released in 1994 by the MPI forum, stating the following as their mission [111]:

> The goal of the Message-Passing Interface, simply stated, is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message-passing.

A portable interface, however, is counter to vendor-specific programming models. While proper use of GPUs requires to submit operations to streams, the interface does not have this option and in fact the latest version of the standard at the time of writing [111] does not even mention GPUs at all.

While individual MPI implementations have the freedom to support vendor specific features, they have to adhere to the standard and the interface that it defines. A common approach to reconcile the standard interface with GPUs is to simply use a unified address space for CPU and GPU memory and to initiate the communication on CPU. If the MPI implementation treats GPU memory the same as CPU memory rather than submitting the communication operation to a stream on GPU, however, the communication may execute before the GPU has finished all prior operations. Hence, explicit synchronisation by the programmer is required as shown in Algorithm 2. This often leads to reduced utilisation of GPUs because they may run idle while the host handles the communication instead of submitting further kernels [81].

The second major drawback of current MPI implementations regarding GPUs is that they typically perform all reduce operations on the host. When a reduce operation is called on data residing on GPU, it is copied back to host, the operation is computed on host, and then the data is copied back to GPU, which is an overall very inefficient method.

These issues are not insurmountable for MPI implementations. For instance, UCC, a communications framework behind MPI, can be set up to use NCCL as backend for reduce operations [85]. This way, the programmer interacts with MPI with no special concern for GPUs, but the communication is submitted as a kernel to the default stream and is handled entirely on GPU.

Differences between CUDA-aware MPI and NCCL. The most striking difference between NCCL and current CUDA-aware MPI implementations is that NCCL runs on the GPUs. The host submits NCCL communication operations to a stream on GPU just

Algorithm 3 Ring-send with non-blocking communication in MPI and with NCCL groups. NCCL does not have the concept of non-blocking communication as in MPI. Instead, deadlocks are avoided by posting multiple point-to-point communications within a NCCL group, with are automatically overlapped.

```
r ← rank in the communicator
s ← size of the communicator

x ← some data on CPU
req ← MPI non-blocking send of x to (r + 1)%s
x_2 ← MPI receive from (r − 1 + s)%s
MPI wait for req to complete

y ← some data on GPU
NCCL group start
NCCL send of y to (r + 1)%s
y_2 ← NCCL receive from (r − 1 + s)%s
NCCL group end
```

like any other kernel and they run no sooner than all previous kernels in the same stream have completed. No explicit synchronization is necessary, allowing for better utilization of GPUs. Similarly, reduce operations are tackled completely on GPU.

NCCL also promises to be more optimised than MPI in routing the data. While both libraries utilise the same cables, NCCL automatically detects the topology [120] for selecting possibly more efficient paths.

While NCCL addresses the most critical shortcoming of MPI regarding GPUs by being stream-aware, it is less mature and lacks some features. For instance, it does not officially support complex numbers. An easy workaround is to select the real datatype of the same precision and increase the buffer-size by a factor of two, but working with NCCL can be less streamlined compared to MPI. A more severe restriction is that NCCL supports only a single task per GPU. NVIDIA develop MPS (Multi-Process Service) for using multiple MPI tasks on a single GPU, but this tool is not compatible with NCCL.

Differences in the interface between NCCL and MPI. The interface is generally similar, making porting from MPI to NCCL relatively straightforward. The main difference is that NCCL does not have non-blocking versions of communication functions. Instead, the concept of NCCL groups is used to overlap point-to-point communication. See Algorithm 3 for an illustration of a ring-send implementation in NCCL. When multiple communication operations are posted within a NCCL group, they are automatically overlapped and merged. This has the purpose of preventing deadlocks on the one hand, but also to speed-up communication by submitting it as a single kernel with efficiently routed data.

Another difference is that NCCL supports only the collectives reduce, all-reduce, broadcast, all-gather and reduce-scatter. All other collectives in the MPI standard, such as all-to-all, have to be implemented manually. Many collectives can be implemented efficiently and easily without explicit optimisation by the user with NCCL groups.

2.7.3 NVSHMEM

We have introduced NCCL as an alternative to MPI for NVIDIA GPUs that promises much better performance by running on the GPU as opposed to the CPU in the previous section. Similarly, NVIDIA develop NVSHMEM [81] as a library for Symmetric Hierarchical MEMory (SHMEM) [30] that efficiently utilizes their hardware.

SHMEM is a communication framework that is fundamentally different from MPI in that communication is not handled by communicators and does not involve all tasks participating in the communication. Instead, a global address space is defined across multiple tasks and communication is then handled by individual tasks. Where point-to-point communication in MPI requires a send on one task and a receive on another task (two-sided communication), in SHMEM, the receiving task executes a get operation on the global address space and the sending task is not explicitly involved (one-sided communication).

The advantage of one-sided communication is that synchronization overhead can be reduced, in particular when initiating the communication directly on GPU, e.g. [164]. This is a major advantage when communicating small messages, where synchronization overhead makes up a significant fraction of the communication cost. On the other hand, when the message size is larger, the network bandwidth is the limiting factor, reducing the benefit from one-sided communication.

We mention SHMEM-based communication only briefly here because we have not found a way to use it in Python, which is the programming language that we will use exclusively in this thesis and which we will introduce in the next section. For this simple reason we use only MPI and NCCL for communication in parallel implementations later on.

2.8 Codes used in experiments

We briefly introduce the Python language, which we use for all experiments later on, as well as the codes that we work with or modify later.

2.8.1 Python for HPC

Python often gets the reputation of being suitable for toy problems on laptops only, while being incapable of running on HPC systems at scale [99]. We briefly tackle the issues that this rumour originates from and describe how to avoid them. In fact, the experiments we do in chapter 6 and chapter 7 show that code developed in Python can indeed be HPC capable.

The philosophy of Python. Python is an interpreted language. When a Python script is run, the interpreter translates the code into machine instructions at runtime, one line at a time. This is in contrast to compiled languages where the compiler reads in the entire code and translates to an optimised machine-executable before runtime. Interpreted languages allow for greater convenience to the programmer, but usually at a substantial cost to overall performance.

It is true that the same code written as pure Python code is not competitive with a compiled version. However, the philosophy of Python is to run actually very little pure Python code and use the language for interfacing wrapped compiled code instead [157]. The community maintains a wide array of Python modules that provide such wrappers

Listing 2.1: Example for porting `NumPy` and `SciPy` code to GPU using `CuPy`. In many cases, swapping the modules is sufficient.

```
import numpy
import cupy
import scipy
import cupyx.scipy as cupy_scipy

# generate data containing 0, 1,..., 7
data_CPU = numpy.arange(8)
data_GPU = cupy.arange(8)

# basic linear algebra: add 1 elementwise
data_CPU += numpy.ones(8)
data_GPU += cupy.ones(8)

# discrete cosine transform
dct_CPU = scipy.fft.dct(data_CPU)
dct_GPU = cupy_scipy.fft.dct(data_GPU)
```

for functionality such as basic linear algebra and sparse linear solvers. Python is sometimes dubbed a "glue language" [43], as typical Python code assembles complex structures of compiled code with only the interface between the compiled modules written in pure Python.

CPU computing The cornerstone of linear algebra in Python consists of `NumPy` [75] for array manipulation and `SciPy` [157] for algorithms and sparse matrices. Performing complex operations is as simple as wrapping the data with the class NUMPY.NDARRAY, and then passing the wrapped data to various functions. `NumPy` is so ubiquitous that many Python modules accept NUMPY.NDARRAY as input. However, these libraries are explicitly limited to non-distributed computing on CPUs.

Distributed computing. The Python interpreter uses a global interpreter lock (GIL), which prevents multiple threads from executing Python bytecodes simultaneously [2]. This essentially prohibits thread-parallelism in pure Python code. This does not mean that Python code cannot run in parallel, however. A way around the GIL is to use `mpi4py` [43]. The concept of MPI-parallelization is to launch the program multiple times on separate tasks and communicate messages between tasks (see section 2.1.3).

GPU computing. While some Python libraries support both CPUs and GPUs, `NumPy` and `SciPy` functionality is achieved on GPU by replacing both libraries with `CuPy` [122]. This is developed as a drop-in replacement, designed to require little change except for the import path. We show an example in Listing 2.1. This simple porting with no further optimisation by the user can already result in more than 100× speedup, depending on the operation [51]. Note that `CuPy` not only replicates `NumPy` and `SciPy`, but also provides

access to many GPU specific tools and features such as stream manipulation, NCCL or CUDA graphs (section 2.7).

2.8.2 pySDC

`pySDC` [142] is a Python prototyping code, built for rapid translation of any SDC related research question into suitable code. Its structure is highly modular and object oriented in order to facilitate separation of concerns. The goal is for potential users of the library to assemble preexisting modules into a simulation and swap out individual modules for custom implementations in order to do research on SDC.

The modules that are typically changed in SDC research are the sweeper, determining the precise nature of the SDC iterations, the problem, the inadequately named convergence controllers, which allow to modify the solution or simulation parameters at any point throughout the simulation, or hooks which are used for gathering output. For instance, new preconditioners or novel splitting schemes would be implemented as a sweeper. Our adaptive step size selection algorithms from chapter 3 are implemented as convergence controllers. These have an interface for defining dependencies, which we used to implement the error estimates and step size updates as separate convergence controllers, which are both loaded when the user selects an adaptivity algorithm.

One crucial functionality of `pySDC` that we do not explore here is multi-level SDC [145]. Here, multiple levels of accuracy, for instance using different numbers of collocation nodes or different problem resolution, are set up to solve the collocation problem in a multi-grid fashion. For this purpose, transfer modules exist, which may need modification by researchers interested in multi-level SDC.

Once the user has assembled a simulation from a combination of preexisting and custom modules, they have access to PFASST and various SDC methods as special cases of PFASST. For instance, BGSSDC is single-level PFASST and standard SDC is single-level and single-step PFASST. Even some RKMs are implemented as single-iteration SDC with the Butcher tableau taking the place of the quadrature rule defined by a collocation problem. Any collocation problem can be setup via integration of `qmat` [108], which allows to generate the nodes pertaining to various spectral quadrature rules, the quadrature matrices via integrals of the associated Lagrange polynomials, and various popular preconditioners. When `pySDC` is used to investigate new time-stepping schemes, it can be tested with various already implemented problems spanning a wide range of properties and discretizations.

The code is publicly hosted on GitHub[1], thoroughly documented, and extensively tested using continuous integration.

2.8.3 mpi4py-fft

`mpi4py-fft` [42] is a library for distributed fast Fourier transforms (FFTs) (see section 2.5.2) in Python. Its main function is to provide an interface between `NumPy` and libraries for computing the FFTs with communication infrastructure required for distributed FFTs. While multiple FFT backends are supported, the default choice is `FFTW` (Fastest Fourier Transform in the West) [54], which is a very mature and performant library for FFTs.

Many implementations of the FFT algorithm exist, which differ, for instance, in memory access patterns or radix. `FFTW` separates the transforms into two stages. First, the

[1] `https://github.com/Parallel-in-Time/pySDC`

transforms are planned, which means the fastest FFT implementations are selected for the input data on the current machine. Once this is done, the plan is used an arbitrary number of times to compute the actual transforms very efficiently. The planning stage is expensive because it involves benchmarking different implementations and selecting the fastest one at runtime. Since the planning stage allows to automatically select optimized implementations for different machines, `FFTW` is often competitive with vendor optimized implementations.

What separates `mpi4py-fft` algorithmically from other distributed FFT libraries is the use of MPI ALLTOALLW over ALLTOALLV (see section 2.1.3). Recall that distributed FFTs perform concurrent FFTs over a subset of axes, then "transpose" the data and then perform the remaining FFTs. The transpose step involves all-to-all communication and typically results in a different data layout (see Figure 2.9). As ALLTOALL would only allow to use the same amount of data on each task, many distributed FFT libraries, such as heFFTe [9], use the next more elaborate function ALLTOALLV. However, additional local copy operations are needed with ALLTOALLV, which can be forgone when making use of MPI datatypes and ALLTOALLW. Recall that implementations of ALLTOALLV can be significantly more performant than implementations for ALLTOALLW, such that is not a priori clear whether ALLTOALLV with the additional memory operations or ALLTOALLW are faster on a given machine.

2.9 Supercomputers used in tests

We will later showcase the practical benefit of the methods developed here by means of extensive numerical experiments on massively parallel machines. The supercomputers we use here are all hosted at Forschungszentrum Jülich. We list here the key metrics of each machine as relevant for our discussion.

2.9.1 JUSUF

JUSUF [159] is in the petaflop range and was deployed in 2020 as predominantly a CPU cluster. There are 138 standard compute nodes which are equipped with two AMD EPYC 7742 with 64 cores each and 256 GB of DDR4 memory. The interconnect uses InfiniBand HDR100. While the machine also has GPU accelerated compute nodes equipped with NVIDIA V100, we use exclusively the CPU partitions of JUSUF. See Figure 2.15 for an image of the system.

2.9.2 JURECA

JURECA is a modular system that started operation in 2015. It consists of multiple modules, targeting different types of codes. The module that we use here is JURECA-DC [152], which is a multi-petaflop cluster that was deployed in late 2020. We use the standard CPU partition, which has 480 compute nodes, equipped with two AMD EPYC 7742 processors with 64 cores each and 512 GB of DDR4 memory. The interconnect uses InfiniBand HDR100. See Figure 2.16 for an image of the machine.

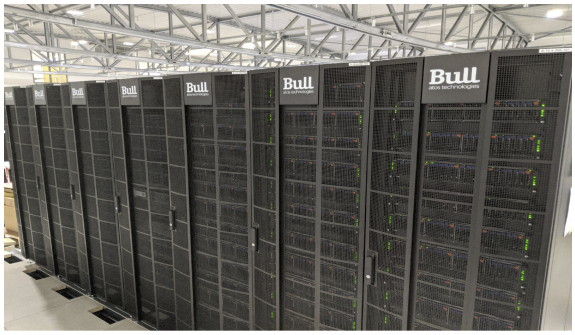


Figure 2.15: The JUSUF supercomputer at Jülich Supercomputing Centre. Image taken from [Figure 1][159] under CC BY 4.0 licence. Copyright: Forschungszentrum Jülich GmbH.

Figure 2.16: The JURECA-DC module at Jülich Supercomputing Centre. Image taken from [Figure 1][152] under CC BY 4.0 licence. Copyright: Forschungszentrum Jülich GmbH / Ralf-Uwe Limbach.

Figure 2.17: The JUWELS booster module (left) and cluster module (right) at Jülich Supercomputing Centre. Image taken from [Figure 1][4] under CC BY 4.0 licence. Copyright: Forschungszentrum Jülich GmbH.

2.9.3 JUWELS

JUWELS [92] is a multi-petaflop CPU machine that was deployed in 2018. The 2271 standard compute nodes are equipped with two Intel Xeon Platinum 8168 CPUs with 24 cores each and 96 GB of DDR4 memory. The interconnect uses two InfiniBand EDR per node.

2.9.4 JUWELS Booster

JUWELS Booster [90] was added as a GPU-centric module to the JUWELS infrastructure in 2020. It entered the world-wide ranking of the Top500 list [153] in place seven in November of 2020. At the time of writing (November 2024), it is in place 33 but still one of the most powerful public machines in Europe. The compute power of the 936 nodes stems predominantly from four 40 GB NVIDIA A100 GPUs per node and is supplemented by two AMD EPIC Rome 7402 CPUs with 24 cores each. Each node is equipped with 512 GB of memory and the interconnect uses two InfiniBand HDR per node. Note that the GPUs within each node are connected by NVLink3, which allows for significantly faster intra-node communication than inter-node. The peak bandwidth per cable is rated at 200 Gbit/s for the InfiniBand and 600 GB/s for the NVLink. See Figure 2.17 for an image of the JUWELS supercomputer.

3 Adaptivity in SDC

In this chapter, we develop generic step-size adaptivity algorithms for SDC. They work analogously to embedded RKM (section 2.4.1), using the same step size update equation but error estimates that are tailored to SDC. We first present an error estimate for fixed iteration number and then an error estimate for converged collocation problems where both step size and iteration number are chosen adaptively. This chapter closely follows [11].

3.1 Δt-adaptivity

We start with a simple algorithm that keeps iteration number k fixed, but chooses step size Δt adaptively. The idea is based on the fact that the order of SDC is usually increased by one in each iteration up to the order of the underlying collocation problem [29], as discussed in section 2.3.4. The increment is therefore the difference between two solutions of different order, which can be used for adaptive step size selection just like in embedded RKM. We show the resulting algorithm including the step size update in Algorithm 4.

Algorithm 4 SDC with Δt-adaptivity

$u^0 \leftarrow u_0$
$k \leftarrow 1$
while $k \leq k_{\max}$ **do**
    $u^k \leftarrow$ SDC iteration applied to u^{k-1}
    $k \leftarrow k + 1$
end while
$\epsilon \leftarrow \|u^{k_{\max}} - u^{k_{\max}-1}\|$
$\Delta t \leftarrow \beta \Delta t \left(\frac{\epsilon_{\text{TOL}}}{\epsilon}\right)^{1/k_{\max}}$
if $\epsilon > \epsilon_{\text{TOL}}$ **then**
    Restart current step with u_0
else
    Move on to next step with $u^{k_{\max}}$
end if

We also use the increment as an error estimate within the PinT algorithms diagonal

SDC and BGSSDC from section 2.3.5. Because most diagonal preconditioners show an increase in the order by one per iteration in practice, the error estimate works exactly the same as in serial SDC.

In BGSSDC, the error estimate also works without modification, but it behaves slightly differently. Recall that the order of accuracy is the same for any number of parallel steps in BGSSDC [57], while the error is increased with increasing number of steps. BGSSDC essentially amounts to solving the entire block of steps with a first order method, then with a second order method, and so on. As the errors of the individual steps accumulate, the increment at the last step gives something akin to a global error within the block. Therefore, we expect that the actual local error of the second to last iteration of the last step within the block is smaller than indicated by the error estimate. The error estimate can be used for adaptive step size selection, nevertheless. One can view the time domain as separated into blocks rather than steps and view the error estimate as the local error of a block within the larger time domain.

Notice that the accumulation of errors of individual steps in the BGSSDC error estimate is actually an advantage. As mentioned, inexactness in the initial conditions increases the error by a problem dependent amount. As this additional error is captured by the error estimate, Δt-adaptivity can be used to make sure that the inexactness does not get out of hand with fixed number of SDC iterations. Since the solution to an individual step will be closer to the initial value if the step size is reduced, the error due to inexactness will decrease with smaller step size, but also increase with larger step size. As the relationship between the step size and this error is not precisely known, we have not found a good way to incorporate it into the step size update equation and resort to using Equation 2.55 also in BGSSDC.

When we find the error estimate to exceed the prescribed tolerance and we need to restart, the rigorous way is to restart from the first step in the block. However, we found heuristically that restarting from the first step in the block where the increment exceeded the tolerance often yields better performance. We outline the restarting procedure using MPI in Algorithm 5.

Algorithm 5 Restarting in BGSSDC

$u^{k_{\max}} \leftarrow$ Local solution after convergence has been declared
$u^{k_{\max}-1} \leftarrow$ Local solution one iteration before convergence has been declared
restart_local $\leftarrow \|u^{k_{\max}} - u^{k_{\max}-1}\|_\infty > \epsilon_{\text{TOL}}$
restart_global $\leftarrow$ MPI ALLGATHER of restart_local
if any(restart_global) **then**
    restart_from $\leftarrow$ Index of first true element in restart_global
    $u^0 \leftarrow$ MPI BROADCAST of u^0 from restart_from
else
    $u^0 \leftarrow$ MPI BROADCAST of $u^{k_{\max}}$ from last step in the block
end if

Note that the order of accuracy of a particular SDC method for a particular problem is best verified numerically when employing Δt-adaptivity, regardless of parallelisation in time. Recall from section 2.3.4 that the order of accuracy after each iteration can often be proven only asymptotically for arbitrary preconditioner. Some preconditioners have

been observed to be unreliable in increasing the order by exactly one per iteration in some situations, e.g. the MIN-SR-NS preconditioner for the Lorenz attractor [25, Fig. 11]. Very stiff problems may suffer from order reduction and require some extra care [82, 98]. The sensitivity of Δt-adaptivity on the order after each iteration also restricts the use of inexact solvers, which have proven useful in SDC [147].

As mentioned in section 2.4, essentially the same adaptive step size selection algorithm has been proposed by van der Houwen and Sommeijer in the context of parallel iterated Runge-Kutta methods [78, 80], which have been shown to be closely related to SDC. Our contribution is to apply this method within the SDC framework, which allows to harness the extensive research on preconditioners and splitting methods, and to use the method within BGSSDC.

3.2 Δt-k-adaptivity

We now turn to a slightly more involved algorithm that removes the requirement to know the order after every iteration. Instead, we iterate until the collocation problem is converged. Because we choose both Δt and k adaptively in this algorithm, we call it Δt-k-adaptivity.

The order of the converged collocation problem is known by the polynomial approximation that defines it. In particular, the order does not depend on how the solution to the collocation problem was obtained, which allows to use inexactness and other SDC variants that are not easily used with Δt-adaptivity. For error estimation, we construct a supplementary solution of lower order by a different polynomial approximation.

The $M+1$ solutions $u_0, u_1, \ldots, u_M$ at the collocation points $\tau = \{\tau_i : i = 0, \ldots, M\}$ define an order M polynomial on $[0, \Delta t]$ that can be evaluated anywhere. Recall that the collocation problem is obtained precisely by integrating the corresponding polynomial approximation of the right hand side evaluations $f_0, f_1, \ldots, f_M$.

Evaluating the polynomial at time t is easily done via barycentric Lagrange interpolation [16], which entails computing an interpolation matrix by computing and evaluating Lagrange polynomials (see Equation 2.24) at t and multiplying by the vector of solutions $\vec{u} = (u_0, ..., u_M)^T$.

To obtain a lower order solution for error estimation, we remove the collocation point τ_{M-1} from the interpolation and use the set of nodes

$$\tau^* = \{\tau_i : i = 0, \ldots, M-2, M\}. \tag{3.1}$$

The resulting polynomial is of degree $M-1$ and we evaluate it at τ_{M-1} to produce an $M-1$-st order accurate approximation of the solution at the removed point. Assuming the u_m^k form a valid polynomial approximation to the continuous solution, i.e. the collocation problem is converged in iteration k, we get

$$u_{M-1}^{(M-1)} = \sum_{i=0}^{M-2} u_i^k l_i^{\tau^*}(\tau_{M-1}) + u_M^k l_{M-1}^{\tau^*}(\tau_{M-1}). \tag{3.2}$$

Here, $l_i^{\tau^*}$ are the Lagrange polynomials of the i-th node in the set τ^*. Note that $l_{M-1}^{\tau^*}$ is associated with with τ_M in the sense that $l_{M-1}^{\tau^*}(\tau_M) = 1$. We can then use the difference

$$\epsilon = \|u_{M-1}^{(M-1)} - u_{M-1}^k\|_\infty, \tag{3.3}$$

as an error estimate of order M, which requires only one interpolation. Since the Lagrange polynomials can be rescaled to the step size easily, we can precompute the interpolation matrix $I_{\tau_{M-1}}$ for the interval $[0, 1]$ and simply multiply it by Δt when estimating the error as in the following equation

$$e = \|\Delta t I_{\tau_{M-1}} \vec{u}^k - u^k_{M-1}\|_\infty, \tag{3.4}$$

$$I_{\tau_{M-1}} = \begin{pmatrix} l_1^{\tau^*/\Delta t}(\tau_{M-1}/\Delta t) \\ l_2^{\tau^*/\Delta t}(\tau_{M-1}/\Delta t) \\ \vdots \\ l_{M-2}^{\tau^*/\Delta t}(\tau_{M-1}/\Delta t) \\ 0 \\ l_{M-1}^{\tau^*/\Delta t}(\tau_{M-1}/\Delta t) \end{pmatrix} \in \mathcal{R}^{1\times M}. \tag{3.5}$$

As SDC is not unconditionally stable for arbitrary preconditioner, in particular when using explicit or IMEX methods, we introduce a relative limit $\gamma = 4$ on how much the step size is allowed to increase and set $k_{\max} = 16$, so that we can return to the last step size that allowed convergence if the desired residual tolerance was not achieved after $k_{\max}$ iterations. Also, we detect instability by an upper limit on the residual and by measuring if the residual increases between iterations. The entire procedure is sketched in Algorithm 6.

Note that we now require adaptive tolerances for both Δt and k. The error estimate for selecting Δt hinges on the u_m forming a useful polynomial approximation to the continuous solution. The quality of this approximation is measured by the residual, but it is also possible to use the increment as an indicator of convergence when viewing SDC as a fixed-point iteration. In practice, one decides ϵ_{TOL} first and then chooses residual or increment tolerance low enough to support the error estimate by solving the collocation problem to sufficient accuracy, but not much lower, which would require additional resources with no benefit to the result. In the experiments in the following chapters, we set the residual tolerance a few orders of magnitude smaller than ϵ_{TOL}.

We selected node τ_{M-1} for the interpolation in the error estimate, but, in tests not documented here, we found the node at which to compute the error can be chosen somewhat arbitrarily. We choose τ_{M-1} because it consistently worked well for various problems and numbers of collocation nodes and leave a more detailed mathematical investigation for future work. Note that for spectral quadrature rules where the last collocation node is not the end point of the interval, we can get an order $M + 1$ error estimate at the end point as opposed to an order M estimate within the interval. We can obtain the order M approximation by extrapolating the u_m to the end point and a higher order approximation by using the collocation update Equation 2.29.

We discussed in section 3.1 that error estimate via the increment has a slightly different interpretation in BGSSDC. The error estimate employed here works entirely the same in BGSSDC as in single-step SDC, on the other hand. This stems from the fact that the solution to the collocation problem remains the same, no matter how many parallel steps are used. Keep in mind that more iterations may be required to achieve an adequately converged solution compared single-step SDC, such that perfect speedup is not necessarily to be expected, even if the error estimate works the same.

Algorithm 6 SDC with Δt-k-adaptivity

$\vec{u}^0 \leftarrow \vec{u}_0$
$k \leftarrow 1$
$r \leftarrow \|\vec{u}_0 + \Delta t Q F(\vec{u}^0) - \vec{u}^0\|_\infty$
$r_{\text{prev}} \leftarrow \infty$
$r_{\max} \leftarrow 10^9$
$no\_convergence \leftarrow False$
while $r > r_{\text{tol}}$ and not $no\_convergence$ **do**
  $\vec{u}^k \leftarrow$ (inexact) SDC iteration applied to $\vec{u}^{k-1}$
  $r \leftarrow \|\vec{u}_0 + \Delta t Q F(\vec{u}^k) - \vec{u}^k\|_\infty$
  if $r > r_{\max}$ or $r > r_{\text{prev}}$ or $k = k_{\max}$ **then**
    $no\_convergence \leftarrow True$
  end if
  $r_{\text{prev}} \leftarrow r$
  $k \leftarrow k + 1$
end while
if $no\_convergence$ **then**
  $\Delta t \leftarrow \Delta t / \gamma$
  Restart current step with u_0
else
  $\tau^* \leftarrow \{\tau_i, i \neq M-1\}$
  $\epsilon \leftarrow \| \sum_{i=0}^{M-2} u_i^k l_i^{\tau^*}(\tau_{M-1}) + u_M^k l_{M-1}^{\tau^*}(\tau_{M-1}) - u_{M-1}^k \|_\infty$
  $\Delta t \leftarrow \max\left(\gamma, \beta \left(\frac{\epsilon_{\text{TOL}}}{\epsilon}\right)^{1/M}\right) \Delta t$
  if $\epsilon > \epsilon_{\text{TOL}}$ **then**
    Restart current step with u_0
  else
    Move on to next step with u^k
  end if
end if

3.3 Mitigating the cost of restarts

Restarting steps that do not meet the accuracy requirements is expensive, but the dense output property of SDC can be used to reduce the cost. Dense output refers to the ability to construct a high-order solution anywhere within the interval after the solution to the step has been obtained. In SDC, this is achieved via barycentric interpolation of the solutions at the collocation nodes, as discussed in section 3.2.

Following a restart, the new step size is smaller than the previous one, $\Delta t_{\text{opt}} < \Delta t$, such that the new collocation nodes all fall within the interval that was just solved, i.e. $\Delta t_{\text{opt}} \tau_i \in [t, t + \Delta t) \, \forall i = 1, \ldots, M$, with normalized collocation nodes $0 \leq \tau_i \leq 1$. Using barycentric interpolation, an initial guess for solving the new interval $(t, t + \Delta t_{\text{opt}}]$ can be generated with

$$u_m^0 = \sum_{j=1}^{M} l_j^{\tau \Delta t}(\Delta t_{\text{opt}} \tau_m) u_j^{-1}, \tag{3.6}$$

where u_m^{-1} is the iterate at collocation node m when the restart was triggered. If the u_m^{-1} are close to the desired solution, this will provide a very good initial guess, which can lead to convergence in few iterations following the restart. This may be the case, for instance, in Δt-k-adaptivity when a restart was triggered despite having reached convergence of the collocation problem. If a restart was triggered in Δt-k-adaptivity because convergence was not reached, there is no indication that the u_m^{-1} are closer to the desired solution than an initial guess generated from the initial conditions such as $u_m^0 = u_0$ and Equation 3.6 should not be used.

In Δt-adaptivity, the number of SDC iterations is constant, such that only the number of iterations of inner solvers, such as a Newton or iterative linear solver, can be reduced by an improved initial guess. On the other hand, the solution to the step after the restart will likely be more accurate than needed due to the good initial guess paired with the normal amount of iterations. The result will be an optimal step size under the assumption that the next step will have access to a similarly good initial guess, which is not available. Therefore, we refrain from using this technique in Δt-adaptivity

4 Improving computational efficiency via adaptivity

Adaptivity is first and foremost a method for boosting computational efficiency. We will demonstrate the capability of the step size selection algorithms in this regard very similarly to [11], with experiments involving the van der Pol problem taken directly from there. We will first verify that the step size selection and error estimate work as expected and then show that this translates to reduced time-to-solution via wall-time measurements. Experiments are performed on JUSUF (section 2.9.1).

4.1 SDC strategies

We benchmark the same four strategies for running SDC against each other as in [11]. We briefly outline them here and collect parameters that we use across all test problems in Table 4.1.

Fixed strategy. SDC can be run with no adaptivity at all, meaning fixed iteration number k and step size Δt, which we call the "fixed" strategy. In all experiments, we select three Gauß-Radau collocation nodes, and $k_{\max} = 5$ iterations, which results in a fifth-order accurate method.

k-adaptivity strategy. We call the scheme of letting Δt fixed, but choosing k adaptively via residual- or increment-based stopping criterion for SDC "k-adaptivity". We use the

	# nodes	node type	$k_{\max}$	accuracy controlling parameter
fixed	3	Gauß-Radau	5	Δt
k-adaptivity	3	Gauß-Radau	99	Δt
Δt-adaptivity	3	Gauß-Radau	5	ε_{TOL}
Δt-k-adaptivity	3	Gauß-Radau	16	ε_{TOL}

Table 4.1: Parameters for SDC that are shared among all problems in computational efficiency experiments for a given strategy. After selection of suitable preconditioner and residual tolerance r_{TOL}, the SDC schemes are all fifth order accurate. We vary the accuracy controlling parameter to record work-precision diagrams in subsequent experiments in this chapter.

	Van der Pol	Lorenz	Gray-Scott	RBC
k-adaptivity	10^{-9}	1.6×10^{-6}	10^{-9}	10^{-2} (10^{-5})
Δt-k-adaptivity	$10^{-5}\epsilon_{\text{TOL}}$	$10^{-4}\epsilon_{\text{TOL}}$	$10^{-4}\epsilon_{\text{TOL}}$	$10^{-4}\epsilon_{\text{TOL}}$ (10^{-5}) $[10^{-2}\epsilon_{\text{TOL}}]$

Table 4.2: Residual tolerances for strategies with adaptive iteration number used in experiments on computational efficiency. The values in round braces denote the increment tolerance, if any is used, and the values in angular braces denote the residual tolerance used in experiments with BGSSDC, if different from the value used in single step SDC.

same collocation problem as in the fixed strategy, thus also expecting order five for all experiments. We control the accuracy in experiments by selecting different fixed step size.

We use exclusively the residual as stopping criterion for all problems except RBC. As discussed in section 2.5.11, the treatment of the boundary conditions in RBC can prevent the SDC residual from reaching arbitrarily small values at finite resolution. We find that a combination of increment and residual based stopping criterion works more reliably for RBC.

Δt-adaptivity strategy. This strategy uses the Δt-adaptivity algorithm (Algorithm 4). Like in the fixed strategy, we use $k_{\max} = 5$ with three Gauß-Radau collocation nodes. We control accuracy in experiments with ϵ_{TOL}.

Δt-k-adaptivity strategy. This strategy uses the Δt-k-adaptivity algorithm from Algorithm 6. Like in the k-adaptivity strategy, we use three Gauß-Radau collocation nodes and terminate using a residual or increment based stopping criterion for RBC and a residual-only based stopping criterion for all other problems. Accuracy is controlled with ϵ_{TOL}.

4.2 Problem setups

We denote the setups for the different problems that we use in experiments on computational efficiency here. Refer to section 2.6 for details on the problems. Note that we select the residual tolerance r_{TOL} depending on the step size, which means a fixed value for k-adaptivity and a value relative to ϵ_{TOL} for Δt-k-adaptivity. We gather the residual tolerances in Table 4.2.

Van der Pol. Here, we set $u_0 = 1.1$ and $u_0' = 0$, $\mu = 1000$ and solve up to $t = 20$. See Figure 2.11 for an illustration of the solution over time. When using Δt-k-adaptivity, we select $r_{\text{TOL}} = 10^{-5}\epsilon_{\text{TOL}}$, and stop the Newton scheme at a tolerance of $10^{-5}r$, with r the current SDC residual, or after a maximum of 9 iterations. When using k-adaptivity, we choose $r_{\text{TOL}} = 1 \times 10^{-9}$.

Lorenz attractor. The solution oscillates around two attractors, as illustrated in Figure 2.12. When using Δt-k-adaptivity, we use $r_{\text{TOL}} = 10^{-4}\epsilon_{\text{TOL}}$. When using k-adaptivity, we choose $r_{\text{TOL}} = 1.6 \times 10^{-6}$.

Gray-Scott. We choose parameters $F = 0.062$ and $k = 0.0609$, which results in a pattern called "U-Skate World" [115]. Here, random rectangles evolve to tube-shaped blobs that

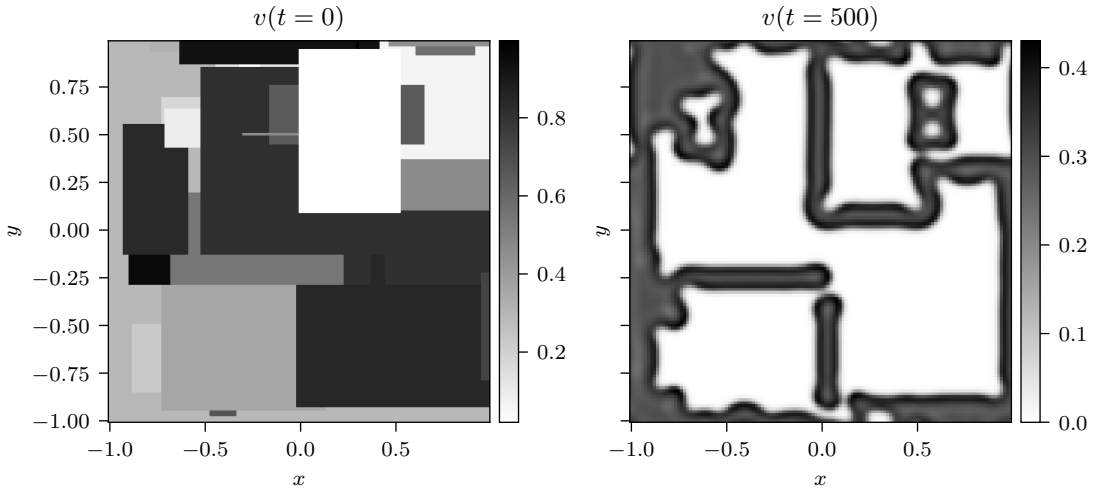


Figure 4.1: Solution of the v component in the Gray-Scott problem with "U-Skate World" configuration used in experiments on computational efficiency. Left: random rectangles as initial conditions. Right: solution after $t = 500$. When the simulation is run for longer, tube-shaped objects will form and travel across the domain. See [10, GrayScott2D.mp4] for a video of the solution over time.

travel across the domain and interact with other blobs. We use resolution of $N = 128^2$, insert 48 rectangles randomly as initial conditions and stop at time $t = 500$. See Figure 4.1 for the initial conditions and the solution at the end of the interval. When using Δt-k-adaptivity, we use $r_{\text{TOL}} = 10^{-4}\epsilon_{\text{TOL}}$. When using k-adaptivity, we choose $r_{\text{TOL}} = 1 \times 10^{-9}$.

Rayleigh Benard convection. We use a resolution of 256×128, a Rayleigh number of $Ra = 3.2 \times 10^5$ and Prandtl number $Pr = 1$. At this Rayleigh number, slight turbulence develops, but the resolution is sufficient for the SDC residual to converge to reasonable values. We start the simulation at $t = 10$, right before macroscopic perturbations emerge due to the random initial perturbation and run until $t = 16$, right before turbulence sets in. See Figure 4.2 for the temperature profile at the beginning and end of the experiments. When using Δt-k-adaptivity, we use $r_{\text{TOL}} = 10^{-4}\epsilon_{\text{TOL}}$ except when running with BGSSDC, where we use $r_{\text{TOL}} = 10^{-2}\epsilon_{\text{TOL}}$ and increment tolerance of 10^{-5}. When using k-adaptivity, we use $r_{\text{TOL}} = 10^{-6}$ or increment tolerance of 10^{-7}.

4.3 Numerical experiments on computational efficiency

We now perform various experiments to showcase the ability to reduce run time by adaptive step size selection while maintaining accuracy. In this section, we use the preconditioners denoted in Table 4.3. They are selected by trial and error, using the ones that perform best for the given example. As of yet, there is no way to know which preconditioner will perform best beforehand.

To start off, we use the van der Pol oscillator as a toy problem to illustrate the benefits of adaptive step size selection in general. This experiment is taken directly from [11]. Figure 4.3 shows the solution (upper panel), local error (middle panel) and computational work, measured in total number of required Newton iterations (lower panel) to solve a van

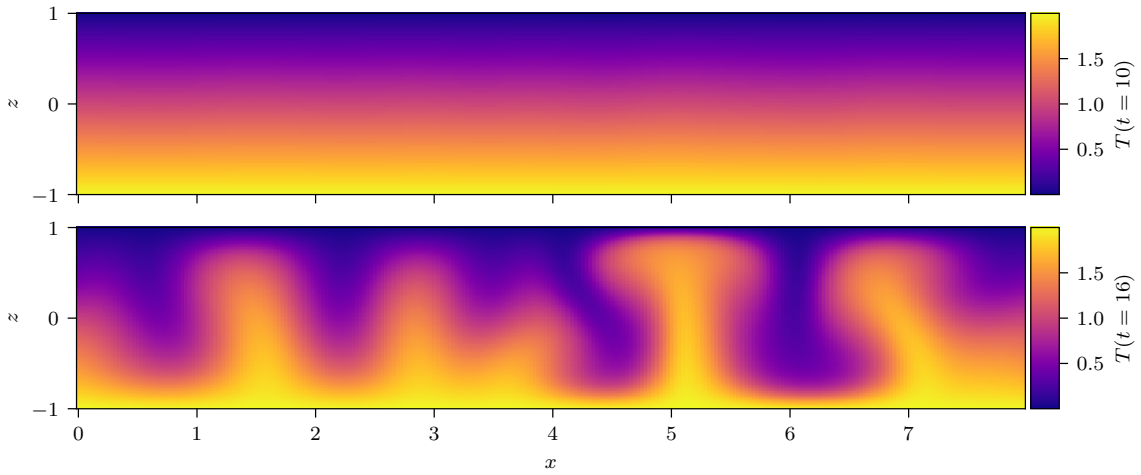


Figure 4.2: Temperature in the Rayleigh-Benard configuration at the beginning and end of experiments for computational efficiency. We start the simulation at $t = 10$, right before the perturbations grow sufficiently to become visible by the naked eye and run until the first plumes start to reach the top of the domain, after which turbulence develops. We chose this time frame because it shows dramatic changes in time-scale throughout the simulation. See [10, RBCLowRayleigh.mp4] for a video of the solution over time.

	Van der Pol	Lorenz	Gray-Scott	RBC
fixed	LU	implicit Euler	MIN-SR-S + Picard	LU + Picard
k-adaptivity	LU	implicit Euler	MIN-SR-S + Picard	LU + Picard
Δt-adaptivity	LU	implicit Euler	MIN-SR-S + Picard	LU + Picard
Δt-k-adaptivity	MIN-SR-S	MIN-SR-NS	MIN-SR-S + Picard	MIN-SR-S + Picard

Table 4.3: Preconditioners used in serial experiments. We selected these because they performed best in our experiments. There is, as of now, no good scheme to choose preconditioners a priori.

	total Newton iterations	relative work	maximum local error
fixed	648,189	71.0	2.027×10^{-05}
k-adaptivity	632,499	69.3	2.010×10^{-05}
Δt-k-adaptivity	12,146	1.3	2.508×10^{-05}
Δt-adaptivity	9,124	1.0	2.639×10^{-05}

Table 4.4: Results from Figure 4.3 in aggregate numbers. The relative work is denoted relative to Δt-adaptivity, which is the most efficient strategy here.

	expected	Van der Pol	Lorenz	Gray-Scott	RBC
fixed	5	4.7	4.3	4.7	4.4
k-adaptivity	5	4.3	4.8	4.3	3.2
Δt-adaptivity	1	1.1	1.0	1.0	1.3
Δt-k-adaptivity	1.25	1.0	1.6	1.1	1.6

Table 4.5: Exponent of the scaling between the parameter controlling the accuracy (Δt or ϵ_{TOL}) and observed accuracy in terms of global error. The values are obtained from the experiments shown in Figure 4.4 by fitting a power law $f(x) \propto x^{-p}$, where the table contains the values inferred for p and x is either Δt or ϵ_{TOL}. Note that the values are not particularly accurate due to small sample size, which also prevents reasonable statistical analysis.

der Pol problem with very large value of $\mu = 1000$, which leads to very fast transitions followed by long periods of slow change. See Table 4.4 for the total number of Newton iterations and maximal local error obtained when solving the interval with the different strategies. Resolving the transitions requires a very small step size due to the high stiffness of the problem, while a much larger step size suffices in between transitions. We select parameters such that the maximal local error during the transition is on the order of 10^{-5}. For fixed step size schemes, the resulting step size is so small, that we solve to machine precision outside of the transition. The associated computational effort to cover this short simulation time is approximately 70 times of what is required by either of the step size adaptive strategies to meet the same accuracy requirements. Solving an entire period of the oscillator with fixed step size is prohibitively expensive, even when choosing k adaptively. In tests not shown here, we found similar, although less pronounced, trends for smaller values of μ.

This experiment shows that the step size can be tuned to the requirements of the problem in a much broader range and more precisely compared to the iteration number. Note that the accuracy of the converged collocation problem is still determined by the step size. Also, the step size can be continuously adjusted in the floating point range, whereas the iteration number is an integer.

Confirming efficacy of error estimate and step size update. We now test that error estimates and step size update equations work as expected for all problems by measuring how the global error responds to changing ϵ_{TOL}. We show the result, including how fixed step size schemes react to changing Δt in Figure 4.4 and Table 4.5.

All methods we use here are fifth order accurate, which is the expected scaling with the step size for strategies with fixed step size. With adaptive step size, we select the step size by controlling the local error of a lower order method, see Equation 2.54 and Equation 2.55. The resulting scaling for the global error is $e \propto \epsilon_{\text{TOL}}^{q/p+1}$, with p and q as in

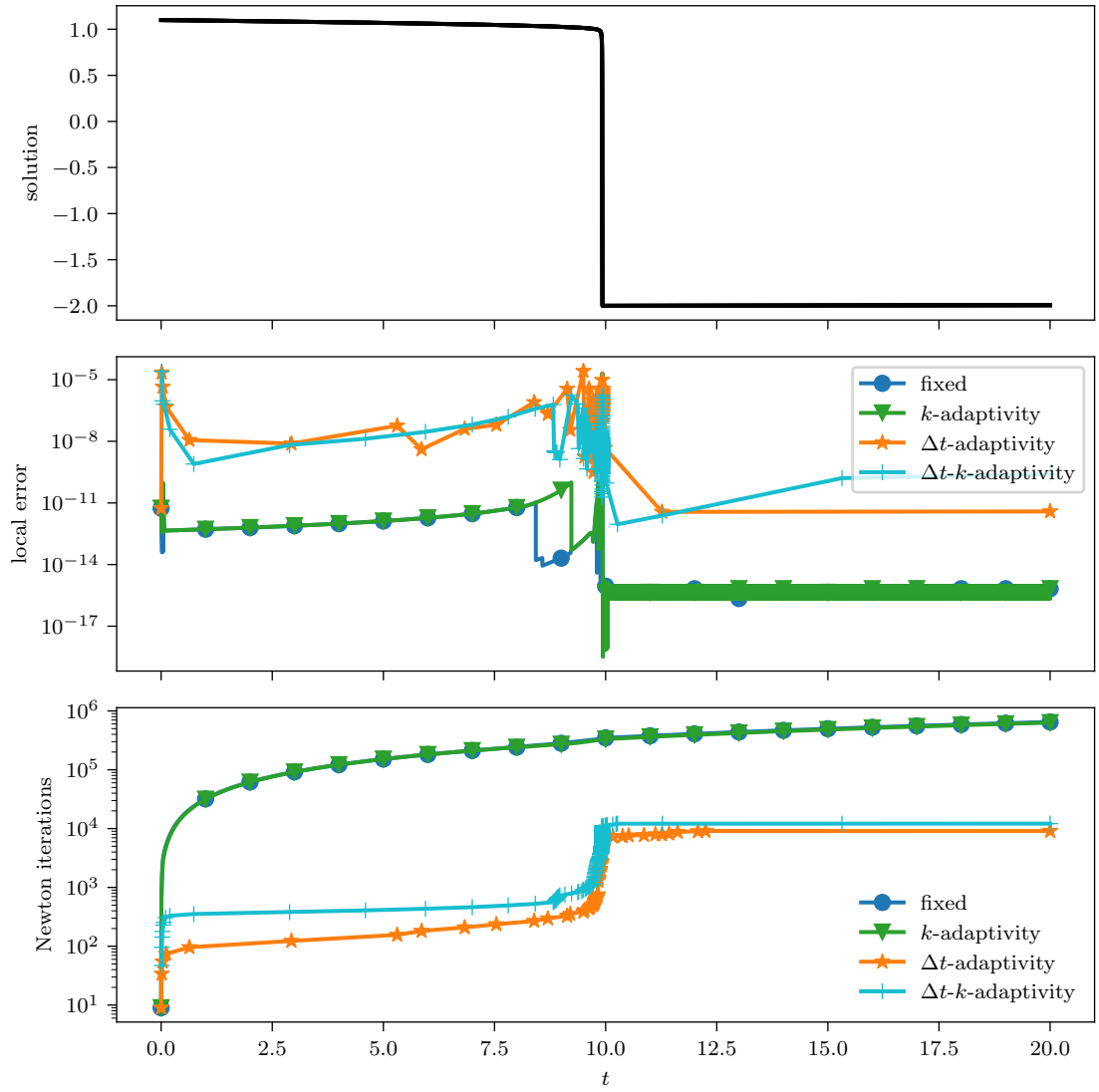


Figure 4.3: SDC for the van der Pol problem with $\mu = 1000$, solved with four different strategies. The top panel shows the solution, the middle panel shows the resolution via the local error compared to a reference solution and the bottom panel shows how many Newton iterations are needed to reach the respective time, which is a good indicator of computational cost. Only one fast transition is shown, which requires very small step sizes due to extreme stiffness. The solution is periodic with the next transition appearing at around $t = 600$ (see Figure 2.11). Each marker represents a single step for methods with adaptive Δt or 10000 for the fixed or k-adaptive methods. See Table 4.4 for the total number of Newton iterations and maximum local error obtained with all strategies in this experiment. Figure is reproduced from [11, Fig. 6] under CC BY 4.0 licence.

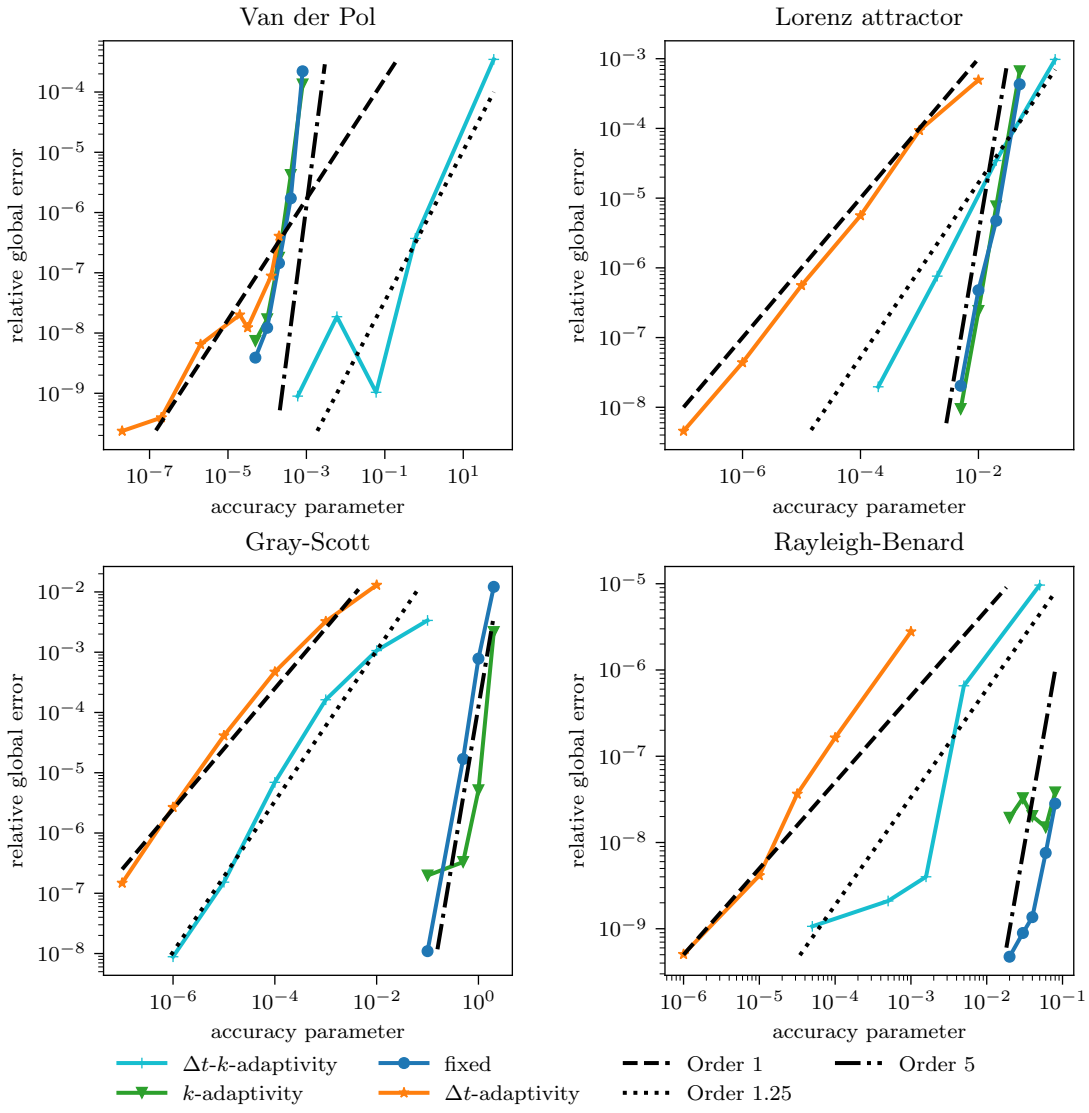


Figure 4.4: Plot of the parameter controlling the accuracy versus the achieved accuracy. The parameter is Δt for the k-adaptivity and fixed strategies, and ϵ_{TOL} for Δt-k-adaptivity and Δt-adaptivity. The collocation problem is fifth order accurate in all cases and we expect this scaling for fixed and k-adaptivity strategies. For Δt-adaptivity, we expect linear dependence between parameter and error, while we expect $e \propto \epsilon_{\text{TOL}}^{5/4}$ for Δt-k-adaptivity. We observe the expected scaling to a reasonable degree for all problems. The top left panel is reproduced from [11, Fig. 8] under CC BY 4.0 licence.

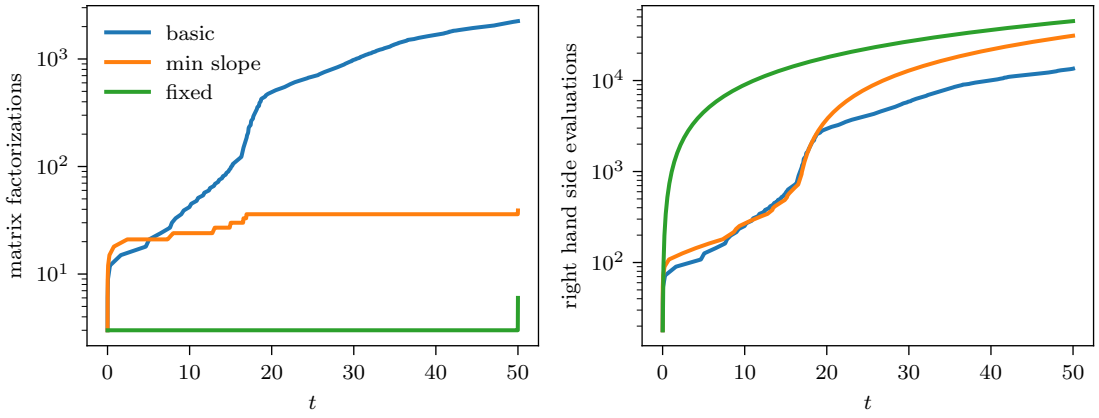


Figure 4.5: Number of expensive operations over simulation time for a run of RBC with three step size controllers. Left is number of LU decompositions in the implicit solves and right is the number of right hand side evaluations. "fixed" is with fixed step size, where factorizations are required only in the beginning and in the end, when the step size is adjusted to reach the final time. The step size is chosen as the minimum of what the "basic" controller uses, which is Δt-adaptivity with the step size update from Equation 2.55. "min slope" only changes the step size when $\|\Delta t_{\text{opt}}/\Delta t - 1\| > 1$ or when the step is restarted and has safety factor $\beta = 0.5$. Note that in other experiments in this chapter we only solve the interval from $t = 10$ to $t = 16$.

Equation 2.54. That means we expect a linear dependence of global error on step size for Δt-adaptivity for any q, since we always have $q = p + 1$. For Δt-k-adaptivity, on the other hand, the scaling depends on the choice of quadrature; in our case the result is $e \propto \epsilon_{\text{TOL}}^{5/4}$, as $q = 5$ and $p = 3$.

We observe this scaling in practice to a reasonable degree. Some deviations due to stiffness or stability restrictions are not unexpected and occur also in embedded RKM. In RBC with k-adaptivity, we have to select very small step sizes due to stability restrictions, which means we cannot observe the order for reasonable residual tolerance.

Step size controller for Rayleigh-Benard convection. As mentioned in section 2.4.3, selecting the largest possible step size is not necessarily the most efficient strategy. In the RBC problem, we solve the linear systems using LU factorizations, which can be reused in subsequent time steps, provided the step size did not change. As the factorizations are usually the most computationally expensive part, these should be kept to a minimum and it may pay off to keep a smaller step size rather than increase it. This comes at the cost of more right hand side evaluations, however. Therefore, we aim to balance factorizations and right hand side evaluations by employing a bespoke step size controller.

The step size controller we use in RBC only changes the step size when $\|\Delta t_{\text{opt}}/\Delta t - 1\| > 1$, or if the step is restarted, and we use safety factor $\beta = 0.5$ when computing Δt_{opt}. The step size is therefore only reduced if the step is restarted and only increased when the optimal step size is more than twice as large as the current one. This, together with the small safety factor helps prevent restarts when operating close to the stability limit of the IMEX scheme.

The resulting computational work with this step size controller compared to the basic one (Equation 2.55) in an example using Δt-adaptivity is shown in Figure 4.5. Keeping the

	matrix factorizations	right hand side evaluations	wall time / s
basic	72	432	126
min slope	12	504	79
fixed	3	5400	664

Table 4.6: Number of operations and wall time needed to simulate the interval $t \in [10, 16]$ with RBC with the step size controllers from Figure 4.5. This is the interval that we cover in subsequent experiments on computational efficiency with RBC. The "min slope" step size controller requires marginally more right hand side evaluations than the basic controller, but significantly fewer matrix factorizations. It requires a few more matrix factorizations than the fixed step size controller, but dramatically fewer right hand side evaluations. This makes the min slope step size controller the most efficient one we considered.

step size constant at a small value requires the least possible amount of matrix factorizations, but a lot of right hand side evaluations. Changing the step size between every step with the basic controller, on the other hand, requires a lot of factorizations, but the least possible amount of right hand side evaluations, given the accuracy requirements. The step size controller that we use is a compromise between the two that uses much fewer matrix factorizations than the basic controller, but also much fewer right hand side evaluations than with using fixed step size, and is the fastest to run as shown in Table 4.6. Because the step size controller avoids many restarts when the step size is close to the stability limit compared to the basic controller, it needs only marginally more right hand side evaluations than the basic controller up to $t \approx 20$. From then on, the step size controller does not increase the step size, leading to more right hand side evaluations compared to the basic controller.

Note that the optimal step size controller depends on the cost of matrix factorizations and right hand side evaluations, as well as the dynamics of the problem. Generally speaking, the more expensive the matrix factorizations are compared to using the factorizations and evaluating the right hand side, the larger the threshold for changing the step size should be and vice versa. Limiting step size changes via step size controllers based on low-pass filters or PID controllers [140] is common practice, e.g. [38, 84, 131]. These could provide even more efficient schemes for the RBC example and should be explored in future research.

For the other problems, we use the basic step size controller that simply updates $\Delta t = \Delta t_{\text{opt}}$ and therefore selects the largest possible step size that fulfills the accuracy requirements. In the Gray-Scott example, this is sensible as the computational cost of the linear solves is independent of the step size. While more elaborate step size controllers may be able to reduce the number of Newton iterations in the ODE examples slightly, the difference in computational cost between implicit solves and right hand side evaluations is much less pronounced compared to RBC. Therefore, we do not expect significant reduction in computational work by custom step size controller.

Work-precision diagrams. Next, we present work-precision diagrams for all problems showing that adaptive step size selection results in actually decreased wall-time for a range of parameters in Figure 4.6. We see that it is entirely unpractical to solve problems like the van der Pol example with such stark contrast in time-scales with fixed step size. For the other problems, the benefits are less pronounced, but adaptive step size selection is

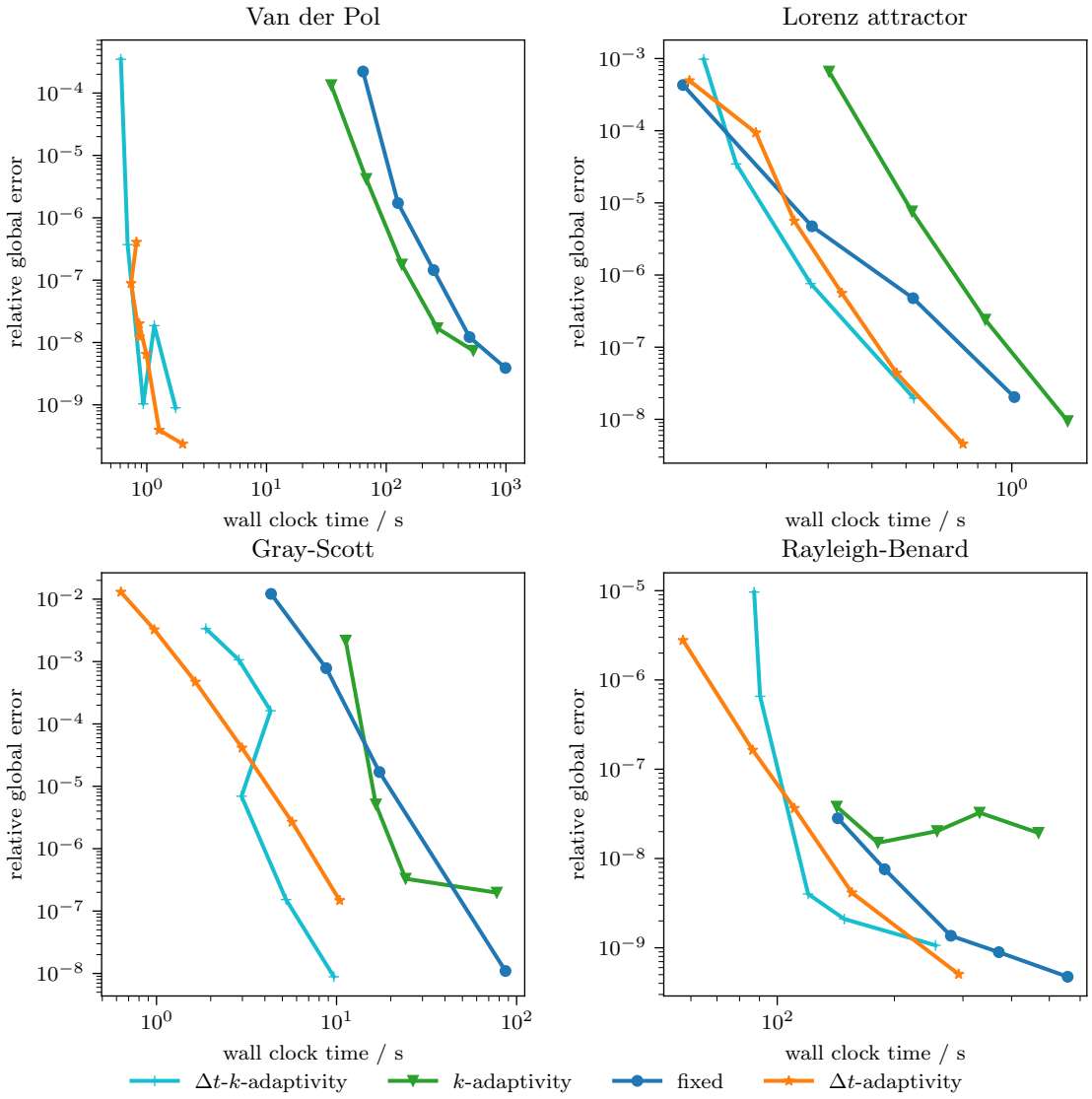


Figure 4.6: Work precision diagram with the four SDC strategies for all problems from section 2.6. With the van der Pol oscillator at $\mu = 1000$, the benefit from adaptive step size selection is especially pronounced, but all problems benefit from adaptive step size selection. Even RBC, where additional matrix factorizations are performed when the step size is changed, is solved more efficiently with adaptive step size selection. The top left panel is reproduced from [11, Fig. 7] under CC BY 4.0 licence.

	Van der Pol	Lorenz	Gray-Scott	RBC
Δt-adapt. N=4x1	implicit Euler	implicit Euler	IMEX Euler	LU + Picard
Δt-adapt. N=1x3	MIN-SR-S	MIN-SR-S	MIN-SR-S + Pic	MIN-SR-S + Pic
Δt-adapt. N=4x3	MIN-SR-S	MIN-SR-S	MIN-SR-S + Pic	MIN-SR-S + Pic
Δt-k-adapt. N=4x1	MIN-SR-S	MIN-SR-NS	-	MIN-SR-S + Pic
Δt-k-adapt. N=1x3	MIN-SR-S	MIN-SR-NS	MIN-SR-S + Pic	MIN-SR-S + Pic
Δt-k-adapt. N=4x3	MIN-SR-S	MIN-SR-S	-	MIN-SR-S + Pic

Table 4.7: Preconditioners used in parallel experiments. They were again selected by trial and error. "Pic" is short for Picard here.

always the most efficient scheme.

A reduction in the accuracy controlling parameter (Δt or ϵ_{TOL}) leads to a fairly predictable increase in computational cost with all strategies except Δt-k-adaptivity. This is because the collocation problem generally converges faster for smaller step size, meaning that the work per step is often reduced at the same time that the number of steps is increased. This makes Δt-k-adaptivity especially suitable when high accuracy is desired. Similar effects with Newton solvers converging faster for smaller step size have led to the design of special step size controllers [49].

We do not find that k-adaptivity is always better than the fixed scheme. In the case of the Lorenz attractor, the computation of the residual adds substantial computational cost that outweighs any savings by adaptivity. In the RBC example, we have to select a very small step size to prevent instability in the IMEX solver, but we cannot select a very small residual tolerance because the perturbations due to the boundary conditions are non-zero (see section 2.5.11).

4.4 Parallel-in-time speedup

We now attempt to accelerate adaptive SDC by the parallel-in-time methods discussed in section 2.3.5, namely the parallel-across-the-method algorithm diagonal SDC, the parallel-across-the-steps algorithm Block Gauß-Seidel SDC (BGSSDC) and a combination of both. Note that we cannot compute speedup as a number because the accuracy changes as the number of steps is increased in BGSSDC or the preconditioner is changed for diagonal SDC. Instead, speedup is inferred from comparing lines in work-precision diagrams.

The efficiency of SDC relies crucially on the preconditioners, and this is only exacerbated with BGSSDC. See Table 4.7 for the preconditioners we selected for parallel experiments. Like in the single-step case, the optimal preconditioner is unknown beforehand and intuitions from single-step SDC may not hold. For instance, one may expect LU to work well for stiff problems, but Gray-Scott is a counter example that worked significantly better with the IMEX Euler combination. In particular, the combination of BGSSDC and diagonal SDC, which requires a diagonal preconditioner, did not converge within a reasonable time frame relative to the other methods at all for Gray-Scott.

Note that, as PFASST (see section 2.3.5) also shows order k after k iterations [57], if it converges at all and up to the order of the underlying collocation problem, and as the algorithmic structure is comparable to BGSSDC, it is conceivable to use the adaptive step size selection algorithms also in PFASST. We did not make experiments with step size adaptive PFASST, since this exceeds the scope of this thesis, but this would certainly be

	Van der Pol	Lorenz	Gray-Scott	RBC
speedup	1.43	1.27	1.83	2.79
parallel efficiency / %	48	42	61	93

Table 4.8: Speedup and parallel efficiency for Δt-k-adaptivity accelerated by diagonal SDC. We can compute speedup as a number here because the serial and parallel methods have exactly the same result.

interesting for future work.

PinT extensions with Δt-adaptivity. See Figure 4.7 for a work-precision diagram with the different modes of parallelism added on top of the Δt-adaptivity strategy.

We find decent speedup with BGSSDC for the van der Pol and Gray-Scott problems and modest speedup for the Lorenz example. In RBC, however, we are unable to obtain speedup with BGSSDC. How much speedup one gets depends on how much the added inaccuracy in the initial conditions in early iterations impacts convergence. If the added inaccuracy is still present in later iterations, the error will be estimated larger and the step size selected smaller, leading to additional work for similar accuracy compared to single step SDC.

We find good speedup with diagonal SDC in Δt-adaptivity for all problems except van der Pol. Here, it seems the LU preconditioner performs significantly better than diagonal ones.

The only problem where the combination of both PinT methods performs well and better than serial is Lorenz. For van der Pol and Gray-Scott, it performed worse than the serial run and for RBC it performed about the same as the serial run. Given that it requires twelve ranks in time, parallel efficiency is low.

PinT extensions with Δt-k-adaptivity. We show a work-precision diagram with PinT versions of SDC and Δt-k-adaptivity in Figure 4.8. Because the diagonal preconditioners performed well in serial in this strategy, we obtain the same result with diagonal SDC as in serial. This enables the computation of speedup and parallel efficiency, which we show in Table 4.8. The largest speedup of approximately 2.8 is achieved with RBC, which is a parallel efficiency of 93 %. With Gray-Scott, we also get decent parallel efficiency with speedup of 1.8, while the ODE examples profit relatively less and gain only mild speedup. Diagonal SDC parallelizes the implicit solves and the right hand side evaluations, but adds some communication, mainly in computing the residual and the solution at the end of the step. The greater the share of the implicit solves and right hand side evaluations of the overall runtime, the better the parallel efficiency with diagonal SDC.

With Δt-k-adaptivity, we find no speedup with BGSSDC with any problem. The composite collocation problem requires more iterations to converge than each single collocation problem, to the point that the added work negates any benefit from parallelization.

The combination of BGSSDC and diagonal SDC was not competitive for any problem we tested. Note that for the Lorenz example, we found best performance with the MIN-SR-S preconditioner with BGSSDC, and MIN-SR-NS without, further highlighting the mystery around preconditioners in BGSSDC.

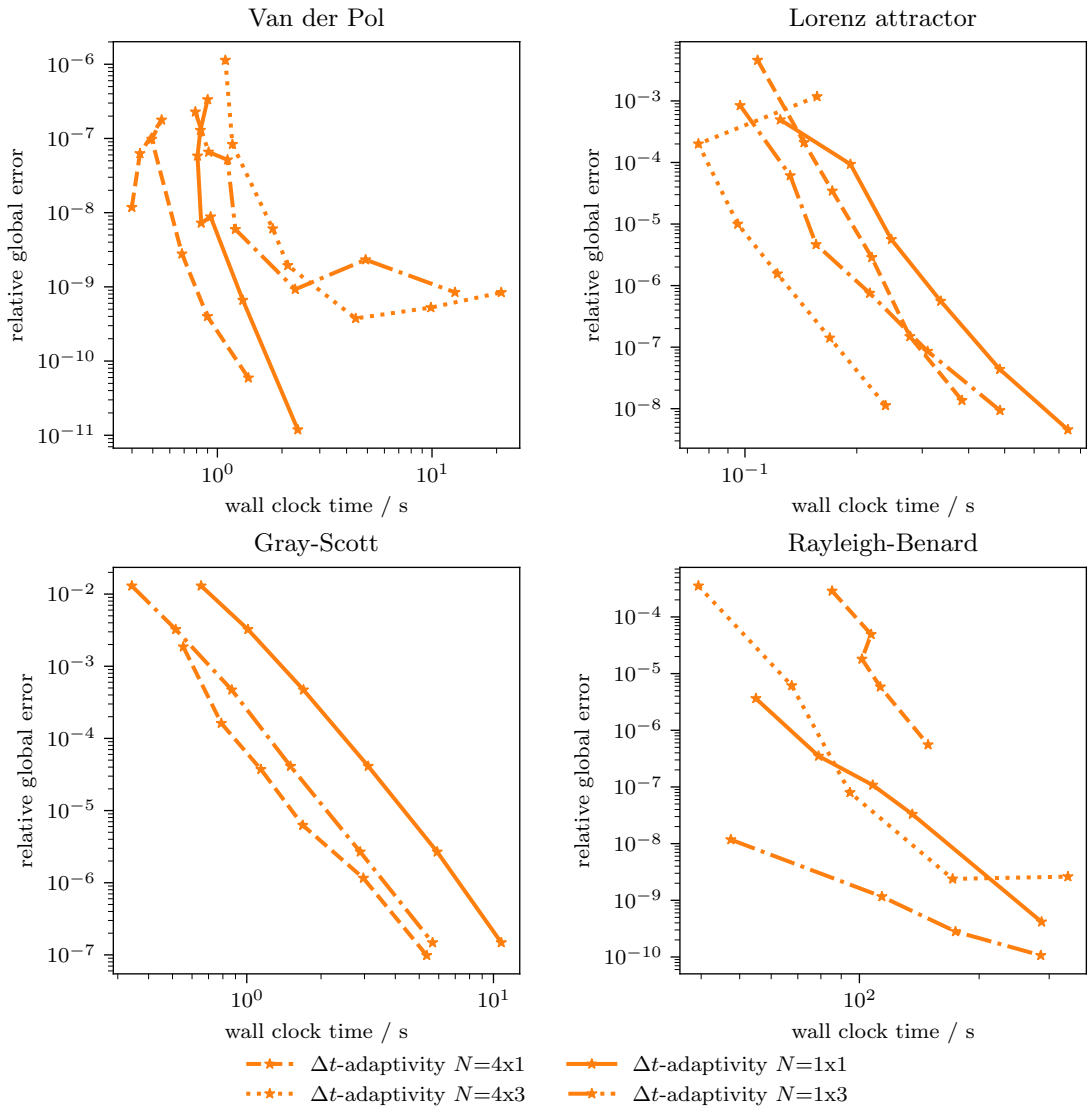


Figure 4.7: Work-precision diagram for parallel-in-time extensions with Δt-adaptivity. If more than one processor is used, it is indicated as N=<processes across the steps>x<processes across the method>, meaning N = 4x1 is BGSSDC with four parallel steps, N = 1x3 is diagonal SDC with three processes and N = 4x3 is a combination of both. We find good speedup for all problems with diagonal SDC except van der Pol and good speedup for all problems but RBC with BGSSDC. The combination of both, on the other hand, does not work well for any problem except the Lorenz attractor. We do not show the combination of diagonal SDC and BGSSDC for Gray-Scott because it did not reach the final time for any tolerance within 100 s.

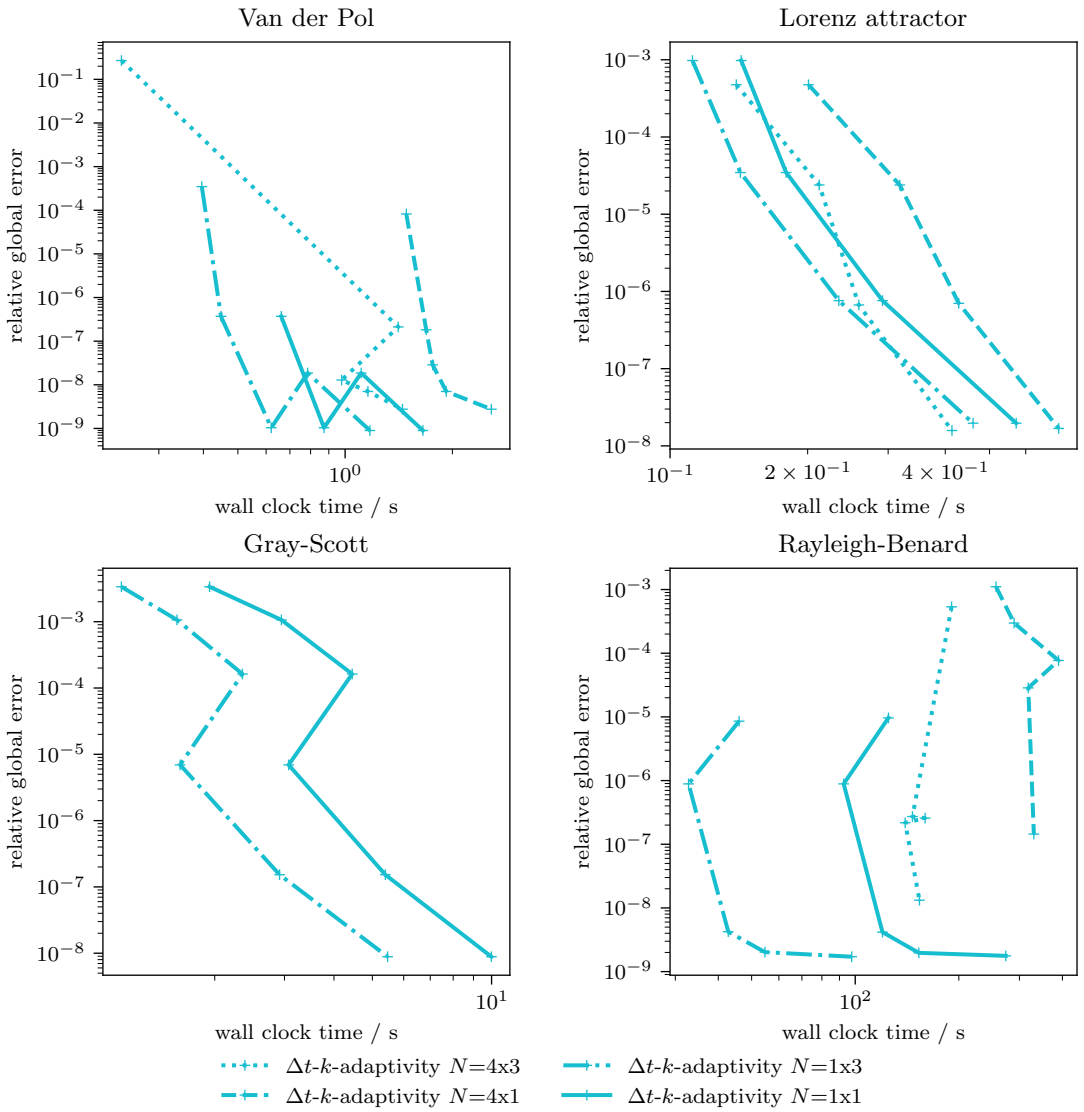


Figure 4.8: Work-precision diagram for parallel-in-time extensions with Δt-k-adaptivity. The legend is analogous to Figure 4.7. Diagonal SDC gives excellent speedup for the PDEs and decent speedup for the ODEs. BGSSDC does not work well for any problem, except for Lorenz when it is combined with diagonal SDC and even then delivers poor parallel efficiency. Gray-Scott with BGSSDC did not finish within 100 s for any tolerance.

4.5 Comparison against embedded Runge-Kutta methods

We compare against state-of-the-art embedded RKM here. The problems and their discretizations differ enough to require different reference RKM for each. We start by noting the reference methods we used, which we believe to be among the closest competitors to the SDC schemes from the pool of RKM that are readily available in the literature.

Reference methods. Because Lorenz is not very stiff, an explicit RKM is actually most efficient. We use Cash-Karp's method [28], which is a six-stage embedded explicit method of orders five and four. We refer to this method as CK5(4). We do not show this method for the other problems because it did not give results within a reasonable time frame and was not competitive at all. We also test a singly diagonally implicit embedded method with explicit first stage of orders five and three [88], which we call ESDIRK5(3), on Lorenz and van der Pol.

For Gray-Scott, we need an additive RKM, specifically an IMEX RKM, in order to employ the same IMEX solver as in SDC. Recall that IMEX RKM are a pair of explicit and implicit methods. Since embedded RKM are also pairs of RKM, IMEX embedded RKM are four distinct coupled methods. The method we use is an additive pair of a singly diagonally implicit stiffly accurate L-stable embedded method of orders five and four and an explicit embedded method of orders five and four [89, 5(4)8L[2]SA$_2$], which we call ARK5(4).

RBC poses additional challenges for RKM because of the algebraic incompressibility constraint, which is solved in place of an evolution equation for the pressure. The solutions corresponding to all of the stages satisfy the algebraic constraint because the IMEX solver (Equation 2.117) respects it, but the correct pressure cannot be obtained by linear combination of the stages, as in the collocation update. The simplest solution is to use methods where the solution is used in one of the stages, as is the case in stiffly accurate RKM and SDC with quadrature rules that include the right endpoint [6]. Recall that stiff accuracy means the last line in the matrix is equal to the weights $A_{si} = b_i$.

While stiff accuracy is a common property in RKM, we require an IMEX RKM for our RBC implementation. There are already few IMEX RKM of high order available [89] and we were unable to procure a method of order higher than four where both implicit and explicit methods are stiffly accurate. Recall that when using the same weights for implicit and explicit methods, the number of IMEX coupling conditions reduces significantly [126], but an SDIRK and an explicit RKM cannot both be stiffly accurate and share the weights. Instead, we tested a four-stage third order method [7, Section 2.8], where the implicit method is singly diagonally implicit and has explicit first stage, and a seven-stage fourth order method [105, RK.4.A.2], where the implicit method has two distinct non-zero values on the diagonal.

The number of distinct non-zero values on the diagonal dictates how many matrix factorizations are required per step. In particular, singly diagonally implicit methods, where all non-zero values on the diagonal are the same, necessitate only a single factorization per step. The number of stages dictates how often the right hand side needs to be evaluated during a single step. In tests not shown here, we found the third order method to be more computationally efficient than the fourth order method because it needs half the number of matrix factorizations and little more than half the number of right hand side evaluations

per step. Therefore, we compare third order accurate settings of SDC against the third order accurate RKM for RBC.

Note that none of the stiffly accurate IMEX RKM methods we found are embedded methods. Therefore, we use an alternative adaptive step size selection mechanism based on the CFL limit [40]

$$\Delta t = c \min(\frac{u}{\Delta x}, \frac{v}{\Delta z}), \tag{4.1}$$

where Δx and Δz are the grid spacing in x and z direction respectively and $0 < c \leq 1$ is called the CFL number. The CFL limit is a stability criterion for explicit and IMEX methods. In our experiments, the CFL number also takes the role of the accuracy parameter, which is to say we select smaller c when we desire higher accuracy and vice versa.

Computing the CFL limit is rather expensive because the ratio of velocity to grid spacing has to be computed locally, i.e. in physical space, thus requiring an extra pair of transforms. As shown in Figure 4.5, a step size controller that prevents small step size changes is critical for performance. With the CFL limit, we are not as free to ignore step size reduction because there is no mechanism to check the accuracy and restart. Therefore, we change the step size already when it changes by more than 20 %.

Numerical results. We show work-precision diagrams comparing step size adaptive SDC against the reference RKM in Figure 4.9. We use parallel-in-time SDC as suitable, selecting the best performing method from Figure 4.7 and Figure 4.8. Note that we found time-serial SDC with LU preconditioner to outperform diagonal SDC with MIN-SR-S preconditioner for RBC with Δt-adaptivity for the third order configuration here.

We find that SDC is more efficient than the reference RKM for all problems but the Lorenz attractor, where the explicit RKM performs best. We find the largest differences for the PDE examples with high accuracy, where Δt-k-adaptivity is very efficient. In RBC, we find lower global error with SDC at the same average step size compared to the RKM. Note that with diagonal SDC, only one matrix factorization is required per step per process, giving similar advantages as *singly* diagonally implicit RKM.

As discussed above, we selected the ARK3 method as a reference RKM for the RBC problem because it was the best performing RKM that satisfied our requirements. While third order adaptive SDC already outperforms this method in Figure 4.9, we compare separately against the best performing fifth order adaptive SDC methods from Figure 4.7 and Figure 4.8 in Figure 4.10. We find that fifth order adaptive SDC performs significantly better than the third order RKM, even serially in time. When using diagonal SDC with three processes and Δt-k-adaptivity, we find up to five orders of magnitude smaller error at the same run-time. Recall that the easy tuning of the order, and the ease with which high order solutions can be obtained in general, is one of the major advantages of SDC. In this case, we increase the order simply by adding one collocation node in Δt-k-adaptivity or adding one node and increasing the number of iterations by two in Δt-adaptivity. While we did not find a suitable stiffly-accurate reference RKM, this does not mean RKM cannot be used to obtain fifth-order solutions. The ARK5(4) method we use for Gray-Scott can be used, if the pressure is computed separately after the differential equations have been integrated, e.g. using a projection scheme [33]. We leave comparison of high order SDC to high order RKM with a projection scheme for future work. Finally, we note that this is a good example for a problem where high-order solutions are easily obtained with SDC

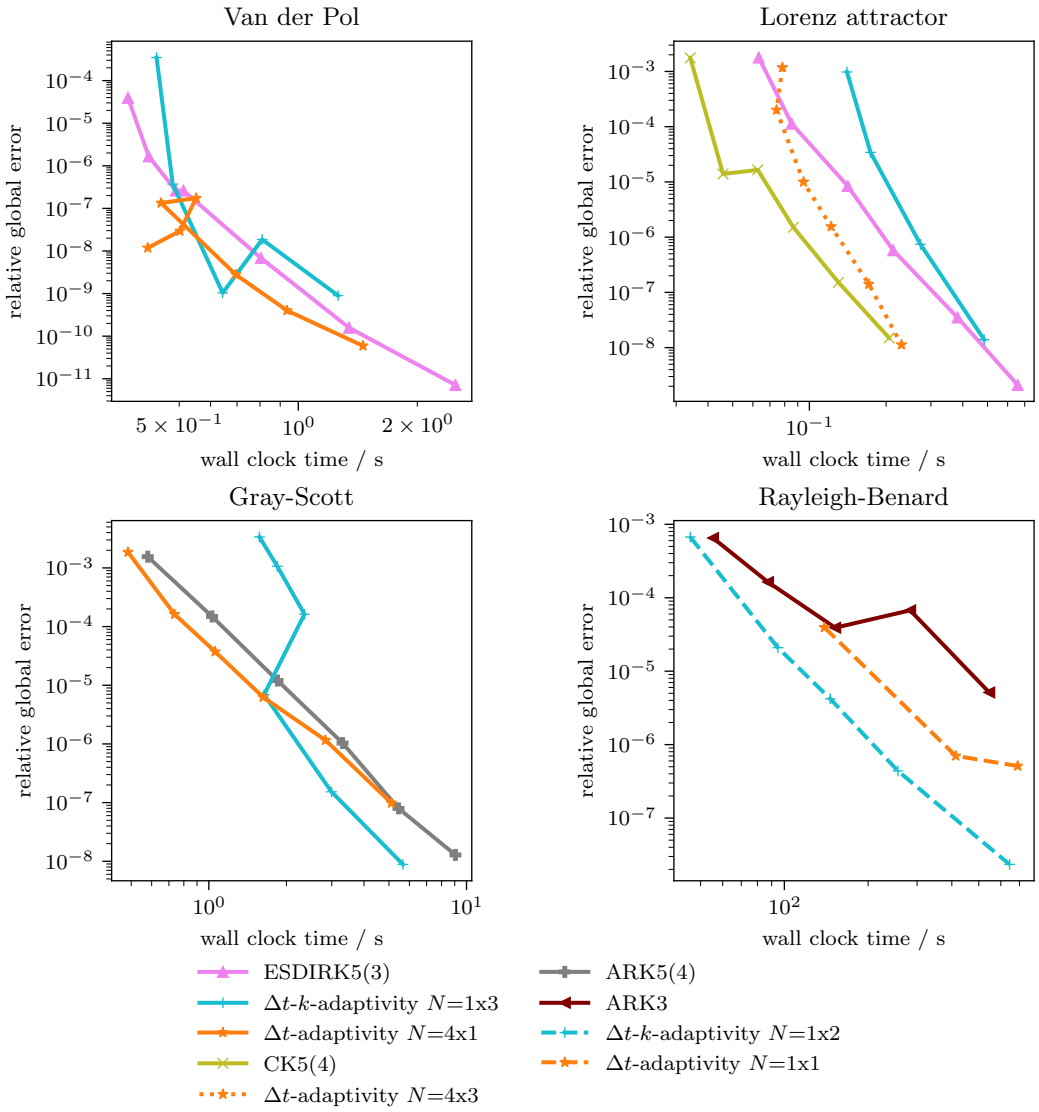


Figure 4.9: Comparison of step size adaptive time-parallel SDC against reference RKM. As the Lorenz attractor problem is only mildly stiff, the explicit Cash-Karp's method (CK5(4)) is most efficient. For all other problems, SDC outperforms the RKM. Note that we use step size adaptive, fifth order methods for all problems but RBC, where all methods are third order accurate and the step size for the RKM is determined via CFL limit. For third order Δt-adaptivity with RBC, time-serial SDC with LU preconditioner performed better than either parallelization strategy. See Figure 4.10 for comparison of ARK3 against fifth order SDC for the RBC example. The top left panel is reproduced from [11, Fig. 10] under CC BY 4.0 licence.

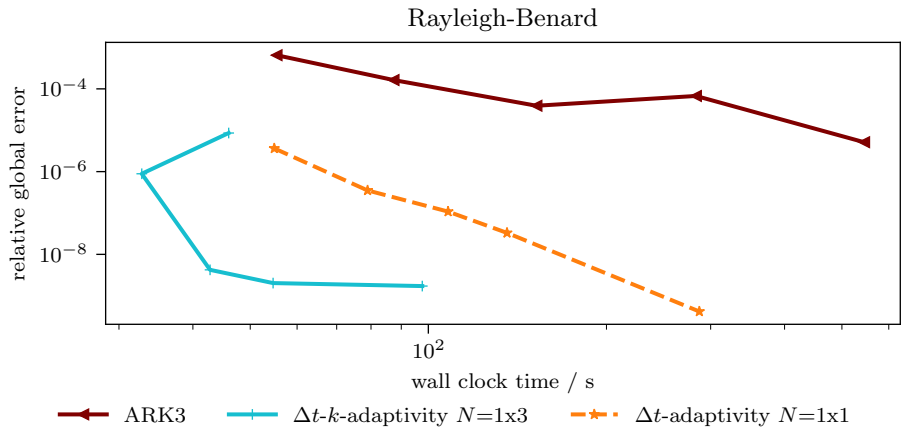


Figure 4.10: Comparison of third order additive RKM versus fifth order SDC methods for the RBC problem. In Figure 4.9, we compared the same RKM against third order SDC methods. Here, we show that the easy tuning of the order in SDC can give even larger performance benefits over RKM.

while substantially more effort is required for implementing a high-order RKM.

5 Resilience against soft faults by means of adaptivity

After having shown that adaptive step size selection can be employed for boosting computational efficiency in chapter 4, we now show that it improves resilience against soft faults as well. This chapter closely follows [12], with experiments involving the Lorenz and RBC problems taken directly from there.

5.1 Resilience strategies

We test the resilience capabilities of all SDC strategies from section 4.1, as well as the dedicated resilience strategy Hot Rod, for which we develop an SDC specific version. We discuss only properties with respect to resilience here and refer to section 4.1 for more information. We again use fifth order accurate methods throughout. See Table 4.1, Table 4.2 and Table 4.3 for simulation parameters.

fixed strategy. The fixed strategy has no particular mechanism for mitigating faults. It serves as a baseline for determining the impact of faults in absence of any resilience strategy.

k-adaptivity strategy. We expect the SDC residual to reflect faults, which means k-adaptivity will automatically attempt to correct the fault by continued iteration. The range of faults that SDC with k-adaptivity can recover from thusly depends on the preconditioner and the problem. A plain Newton solver in a non-linear problem, for instance, may not converge from a faulty initial guess. Similarly, convergence is not always guaranteed in SDC. Still, for a wide range of faults, k-adaptive SDC has been shown to provide an adequate solution in the presence of faults [63].

Δt-adaptivity strategy. In Δt-adaptivity, we expect to detect the fault in the estimate of the local error that is part of the adaptive step size selection algorithm. Recovery happens by rejection of steps with local error exceeding the target. This has already been shown to be an effective resilience strategy for the embedded RKM [13] that this algorithm is based on.

Δt-k-adaptivity strategy. In Δt-k-adaptivity, we combine the resilience capabilities of Δt-adaptivity and k-adaptivity. Faults can be detected in the residual as well as the local estimate and corrected by continued iteration as well as restarts.

Hot Rod. Hot Rod [65] is a dedicated resilience strategy that we use for reference. It works by estimating a quantity derived from the local error and restarting if it is unexpectedly large. It was developed for explicit Runge-Kutta methods, but we present an adaptation that can be used with implicit and explicit SDC at any order in the following. In our experiments, we add Hot Rod on top of the fixed strategy with identical SDC parameters, except that we perform one extra SDC iteration with no gain in accuracy, as discussed later.

5.2 Adapting Hot Rod to SDC

We start by describing the method for the explicit Cash-Karp's method [28] of orders four and five for which it was developed in [65] and then discuss the modifications needed to use it with SDC.

Hot Rod for explicit RKM. The local error is estimated by subtracting two separate error estimates to obtain one estimate of higher accuracy. The first error estimate is the same as used in embedded RKM, that is to say the stages of the RKM are assembled with two different weights to obtain solutions of different order. The difference is an error estimate of the lower order method, which is order four in the case of Cash-Karp's method. See section 2.4.1 for more details on embedded RKM.

The second estimate is based on Radau's quadrature and Taylor expansion. Similarly as in embedded methods, a secondary solution is computed and compared to the solution of the integrator. In this case, the secondary solution is obtained by extrapolation from solutions and right hand side evaluations at previous time steps. The specific formula used in [65] for orders $p \leq 5$ is

$$\begin{aligned}\mathcal{R} = &\frac{\Delta t}{10}\left[3f\left(t_{n-2}, u_{n-2}\right) + 6f\left(t_{n-1}, u_{n-1}\right) + f\left(t_n, u_n\right)\right] \\ &+ \frac{1}{30}\left[u_{n-3} + 18u_{n-2} - 9u_{n-1} - 10u_n\right],\end{aligned} \tag{5.1}$$

where the index n marks the current time step and $n - j$ indicates that the quantity is associated with j steps prior. Note that, in order for the two error estimates to be of the same order, we have to use the fourth order solutions in Equation 5.1. Even though Cash-Karp's method is order five accurate, we have to advance in time with the fourth order accurate secondary solution.

The error estimate used for fault detection is then obtained as

$$\Delta = \epsilon_{\text{embedded}} - \mathcal{R}, \tag{5.2}$$

with $\epsilon_{\text{embedded}}$ the error estimate from the embedded method. Faults are detected if

$$\Delta > \Delta_{\text{TOL}}, \tag{5.3}$$

with Δ_{TOL} the user defined threshold for error detection. Detection of a fault then triggers a restart of the step.

Hot Rod in SDC. To use Hot Rod in SDC, we need to adapt both error estimates. For $\epsilon_{\text{embedded}}$ we use the increment, which behaves analogously, as discussed in section 3.1. The second estimate is slightly more involved. In principle, Equation 5.1 can be used directly for orders lower than five. However, as one of the virtues of SDC is the easy tuning of the order, we compute coefficients for comparable error estimates at runtime to the desired order.

The Taylor expansion of the solution around t with step h is

$$u(t-h) = u(t) - hu'(t) + \frac{h^2}{2}u''(t) + ... = \sum_{i=0}^{k} \frac{(-h)^i}{i!}u^{(i)}(t) + \mathcal{O}(h^{k+1}). \tag{5.4}$$

For autonomous problems $u_t = f(u)$, we also have the Taylor expansion of $-hf(t)$

$$-hf(t-h) = -hu'(t) + h^2u''(t) + ... = \sum_{i=1}^{k} \frac{(-h)^i}{(i-1)!}u^{(i)}(t) + \mathcal{O}(h^{k+1}). \tag{5.5}$$

We now seek a linear combination of n solutions and right hand side evaluations at previous time steps $t - h_j$ that cancel all terms but the solution at the current time to order q

$$u(t) = \sum_{j=1}^{n} a_j u(t-h_j) + b_j h_j f\left(u\left(t-h_j\right)\right) + \mathcal{O}(\bar{h}^{q+1}), \tag{5.6}$$

where $\bar{h}$ should be understood as the average step size. To this end, we solve the $(q+1) \times (2n+1)$ system of equations

$$\begin{pmatrix} 1 & 0 \\ \frac{(-h)_j^i}{i!} & \frac{(-h)_j^i}{(i-1)!} \end{pmatrix} \begin{pmatrix} a_j \\ b_j \end{pmatrix} = \begin{pmatrix} \delta_{j0}a_0 \\ 0 \end{pmatrix}, \tag{5.7}$$

where i should be understood as a row index, j as a column index and δ is the Kronecker delta. In order for the system to be invertible, we require $n = q/2$.

The coefficients resulting from solving Equation 5.7 essentially define a linear multi-step method (LMM) [72, Chapter III.1] for obtaining the secondary solution. However, the solutions entered into the extrapolation, Equation 5.6, carry local truncation errors, which accumulate within the LMM.

In order to compare to $\epsilon_{\text{embedded}}$, we need to extract only one times the local truncation error from the error estimate $\mathcal{R}$. To this end, we linearize and make the approximation that $u(t-h_n)$ is exact and the local error accumulates linearly with $u(t-h_{n+i})$ carrying i times the local truncation error of the method. We then compute the local truncation error estimate as

$$\begin{aligned} \mathcal{R}^{(q)} &= \mathcal{P}\left|\sum_{j=0}^{n} a_j u(t-h_j) + b_j h_j f\left(u\left(t-h_j\right)\right)\right|_\infty + \mathcal{O}(\bar{h}^{k+1}), \\ \mathcal{P} &= 1/\left|\sum_{j=0}^{n-1}\left(n-j\right)a_j\right|, \end{aligned} \tag{5.8}$$

which assumes the step size to be constant, i.e. $h_j = j\Delta t$. The more general prefactor for adaptive step sizes is

$$\mathcal{P}_{\text{adaptive}} = 1/\| \sum_{j=0}^{n-1} \left(\sum_{i=0}^{n-j-1} \left(\frac{\Delta t_i}{\Delta t_0} \right)^{k+1} \right) a_j \|, \tag{5.9}$$

but we use Hot Rod only with fixed step size here.

Note that Hot Rod requires a startup period. $\mathcal{R}^{(q)}$ can only be computed after $q/2$ time steps have been completed. Before that, Hot Rod cannot detect faults.

Overhead. There are two sources of overhead in Hot Rod. First, additional computation is needed because the error estimates need to have the same order. As mentioned before, this requires to advance in time with a solution of the same accuracy as the one we estimate the accuracy of in the embedded method / increment. In the SDC version, this means we perform one extra iteration with no gain in accuracy.

Second, Hot Rod increases the memory requirements immensely. For an order q method in SDC, we require at least q solution size objects in memory made up of solutions at collocation nodes and right hand side evaluations. Computing the extrapolated solution in $\mathcal{R}^{(q)}$ requires q additional solution size objects in memory, doubling the memory footprint.

Detection thresholds. Hot Rod requires the user to set a tolerance for error detection Δ_{TOL}. When considering adaptivity as a resilience scheme, the tolerance that controls accuracy is a natural choice for controlling resilience as well. Note that there are no false positives with adaptive methods because steps are restarted or additional iterations are performed only whenever the accuracy is not sufficient for any reason, including faults.

In Hot Rod, on the other hand, the tolerance for resilience is not tied to the accuracy of the problem. The user must choose a threshold that balances sufficiently low false negative rate against sufficiently low false positive rate. In [65], methods for choosing thresholds during runtime based on machine learning are proposed. We resort to choosing the lowest threshold that yields 0 % false positive rate in our tests for all examples but van der Pol. Due to larger changes in time-scale, we select a threshold in van der Pol that balances false positives with false negatives.

If a false positive is triggered in Hot Rod because the accuracy of the SDC scheme was insufficient to satisfy the tolerance, restarting the step will yield the same result and another false positive will be triggered. In order to break this circle, we attempt to correct faults in Hot Rod via restart only once and simply ignore the fault detector in the second try of the same step. In practice, allowing multiple restarts could be needed if the fault rate is sufficiently high.

5.3 Fault insertion methodology

While a realistic fault insertion simulation would consider faults in instructions at the lowest level, we only insert faults by flipping bits in the solution. We do this by converting to binary IEEE 754 representation, flipping the bit and then converting back to floating point representation. Evidence suggests that this is a sensible strategy to investigate silent data corruption, while lower level fault injection mainly increases the probability of

	Van der Pol	Lorenz	Gray-Scott	RBC
Δt (fixed)	0.045	10^{-3}	0.4	0.05
Δt (k-adaptivity)	0.04	8×10^{-3}	1.0	0.07
r_{TOL} (k-adaptivity)	10^{-7}	10^{-9}	10^{-9}	10^{-6} (10^{-7})
ϵ_{TOL} (Δt-adaptivity)	2×10^{-7}	10^{-7}	2×10^{-6}	10^{-4}
ϵ_{TOL} (Δt-k-adaptivity)	5×10^{-4}	2×10^{-4}	10^{-4}	5×10^{-7} (10^{-5})
r_{TOL} (Δt-k-adaptivity)	4×10^{-8}	10^{-11}	4×10^{-7}	5×10^{-3}
t_{fault}	5.25	10	100	20.2

Table 5.1: Parameters specific to resilience experiments. Values in braces in the r_{TOL} rows denote increment tolerances, which we use only with the RBC problem. The values were chosen by trial and error to obtain similar accuracy with all strategies. t_{fault} is the simulation time at which we insert faults.

termination of the program by the operating system [160]. Also, we restrict the discussion to transient bit flips. This means we assume that the fault does not occur again if we repeat the operation.

A fault can be inserted in a variety of ways. We can insert in any iteration, at any collocation node, in any space position, bit, and at any time. We pick a single time for fault insertion up front in order to limit the number of options. We then insert faults at random in solution variables in up to 4000 experiments for the Gray-Scott and RBC examples. For the van der Pol and Lorenz attractor problems, there are only 2560 and 3840 combinations for inserting faults at a fixed time because there are 5 options for iteration number, 4 choices for collocation node (including initial conditions), 2 and 3 solution components, each with 64 bits.

Since we want the same faults for all strategies, we restrict the random insertion to faults that would target the fixed strategy in the same way. That is, when we perform a fixed number of $k_{\max}$ iterations in the fixed and Δt-adaptivity strategies, we insert faults only in the first $k_{\max}$ iterations in all strategies, even if the k- and Δt-k-adaptivity strategies may perform more iterations in this step.

5.4 Problem setups

Here, we note the setups for the problems from section 2.6 that we use in experiments regarding resilience. As we perform thousands of experiments here, some configurations from experiments on computational efficiency are not viable. We denote only the differences to the configurations from section 4.2 and refer there (Table 4.1, Table 4.2, and Table 4.3) for more details. We collect SDC parameters specific to resilience experiments in Table 5.1.

Van der Pol oscillator. The van der Pol configuration used in experiments on computational efficiency is entirely unsuitable for fixed step size schemes due to the extreme changes in time scales. We therefore choose a less stiff configuration with $\mu = 5$ for tests on resilience. We use initial conditions $u_0 = 2$ and $u_0' = 0$ and solve up to $t = 11.5$, which covers one full oscillation, as shown in Figure 5.1. We insert faults at $t = 5.25$.

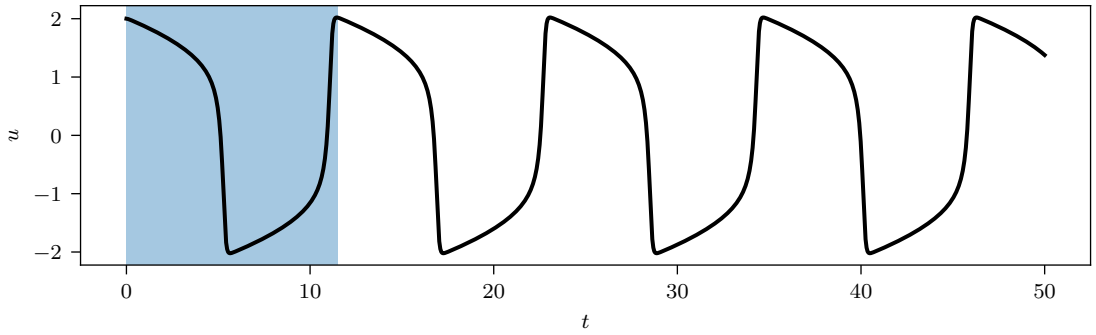


Figure 5.1: Solution of the van der Pol configuration used in tests on resilience over time. We solve only the first oscillation, shown shaded here, in experiments.

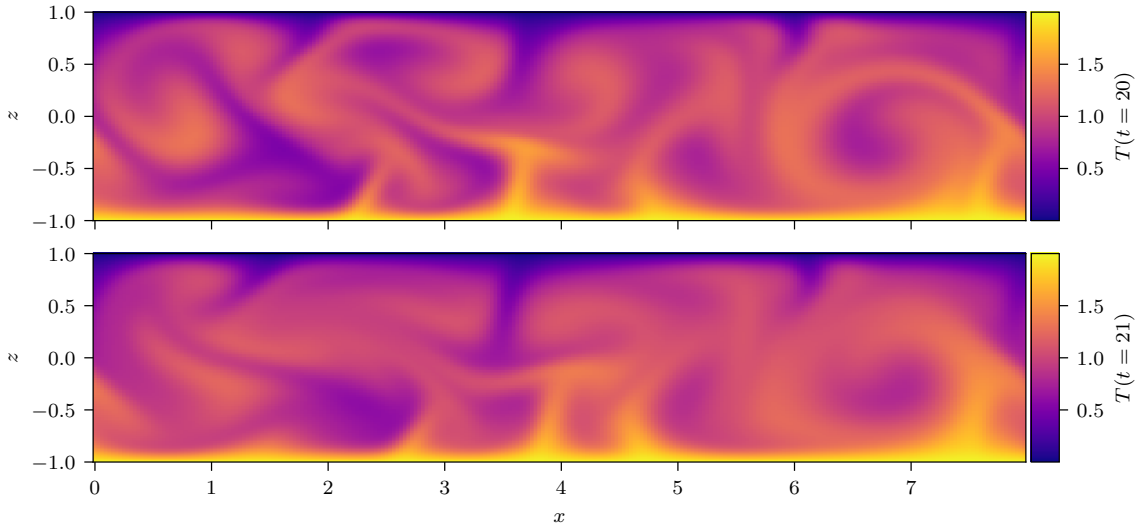


Figure 5.2: Temperature profile at the beginning and end of a Rayleigh-Benard resilience experiment. Here, we consider the interval between $t = 20$, where turbulence is already under way and $t = 21$, when the simulation is still in the same regime. See [10, RBCLowRayleigh.mp4] for a video of the solution over time. The bottom panel is reproduced from [12, Fig. 12].

Lorenz attractor. We use exactly the same configuration as in experiments on computational efficiency (chapter 4) here. See section 4.2 for details. We insert faults at $t = 10$.

Gray-Scott. We use exactly the same configuration as in experiments on computational efficiency (chapter 4) here. See section 4.2 for details. We insert faults at $t = 100$.

Rayleigh-Benard. Here, we only change the time domain of the simulation. We start at $t = 20$, when turbulence has just emerged and continue to $t = 21$. See Figure 5.2 for the temperature profile at the beginning and end of the experiments. We insert faults at $t = 20.2$.

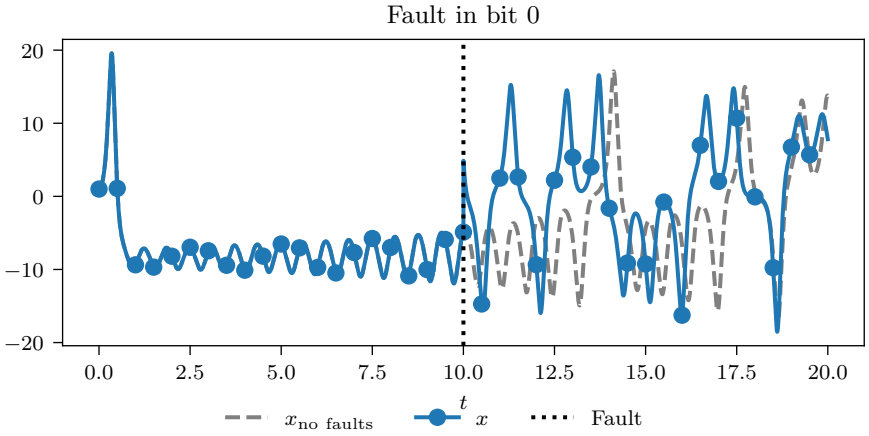


Figure 5.3: Horizontal coordinate of the solution of the Lorenz problem with bit 0 flipped in x. The dashed line is the solution in absence of faults, while the solid line shows the response of the fixed strategy to the fault. Bit 0 stores the sign, which is flipped following the fault. This dramatically changes the dynamics. Figure reproduced from [12, Fig. 1].

5.5 Numerical results

We now investigate the response to faults of the SDC strategies from section 5.1. Before presenting how the strategies perform for all problems from section 2.6, we investigate them in detail with the Lorenz attractor problem. Because of its chaotic behaviour, this problem is ideally suited to put resilience strategies to the test. The impact of faults is very significant, as demonstrated in Figure 5.3 and Figure 5.4. In the former, a significant fault is inserted, which leads to an entirely different trajectory afterwards. In the latter, the solution is perturbed invisibly to the naked eye, but the perturbation grows over time to eventually lead to a vastly different outcome.

Determining recovery rate. We select a recovery threshold that controls how much the global error is allowed to be increased relative to a fault-free run and compute the recovery rate as the ratio of experiments where the global error did not increase beyond the threshold to total number of experiments. The threshold should be set according to the requirements on the simulation and there is no one-size-fits-all solution. We show the recovery rate across all faults for different thresholds in Figure 5.5 and settle on a value of 1.1, which means we count successful recovery if the increase of the global error is no greater than 10 %. That is to say, we accept a fault as recovered if the global error is not increased by more than 10 %.

Recovery rate by node. Figure 5.6 shows recovery rates dependent on the collocation node that is targeted by the fault. In the `pySDC` implementation, collocation node 0 stores the initial conditions. This means faults that occur in collocation node 0 cannot be recovered by restarting, since the initial conditions will continue to include the fault. This causes significantly lower recovery rate for faults in node 0 than in other nodes.

We also find that the fixed strategy has slightly lower recovery rate in node 3 than in nodes 1 and 2. After SDC has converged, the solution to the step is computed as a

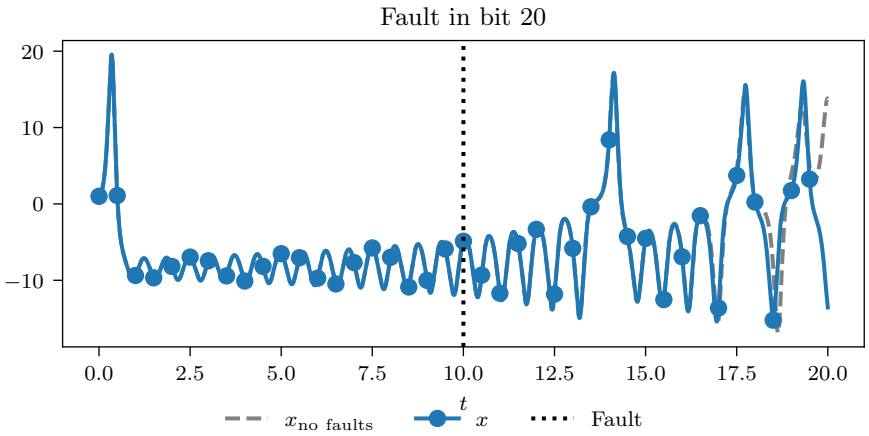


Figure 5.4: Horizontal coordinate of the solution of the Lorenz problem with bit 20 flipped in x. The legend is shared with Figure 5.3. Bit 20 is the eighth bit in the mantissa, meaning it changes the solution only a little. The impact of the fault is invisible to the naked eye in the beginning, but the chaotic nature of the problem amplifies it, such that the solutions are significantly different at the end of the interval. Figure reproduced from [12, Fig. 2].

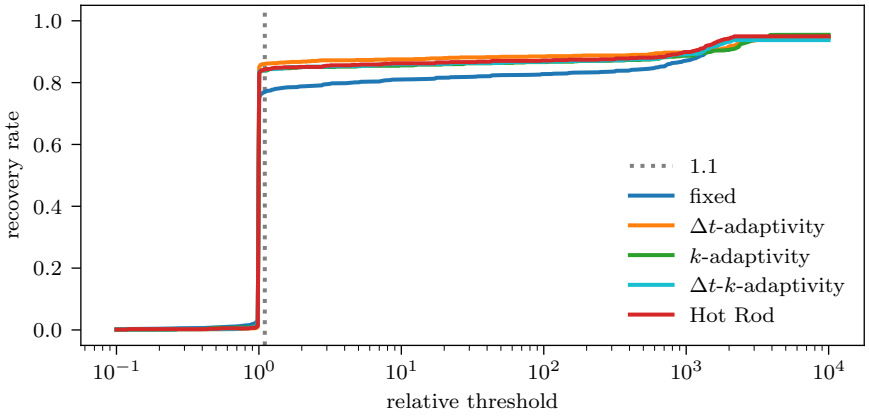


Figure 5.5: Recovery rate for the Lorenz attractor problem depending on the acceptance threshold. The threshold is shown as the ratio of the global error to the global error of a fault-free run, meaning a value of one requires fault correction to deliver the same or lower error as a simulation without faults. A value of two would count twice the error as without faults as recovered. Any threshold larger than one leads to high recovery rates for all recovery strategies. We choose a threshold of 1.1 for computing recovery rates in subsequent figures. Figure reproduced from [12, Fig. 3].

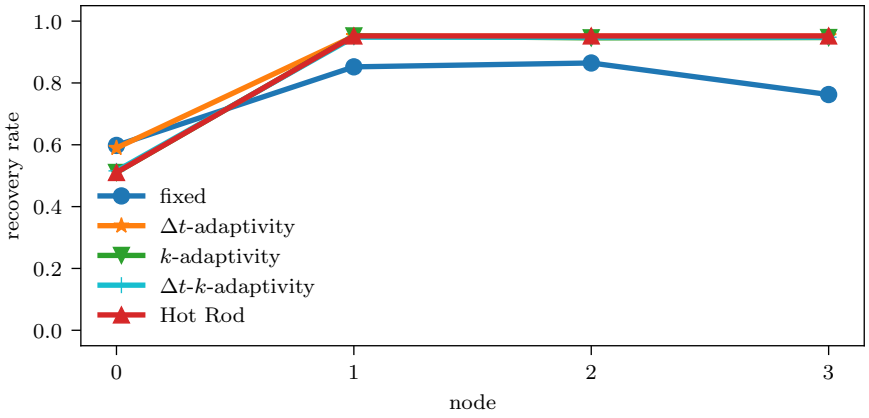


Figure 5.6: Recovery rate for the Lorenz attractor problem depending on the collocation node that was hit by a fault. As node 0 stores the initial conditions in the `pySDC` implementation, faults targeting it can not be recovered by restarting, which leads to decreased recovery rate compared to other nodes. Figure reproduced from [12, Fig. 4].

weighted sum of the u_m during the collocation update. With the Gauß-Radau quadrature rule with three nodes that we use here, the last node is the end point of the interval and the collocation update amounts to simply copying u_3. Hence, when a fault targets any but the last node in the last iteration, it has no impact on the solution to the step.

Recovery rate by iteration. Figure 5.7 shows recovery rates depending on the iteration in which the fault occurs. Since faults in the last iteration and any but the last node do not affect the collocation update, the fixed and Δt-adaptivity strategies have the highest recovery rate in the last iteration. Because the fault nevertheless impacts the residual, strategies with adaptive k may continue to iterate and propagate the fault into the collocation update. Keep in mind that Hot Rod performs one extra iteration, but advances in time with the solution of the fifth iteration. If a fault occurs in the initial conditions in the fifth iteration, Hot Rod will trigger a restart and the fault ends up affecting the solution. Otherwise, the recovery rate of the fixed strategy decreases with iteration number. The earlier the fault occurs, the more iterations are available to smooth out the perturbation.

Recovery rate by bit. Finally, we look at the recovery rate depending on which bit was flipped in Figure 5.8. In IEEE 754 double representation, bit 0 stores the sign, bits 1 to 12 store the exponent and the remaining bits store the mantissa. We find that flipping bits beyond 35 perturbs the solution too little to be noticeable in this problem.

When flipping bits 2, 3, or 4, overflow errors in the Newton solver crash the code, resulting in 0 % recovery rate for all strategies. For k-adaptivity, Δt-k-adaptivity and Hot Rod strategies, we observe that 75 % of faults to bits 0, 1, and 9 to 28 are fixed. This is because all faults that target these bits and not the initial conditions are fixed, while all faults that target these bits and the initial conditions are not fixed. At 80 %, the recovery rate for Δt-adaptivity is slightly higher in these bits because also faults that occur in the initial conditions and in the last iteration are fixed.

After having identified the faults that cannot be recovered by these strategies as faults that target the initial conditions and faults that cause overflows, we can exclude these

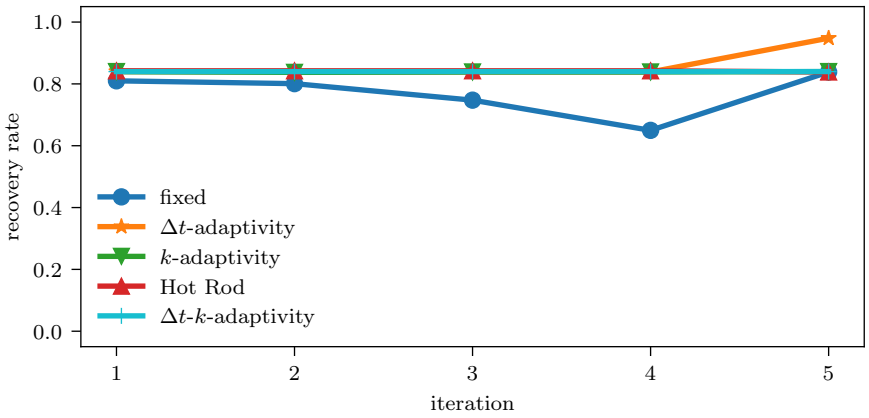


Figure 5.7: Recovery rate for the Lorenz attractor problem depending on the iteration in which the fault occurred. With the fixed strategy, the earlier the fault happens, the more likely it is to be smoothed out. This causes the recovery rate to decrease with iteration number. The exception to this rule is the last iteration, where only faults to the last node have an impact on the solution. As virtually all faults that do not target node 0 are fixed with all other strategies, we find no dependence of recovery rate on the iteration in which it occurs. Again, the last iteration is an exception for Δt-adaptivity, because faults that target the initial condition have no impact with fixed iteration number. Figure reproduced from [12, Fig. 5].

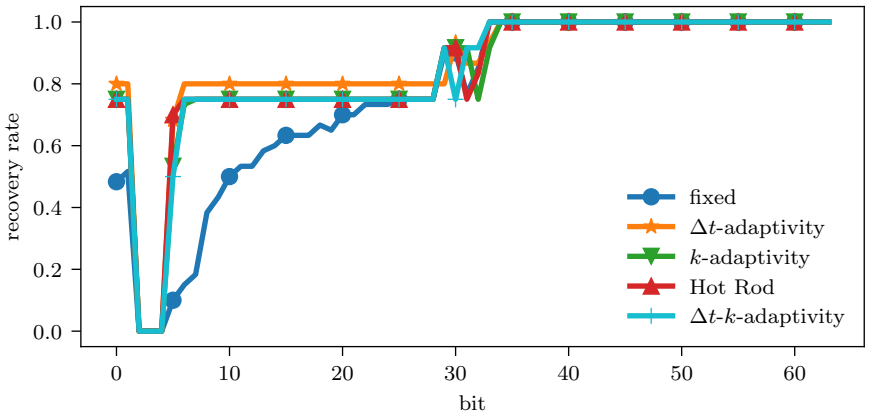


Figure 5.8: Recovery rate for the Lorenz attractor depending on the bit that is flipped by the fault. In this IEEE 754 64 bit floating point representation, bit 0 stores the sign, bits 1 through 12 store the exponent and the remaining bits store the mantissa. Flipping some bits in the exponent leads to overflow errors in the Newton solver and crash the code, resulting in 0 % recovery rate for any strategy. Flipping bits beyond 35 has insignificant impact on this simulation and does not require any recovery strategy. Figure reproduced from [12, Fig. 6].

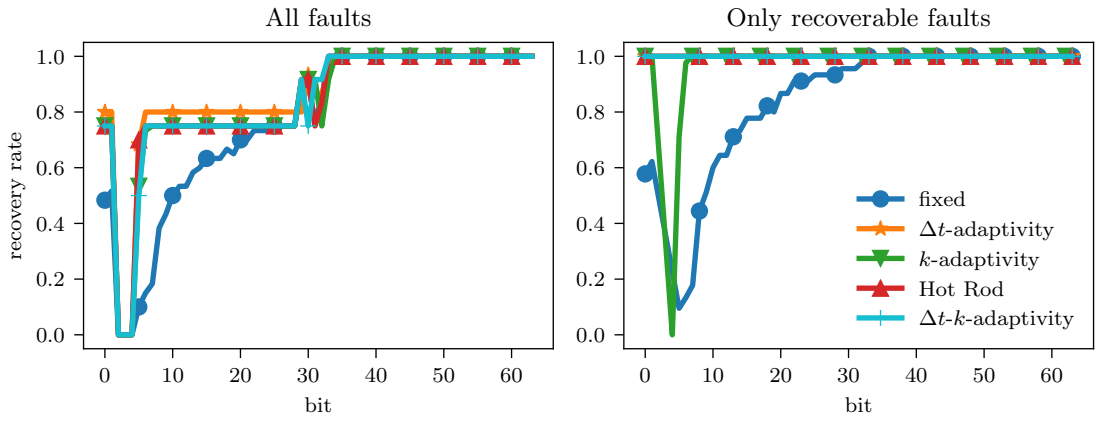


Figure 5.9: Recovery rate for the Lorenz attractor per bit. The left panel shows the recovery rate across all experiments, while the right panel shows only ones with faults that can be recovered. This means we exclude faults to the initial conditions and also faults that crash the code due to overflow errors in the Newton solver. Figure reproduced from [12, Fig. 7].

to compare the capabilities of the strategies amongst each other. The recovery rate per bit with all faults and with only the ones that can be recovered is shown in Figure 5.9. All resilience strategies perform very well and correct nearly 100 % of the faults that they are theoretically able to. Only k-adaptivity struggles with some faults to exponent bits because the Newton solver does not converge for arbitrary initial guess. Strategies that include restarts are better equipped to deal with this type of fault.

Recovery rate for all problems. The resilience properties we identified for the Lorenz problem carry over to the other problems from section 2.6 as well. We show the recovery rate for recoverable faults only in Figure 5.10. We again accept faults as recovered if the global error is no more than 10 % larger than in a fault free reference run. For the Gray-Scott problem, we find perfect recovery rates with all resilience strategies. Since this problem is solved using an IMEX scheme, which does not contain a Newton solver, k-adaptivity does not struggle with faults to exponent bits as is the case in the Lorenz attractor problem. For van der Pol and RBC, we find a few faults in intermediate bits and late iterations that we cannot recover with Δt-adaptivity. They do not trigger a restart, but end up noticeably changing the final solution.

There are also a few faults to intermediate bits in the van der Pol example that Hot Rod cannot correct. This is because of the detection threshold that we selected by balancing false positives and false negatives. Because the changes in time-scale are quite pronounced in this example, we had to accept a certain amount of false positives in order to correct a majority of the faults. By selecting a tighter threshold, all faults can be corrected, at the cost of more false positives. Note that combining Hot Rod with adaptivity would likely alleviate this issue, but we do not show this here as adaptivity by itself appears to be resilient enough.

Balancing resilience and overhead. The previous experiments show that the fixed strategy is rather susceptible to soft faults for all problems. Therefore, if the outcome of the simulation is to be trusted, a resilience strategy should be employed.

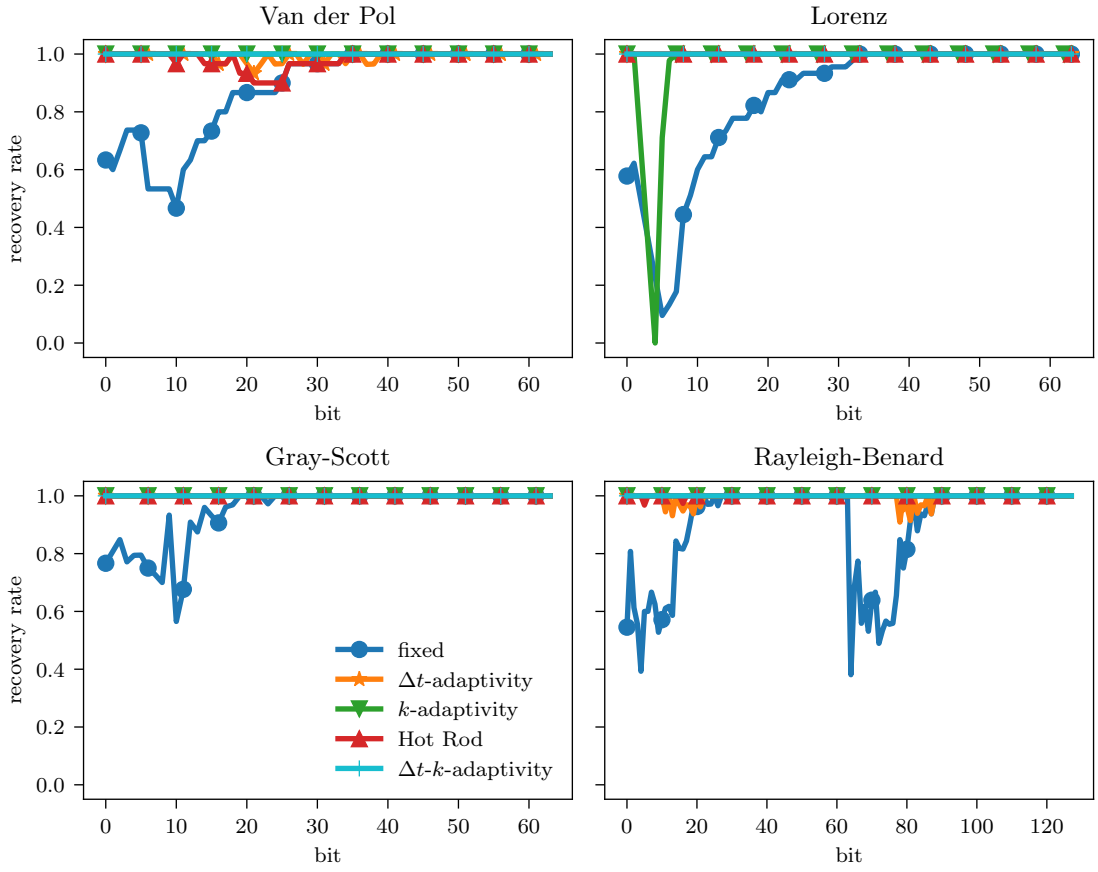


Figure 5.10: Comparison of recovery rate for theoretically recoverable faults for all problems from section 2.6. The Rayleigh-Benard problem uses complex128 numbers instead of the usual float64, which we show as two float64 numbers back to back. The impact of faults appears to be the same per bit regardless of whether the complex or the real part is targeted. The right panels are reproduced from [12, Fig. 8].

In the experiments, we found that all resilience strategies perform well, with only minor differences in their capabilities to correct faults. We found that restarts can be a more effective tool for correcting faults than continued iteration, as convergence is not guaranteed from faulty data. This leaves the k-adaptivity strategy slightly less resilient than the other strategies.

When choosing amongst the step size adaptive schemes and Hot Rod, one should consider that each represents a different balance between resilience and computational efficiency. We have shown in chapter 4 that adaptivity can substantially boost computational efficiency. The Hot Rod strategy, on the other hand, adds substantial overhead in both computation and memory. Recall from section 5.2 that the computational overhead comes in the form of an extra SDC iteration and the memory overhead stems from requiring double the number of solution size objects in memory for the extrapolation method. This can be unpractical and suggests to prefer step-size adaptive strategies over Hot Rod in a wide range of scenarios. Consider that the chance of a fault occurring scales with the physical size of the hardware and the wall time. The added overhead thus increases the likelihood of a fault in the first place. Furthermore, memory is often a limiting factor on GPUs. We will show simulations in chapter 6 and chapter 7 that are distributed across GPUs while operating close to the memory limit of each individual GPU. Dramatically increasing the memory requirement to facilitate resilience would mean similarly increasing the number of GPUs, which likely leads to reduced parallel efficiency and increased use of computational resources.

That being said, Hot Rod is, in principle, capable of detecting faults more accurately, provided the detection threshold is tuned accordingly. Certain very critical applications may therefore still profit from Hot Rod. On the other hand, in order to keep false positives at a minimum, it should then be combined with adaptive step size selection such that the fault detection threshold is tied to the accuracy of the problem.

Additional resilience strategies. Finally, we want to address the faults we deemed uncorrectable by the strategies in section 5.1. The first class of such faults are faults that result in overflows or prevent convergence of the Newton solver. Such faults can be simply corrected by restarting a step whenever this occurs. Only if the issue persists in the restarted step, should the code be crashed. However, this is not specific to SDC, which is why we do not examine this simple resilience strategy here.

Second, we found that restarting does not help in correcting faults when the fault is in the initial conditions, as it will be carried over to the next attempt at the same step. A possible remedy would be to apply replication-based resilience strategies to the initial conditions only. This could take the form of hardware error correction codes or software replication. At the same time, the other solution size objects need not be similarly treated. They can be stored in regions of memory not protected by error correction codes at all, for instance, which would increase the amount of memory per area and per unit of energy.

6 Extending the PDE implementations to GPUs and HPC

In chapter 4, we showed that adaptive step size selection can speed up simulations on an algorithmic level and that PinT extensions of SDC can further speed up simulations via parallelism. In chapter 5, we showed that adaptive SDC is well equipped to recover from faults that are expected on large HPC machines. In this chapter, we turn to running the Gray-Scott and RBC examples on actual HPC machines, parallelizing in both space and time, and discuss necessary modifications in the code. For time parallelism, we will use diagonal SDC, which performed particularly well in chapter 4 and parallelize the spatial discretizations as discussed in section 2.5.4 and section 2.5.9.

The machines at our disposal (see section 2.9) conform to the larger trend that most of the compute power is provided in the form of GPUs rather than CPUs. If we want to harness the approximately 44×10^{15} FLOPS of the JUWELS Booster machine rather than just the approximately 9×10^{15} FLOPS of the most powerful CPU machine that we have access to, JURECA-DC, we have to port the code to GPUs as described in section 2.8.1.

A crucial ingredient for parallelising spectral methods is a library for distributed FFTs, where we use `mpi4py-fft` in the `pySDC` implementations of Gray-Scott and RBC. Therefore, we start by porting this library to GPU and afterwards discuss porting the remainder of the PDE implementations within `pySDC` as well as the infrastructure required for diagonal SDC. We assess performance by comparing strong and weak scaling against CPU versions.

6.1 Porting mpi4py-fft to GPU

As discussed in section 2.8.3, `mpi4py-fft` [42] is an easy to use Python library for distributed FFTs. It relies on MPI ALLTOALLW for communication and uses `FFTW` [54] to compute the transforms. Following the steps outlined in section 2.8.1, we extend the interface of `mpi4py-fft` to `CuPy`. Then, we add NVIDIA's `cuFFT` as GPU capable FFT backend and add a NCCL communication backend to achieve better strong scaling compared to MPI (see section 2.7.2). We opened a pull request[1] to add the GPU port into the main repository of `mpi4py-fft`, which has not been merged at the time of writing.

As `CuPy` is intended to be a drop-in replacement for `NumPy`, adapting the interface is very straightforward. `CuPy` ships with very efficient functions to compute FFTs. The only

[1] `https://github.com/mpi4py/mpi4py-fft/pull/37`

Listing 6.1: Example for deriving a class from `NumPy` arrays. In the `__NEW__` method, a communicator is added. The function for computing the absolute value is overloaded to compute the maximum absolute across all instances of the class that share the same communicator.

```
import numpy as np

class my_array(np.ndarray):
    def __new__(cls, communicator, **kwargs):
        obj = np.ndarray.__new__(cls, **kwargs)
        obj.communicator = communicator
        return obj

    def __abs__(self):
        return self.communicator.allreduce(np.linalg.norm(
                                            self, np.inf))
```

part requiring more attention is the communication. We will start by describing the layout of the code and detail where changes need to be made for the GPU port and then discuss communication in more detail.

Layout of the library and changes needed for GPU port. `mpi4py-fft` is neatly separated into front-end and back-end, which consist of two modules each. The modules in the front-end are DISTARRAY, which inherits from `NumPy` array and adds an interface for distribution functionality, and PFFT, which is the interface for the distributed transforms.

Regarding DISTARRAY, we refactor the code slightly, adding an abstract base class containing only the distribution interface and add two classes that are derived from this base class as well as `NumPy` and `CuPy` array respectively. The main difference between the derived classes is in the `__NEW__` methods, which have a different signature in `NumPy` and `CuPy`. In Python, object instantiation is split between the static method `__NEW__` and the instance method `__INIT__`, where `__NEW__` is called first and can return any object. By overloading `__NEW__` instead of `__INIT__` in subclassing of `NumPy` and `CuPy` arrays, one can make use of the universal function interface [119] for efficiently performing elementwise operations on the arrays in compiled code [100]. We provide an example of a simple subclass of the `NumPy` array class with an added attribute and overloaded function for computing the absolute value in Listing 6.1. The only meaningful change we do to PFFT is to add an argument for the user to select a communication back-end.

The back-end of `mpi4py-fft` is split into LIBFFT, which provides wrappers for various FFT back-ends and PENCIL, which takes care of communication. In LIBFFT, we add a wrapper for the FFT functions in `CuPy`, which are themselves wrappers for `cuFFT`. `cuFFT` is a library developed by NVIDIA to provide the same functionality and similar interface as `FFTW`, but on GPUs. It also plans the FFTs first, which means it selects the fastest implementation among a few candidates at runtime. In `CuPy`, the plans are generated when an FFT of a certain type and shape is first computed, and then the plan is cached and reused in subsequent calls of the FFT functions. This allows for very efficient FFT computations without explicit optimization by the user.

The PENCIL module is named after the common distribution style "pencil decomposition," referring to distributing three-dimensional data along two dimensions [9, Fig. 2]. It

is capable of arbitrary decomposition styles that can be set by the user when instantiating a PFFT object.

Internally, the module is split into TRANSFER, which performs the data redistributions given input and output layouts with MPI ALLTOALLW, and PENCIL, which computes layouts for the data with alignment along a given axis. PENCIL also acts as an interface between the actual communication and the higher level transforms. The only change we make to PENCIL are again adding the communication back-end as a parameter. The main change to the library overall is to add a NCCL communication back-end to the TRANSFER class, which we will discuss now.

The NCCL communication back-end. The MPI installations on JUWELS Booster are CUDA aware, which means they are capable of communicating data directly between GPUs while leveraging the fast NVLINK interconnect. However, two issues arise.

First of all, ALLTOALLW remains a niche communication routine and the implementation on GPU is still very immature at the time of writing. In general, the cost of each individual point-to-point communication is a function of the network's latency and bandwidth. When the data to be communicated is small, latency dominates the cost, whereas the cost of communicating large chunks of data is bound by the bandwidth. For all-to-all communication, there is a multitude of algorithms that are split in two groups. There are algorithms where the number of communication steps scales linearly with number of involved tasks, and there are algorithms where the number of communication steps scales logarithmicaly [21, 96]. A communication step here means a round of concurrent point-to-point communications among all tasks. The former optimises the amount of data traversing the networks, whereas the latter optimises the number of communication steps. MPI implementations attempt to select algorithms that minimise the communication cost by choosing among these algorithms based on the size and number of individual point-to-point communications.

We compare the time taken for communication of the same data across four GPUs with different all-to-all versions and MPI implementations (OpenMPI and ParaStationMPI) in Figure 6.1. It becomes apparent that ALLTOALLV is the most robust routine and that ALLTOALLW performs extremely poorly in both implementations. For large data with OpenMPI, there is up to almost three orders of magnitude difference between ALLTOALLW and ALLTOALLV.

Most distributed FFT libraries use ALLTOALLV and it would be possible to implement ALLTOALLV also in `mpi4py-fft`. However, the second issue is that MPI runs on the host and is not stream aware. This means communication is not submitted as a kernel to the GPUs, but requires explicit synchronization of GPU to host before communication. NCCL, on the other hand, is stream aware and available to Python users via `CuPy`, making it an attractive alternative. See section 2.7.2 for an overview of MPI on GPUs and NCCL as a GPU specific alternative.

On the other hand, NCCL is much less mature than MPI and is missing many routines including any of the all-to-all versions. Instead, they can be implemented using multiple calls to point-to-point communication within NCCL groups. Recall that NCCL groups are used to efficiently launch multiple overlapping point-to-point communications in a single kernel.

ALLTOALLW is replicated here via a series of ring-sends. In each iteration, contiguous

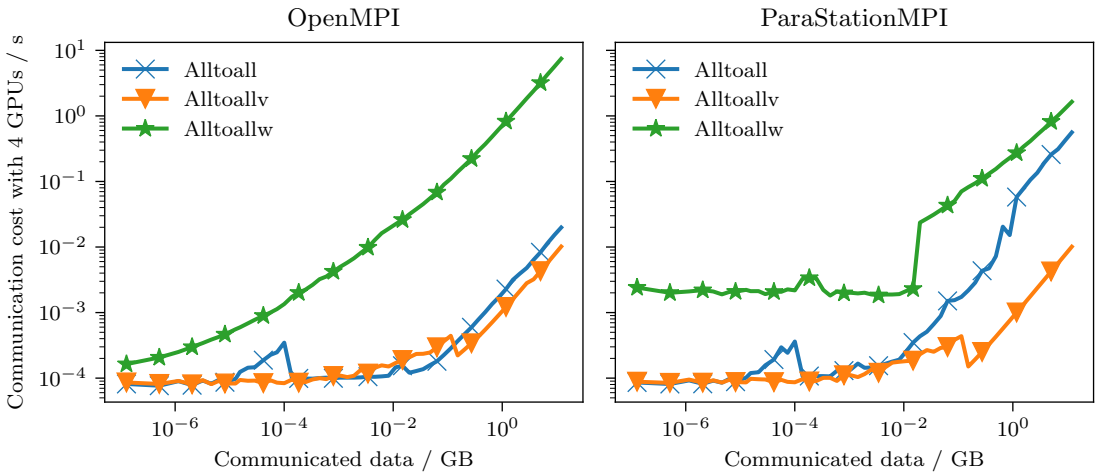


Figure 6.1: Time required for all-to-all communication between all GPUs on a single node on JUWELS booster using different MPI routines of two MPI implementations. We start by generating $N \times NS$ random double-precision floating point numbers on each task in a contiguous array, where we vary N to determine the size of the communicated data and $S = 4$ is the number of MPI ranks involved in the communication. Because the shape of the distributed data is divisible by the number of MPI ranks, all all-to-all versions can be used and yield the same result after communication. Due to the use of different communication algorithms, ALLTOALLV performs better than other routines, while ALLTOALLW performs extremely poorly. See section 2.1.3 for more information on the different all-to-all communication routines. Note that this experiment is not necessarily representative of communication cost with larger number of tasks.

buffers are prepared for the data to be sent and received on all ranks. Then, the ring-send is launched via point-to-point communication in a NCCL group. Finally, the received data is unpacked from the buffer to a potentially non-contiguous part of the result array. The procedure is sketched in pseudocode in Algorithm 7.

While the advantage of MPI ALLTOALLW over ALLTOALLV is that the steps of copying to and from buffers can be omitted and is quoted in [42] as the reason for using ALLTOALLW in `mpi4py-fft`, the additional overhead can be mitigated on GPUs by using CUDA graphs (see section 2.7.1). In this ALLTOALLW implementation, all iterations of the ring-sends, including copy as well as send operations, are recorded to a single graph during the first execution, and then launched as a single kernel in subsequent calls.

We compare performance of our NCCL ALLTOALLW implementation to what is available to us with MPI in Figure 6.2. We significantly outperform the ALLTOALLW implementations that are the basis for the original transfer classes in `mpi4py-fft`. We find that MPI ALLTOALLV performs even better than our NCCL implementation purely concerning communication in the experiment with four tasks. However, because MPI is not stream aware, we expect the NCCL implementation to exhibit better strong scaling and note that ALLTOALLV requires additional copy operations that are not captured here. We leave implementation of MPI ALLTOALLV in `mpi4py-fft` and comparison to the NCCL communication back-end for future work.

Our NCCL all-to-all algorithm scales linearly in the number of communication steps with the number of tasks. We note that this may not be the most efficient algorithm for any combination of data sizes and numbers of tasks. As discussed above, MPI implementations

Algorithm 7 Custom Alltoallw implementation in NCCL. The entire procedure is recorded to a CUDA graph during the first execution, and dispatched as a single kernel in subsequent executions.

```
N, r ← Number of GPUs, MPI rank                    ▷ Initialize variables
sendbufs, recvbufs ← {}, {}

NCCL group start                    ▷ Launch all communications in a single kernel
for i in 0, 1, ..., N do
    send_to ← (r + i) % N
    recv_from ← (r - i + N) % N
    sendbufs[i] ← contiguous copy of data to send
    recvbufs[i] ← contiguous empty array
    NCCL receive from recv_from
    NCCL send to send_to
end for
NCCL group end          ▷ Wait for all communication to finish before unpacking

copy recvbufs to destination array
```

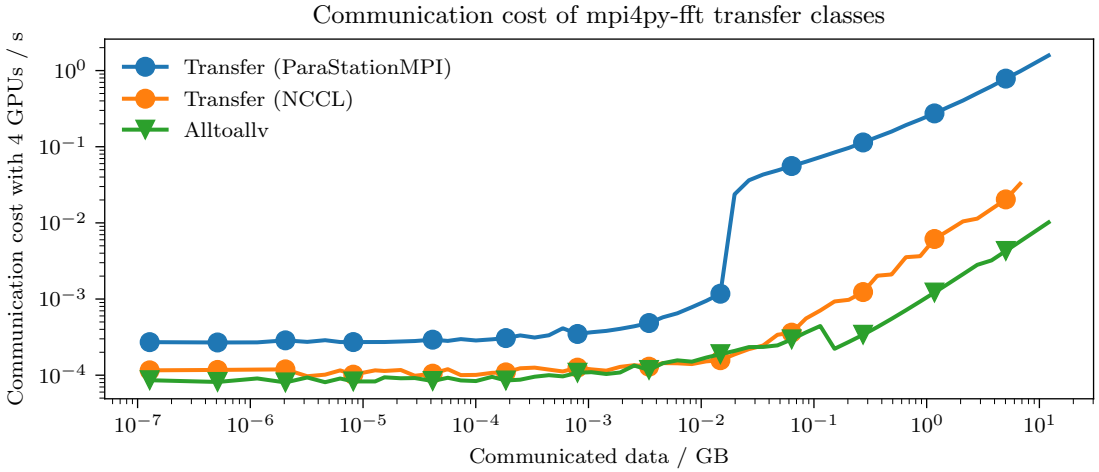


Figure 6.2: Communication cost with `mpi4py-fft` transfer classes analogous to Figure 6.1. The line labeled "Transfer (NCCL)" is using our newly implemented transfer class with NCCL as communication back-end. For reference, we show the original transfer class that uses MPI ALLTOALLW ("Transfer (ParaStationMPI)") and MPI ALLTOALLV, which we found to perform best out of the MPI all-to-all patterns. The NCCL back-end performs substantially better than the original transfer class. While ALLTOALLV appears to be better optimized for large data, the stream-awareness of NCCL is an advantage for small data.

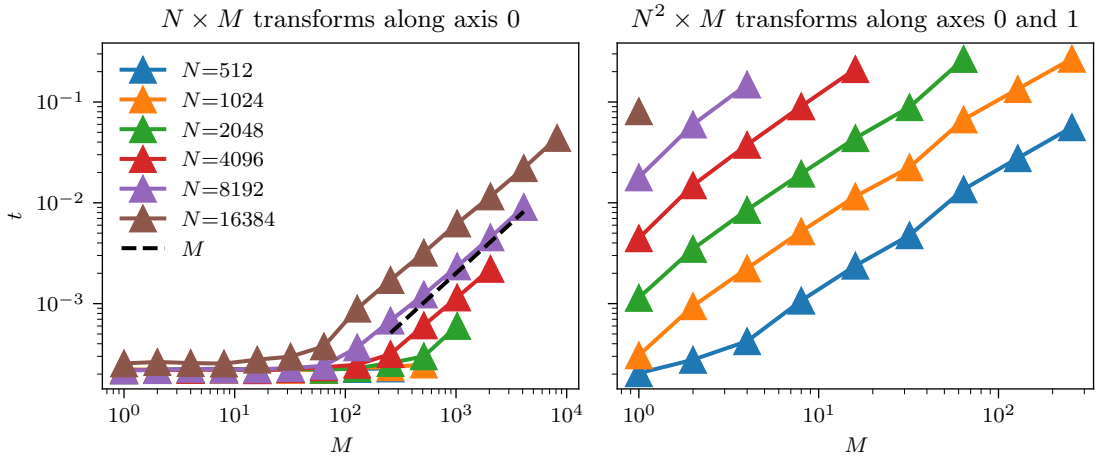


Figure 6.3: Single GPU potential for parallelism in FFTs. By submitting a single kernel with M 1D (left) or 2D (right) FFTs of size N or N^2, we can investigate a single GPU's potential for parallelism. We find below a certain size or number of transforms, the cost is dominated by the kernel launch cost. Only after parallelism is saturated, does the wall time scale linearly with M, providing potential for speedup by distributing along this axis. In particular, we see that small sizes up to 1024^2 offer no potential for speedup by distributed computing at all on this hardware and that strong scaling with sizes of 8192^2 saturates at 64 GPUs, before even factoring in communication cost. In 3D, on the other hand, potential for speedup by distributing the FFTs is already given at $N = 512^3$. We show only transforms that fit in the 40 GB of memory of a single GPU. In 3D, distribution along the third axis is needed already from 1024^3.

select logarithmically scaling algorithms for many tasks and small message size. We leave optimization of the NCCL communication back-end by exploring other communication algorithms for future work.

On-device concurrency. Before considering distributed FFTs among multiple devices, we explore the capabilities for single-device concurrency of the A100 GPUs. To this end, we generate random double precision floating point data of size $N \times M$ and measure the time it takes to compute M one-dimensional FFTs of length N along the first axis, submitted as a single call to `cuFFT`. The GPU then automatically schedules the concurrent FFTs to efficiently utilize all available SMs. We show the result of this experiment and an analogous experiment with M concurrent two-dimensional FFTs of size $N \times N$ in Figure 6.3.

We observe that FFTs of small data are efficiently parallelized to the point that execution time of the kernel is dominated by kernel launch time. Run-time starts to rise linearly with the number of transforms submitted in the same kernel only once on-device parallelism is saturated. Only then is speedup by distributing the FFTs at all possible, which we will investigate next.

Achieved speedup and comparison with cuFFTMp. We now assess the performance of the GPU port by comparing to the original CPU version, as well as the highly optimized CUDA library developed by NVIDIA, `cuFFTMp`. At the time of writing, `cuFFTMp` is still not fully released and has no interface to `CuPy` arrays, which motivated the GPU port of `mpi4py-fft`.

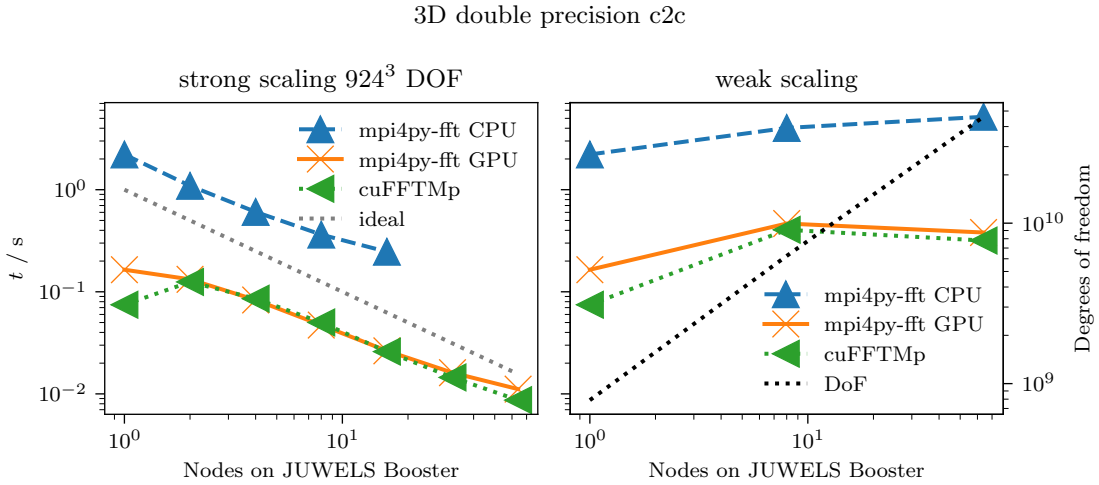


Figure 6.4: Scaling the GPU-ported version of `mpi4py-fft` compared to the CPU version and `cuFFTMp`, the non-Python competition from NVIDIA. The size was chosen to be 924^3 because larger sizes exceed the memory of four GPUs during the transform when using double precision. The x-axis is in nodes on JUWELS Booster, which have four GPUs and 48 CPUs each. Consequently, twelve times as many CPUs as GPUs are used, which means slab decomposition (distribution along a single axis) is saturated for strong scaling already at 19 nodes with CPUs, but only at 231 nodes on GPUs. Strong scaling stops much earlier at 64 nodes (256 GPUs) because compute-times are limited by kernel launches from below and communication cost increases with the number of involved processes.

The major advantage that `cuFFTMp` has, is access to NVSHMEM (see section 2.7.3). At the time of writing, this is the most efficient way of communicating small data between GPUs and is not easily accessible to Python developers. This gives `cuFFTMp` an edge in strong scaling. Otherwise, `cuFFTMp` also relies on `cuFFT` for the actual transforms.

A comparison of 3D distributed FFTs using the CPU version of `mpi4py-fft`, the GPU port and `cuFFTMp` can be found in Figure 6.4. Both GPU based libraries outperform the CPU library by a factor of nine for the same number of compute nodes on JUWELS booster. When maximizing strong scaling, we find `mpi4py-fft` on GPUs to be 22 times faster than on CPUs. In strong scaling, the GPU-port of `mpi4py-fft` is competitive with `cuFFTMp` up to 32 nodes or 128 GPUs, after which the Python code does not continue to strong-scale well. On 32 nodes, we get a speedup of 10.2 with parallel efficiency of 32 %, and on 64 nodes, we get a speedup of 14.7 with parallel efficiency of 23 %. `cuFFTMp` on the other hand was demonstrated by NVIDIA to strong scale beyond that [26]. Weak scaling is competitive with `cuFFTMp` up to the maximum of 256 GPUs that were tested here.

Note that in both modes of scaling, we find the communication cost to increase significantly when moving from one to multiple nodes because the GPUs on each node are connected with the fast NVLINK, but with the slower Infiniband between nodes. We also note that this experiment shows the capabilities of Python in the HPC context. Python code can be MPI parallelized and when it is used to interface compiled code, it can run very efficiently on modern HPC machines. With this GPU port of `mpi4py-fft` we have laid the groundwork for parallel-in-space spectral methods in Python on GPUs. We now turn to the remaining parts of `pySDC` that require attention in order to perform large-scale space-time-parallel simulations on JUWELS Booster.

6.2 Porting pySDC to GPU

pySDC is primarily concerned with timestepping methods, but it also includes implementations of problems for testing SDC related methods on. See section 2.8.2 for more information on pySDC in general. We begin by illustrating how problem classes are implemented in pySDC and subsequently ported to GPU via a simple example and the discuss how to use the sweeper classes on GPUs.

6.2.1 Porting pySDC problem classes to GPU

The equation we implement in the example is $u_t = -u + \sin(t)$. We employ IMEX splitting (see Equation 2.52), treating the term $f^I = -u$ implicitly and the term $f^E = \sin(t)$ explicitly. Therefore, we need a function that can evaluate the individual terms called EVAL_F and a function called SOLVE_SYSTEM that inverts only the implicit part, given some right hand side. The sweeper modules in pySDC then call these functions to assemble the right hand side and solves in the SDC iterations.

In pySDC, we use a MESH datatype for the solution values that is derived from NumPy array and adds a communicator as well as infrastructure for initializing empty arrays and an IMEX_MESH datatype to store right hand side evaluations. The latter is essentially a larger mesh that has attributes IMEX_MESH.IMPL and IMEX_MESH.EXPL to access f^I and f^E respectively. See Listing 6.2 for Python code implementing the above.

Next, we want to port this problem to GPU. Note that we define three class attributes in Listing 6.2, namely the datatype for the solution, the datatype for the right hand side evaluations and the numerical library we want to use to evaluate $\sin(t)$. The datatypes are derived from NumPy array and the numerical library is NumPy itself. As described in section 2.8.1, porting to GPU is done by swapping NumPy for CuPy. CuPy array derived datatypes for pySDC were developed in [100] and porting the example from Listing 6.2 to GPU is as simple as swapping the three class attributes, as shown in Listing 6.3.

6.2.2 Using pySDC sweepers on GPUs

As the sweeper classes use an abstract interface to the problem classes, the same classes can be employed on both CPU and GPU. However, when using diagonal SDC, reduce operations are needed for summing up the distributed f^I and f^E in the right hand side of Equation 2.52, as illustrated in Algorithm 8. Recall from section 2.7.2 that using an MPI communicator on GPUs without modification from the CPU implementation does not work because CPU and GPU run asynchronously. To address this issue, and in order to write GPU agnostic code, we implement a wrapper for MPI communicators that uses NCCL under the hood if the data resides on GPU, but maintains the interface of the MPI communicator. When the code is run on GPUs, the user can simply pass a wrapped MPI communicator instead of a regular MPI communicator.

6.3 Parallel Gray-Scott implementation on GPUs

As discussed in section 2.6.3, we use a Fourier-pseudospectral method with IMEX splitting for this problem. We first discuss spatial parallelism, then porting to GPU and then show performance measurements.

Listing 6.2: Example of a simple problem class implementation in `pySDC` for integrating $u_t = -u + \sin(t)$ with IMEX splitting. The problem class needs to define how to initialize data in `__INIT__`, a function for evaluating the right hand side called `EVAL_F`, and a function for an Euler step of the implicit part called `SOLVE_SYSTEM`. We illustrate here how to pass a communicator to the `pySDC` datatype, even though it is hardly useful in this scalar problem.

```
import numpy as np
from pySDC.core.problem import Problem
from pySDC.implementations.datatype_classes.mesh import mesh,\
    imex_mesh

class my_problem(Problem):
    # Simple pySDC problem class for u_t = -u + sin(t)
    # with IMEX splitting.
    dtype_u = mesh
    dtype_f = imex_mesh
    xp = np

    def __init__(self, communicator):
        init = (1, communicator, np.dtype('float64'))
        super().__init__(init)

    def eval_f(self, u, t):
        # evaluate right hand side
        f = self.f_init
        f.expl[:] = self.xp.sin(t)
        f.impl[:] = -u
        return f

    def solve_system(self, rhs, dt, *args, **kwargs):
        # implicit Euler step for linear part
        sol = self.u_init
        sol[:] = rhs / (1 + dt)
        return sol
```

Listing 6.3: GPU port of the problem class from Listing 6.2. We only need to change class attributes here, swapping `NumPy` for `CuPy`, because we set up the CPU implementation sufficiently abstractly.

```
import cupy as cp
from pySDC.implementations.datatype_classes.cupy_mesh import \
    cupy_mesh, imex_cupy_mesh

class my_problem_GPU(my_problem):
    dtype_u = cupy_mesh
    dtype_f = imex_cupy_mesh
    xp = cp
```

Algorithm 8 Iteration of parallel diagonal IMEX SDC. The first step is to compute the right hand side from Equation 2.52. Note that some terms are absent when the preconditioners are diagonal. Communication using MPI or NCCL reduce is needed when computing the integral using the full quadrature rule. Every task stores the initial conditions to the current step u_0 and the solution and right hand side evaluation at one collocation node. Note that the right hand side evaluations are computed only once per iteration and in parallel in practice.

$r \leftarrow$ MPI or NCCL rank in communicator of size M
$y \leftarrow$ empty solution size array ▷ Assemble right hand side for implicit solves into y
for m in $1, ..., M$ **do**
  if m is r **then**
    recvBuf $\leftarrow y$
  else
    recvBuf $\leftarrow$ None
  end if
  sendBuf $\leftarrow \Delta t q_{m,r}(f^I(u_r^k) + f^E(u_r^k))$
  Reduce(sendBuf, recvBuf, root=m, op=MPI.SUM)
end for
$y \leftarrow y + u_0 - \Delta t \tilde{q}^I_{r,r} f^I(u_r^k)$
$u_r^{k+1} \leftarrow (1 - \Delta t \tilde{q}^I_{r,r} f^I)^{-1} y$ ▷ Perform implicit solves in parallel

6.3.1 Parallelizing the Gray-Scott implementation in space

When evaluating the right hand side of Equation 2.110, we need to multiply by the derivative matrix in spectral space and evaluate the non-linear terms in physical space. To run in parallel, we need to construct local derivative matrices and use distributed FFTs, as discussed in section 2.5.4. Because the nonlinear parts only include local terms, they parallelize trivially and do not require any communication.

In the IMEX SDC iteration only the linear diffusion part is inverted. Spelling out the left hand side of the iteration, Equation 2.52, for the Gray-Scott equation we obtain

$$\left(\begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix} - \Delta t \tilde{q}^I_{m+1,m+1} \begin{pmatrix} \nu_u \Delta & 0 \\ 0 & \nu_v \Delta \end{pmatrix}\right) \begin{pmatrix} u \\ v \end{pmatrix}_{m+1}^{k+1} = \vec{y}, \tag{6.1}$$

where we use $\vec{y}$ short for the right hand side in the solves, which is a sum of various right hand side evaluations and the initial conditions.

Recall from section 2.5.1 that, in Fourier base, the (local) derivative and the identity matrices are diagonal at the same time. Therefore, the matrix we need to invert in Equation 6.1 is diagonal and we can invert simply by dividing the right hand side by the diagonal entries of the local matrix in parallel.

6.3.2 Porting the Gray-Scott implementation to GPU

To port to GPU, we proceed analogously to section 6.2.1, changing class attributes to swap `NumPy` for `CuPy`. In order to perform the FFTs on GPU, we employ the GPU port of `mpi4py-fft` that we developed in section 6.1. Again, we use class attributes to configure

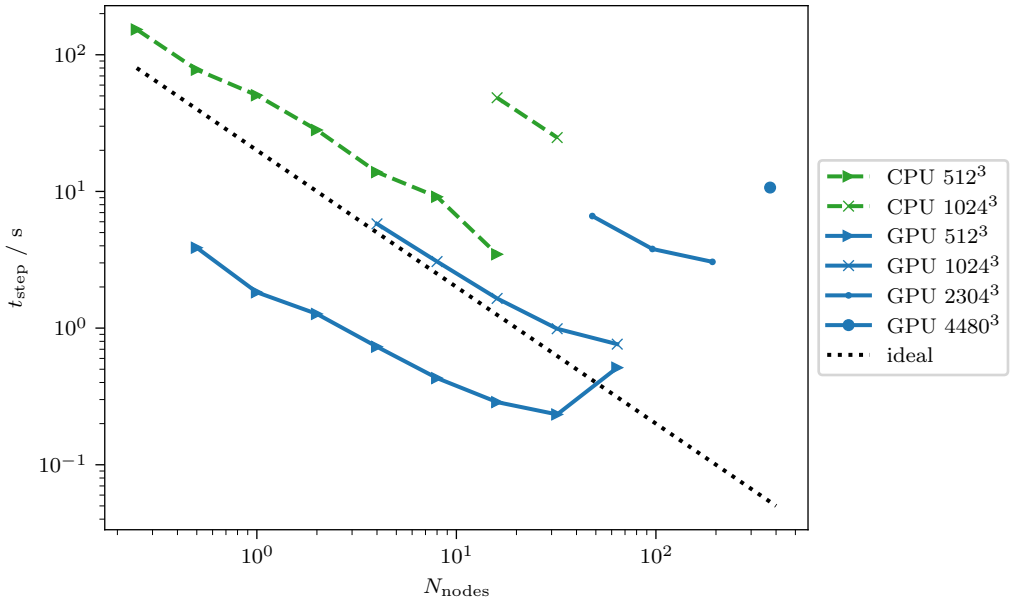


Figure 6.5: Combined strong and weak scaling plot for the 3D Gray-Scott example with serial SDC on CPUs and GPUs showing wall time for a single SDC step versus the number of nodes used. The numbers in the legend indicate the spatial resolution. Lines start when the problem fits in memory and end when further scaling became poor or impossible. We find the CPU implementation to take approximately 10 times as long as the GPU implementation for the same resolution at most node counts.

`mpi4py-fft` to either run on CPUs or GPUs, by setting the communication and FFT back-ends.

6.3.3 Parallel scaling

We now investigate the parallel scaling of the Gray-Scott implementation on both CPUs and GPUs, using serial and diagonal SDC. We choose four Gauß-Radau nodes and perform four iterations, which yields a fourth order method that can be distributed in time on four tasks. For each experiment, we run 15 steps of fixed step size and record the average time for a full time-step. As the computations required for adaptivity are not expensive and the computational cost of the IMEX solver does not depend on the step size, we assume the results to be applicable to SDC with adaptive step size selection as well.

CPU times are recorded on JUWELS (section 2.9.3), where we found that limiting the number of tasks to 16 per node significantly increased performance. The reason that using fewer tasks per node than available yields better performance is likely rooted in saturation of the on-node memory bandwidth. GPU times are recorded on JUWELS booster (section 2.9.4), with four tasks per node and one task per GPU.

Comparing CPU to GPU. We show wall-time measurements of various configurations, showing both strong and weak scaling of both CPU and GPU implementations of serial SDC in Figure 6.5. We find the GPU implementation to massively outperform the CPU implementation by a factor of more than 10 at most node counts. The contrast is so stark

that we record on CPU only up to resolutions of $N = 1024^3$. For further scaling results with larger resolution, we pick the resolution to be a multiple of the number of available tasks to avoid load balancing issues. During normal operation of JUWELS booster, up to 384 nodes ($2304 = 6 \times 384$) can be reserved by individual jobs. On request, the system can be reserved for larger jobs up to all 926 nodes. However, they are not usually all online at the same time, such that we use 896 nodes ($4480 = 5 \times 896$) as representative of the entire machine.

For lower resolution experiments, we find excellent strong scaling on both CPUs and GPUs. With $N = 2304^3$, we find strong scaling to saturate quickly, as the number of involved tasks is large and so is the communication cost. At $N = 4480$, we were able to record only a single data point for space-only and diagonal SDC on GPUs each, because fewer GPUs would not have enough memory and the system does not have more GPUs.

We only use slab decomposition, which sets an upper limit to the number of tasks that we can employ for a given resolution. Since the data in physical space is real, `mpi4py-fft` returns half as many complex numbers in spectral space (see section 2.5.5), and `mpi4py-fft` requires to have at least one data in each dimension per task, we can only use up to $N/2$ tasks for a resolution of N^3 with slab decomposition. Using pencil decomposition, we could extend the scaling to up to $N/2 \times N$ tasks for the same resolution, at the cost of one additional all-to-all communication. With GPUs, we find strong scaling to saturate before reaching the limits of slab decomposition and as CPUs are not competitive in slab decomposition, it is unlikely that they could compete with the GPU port with more tasks and also more all-to-all communications.

Diagonal SDC. We show wall-time measurements of various configurations, showing both strong and weak scaling of the GPU implementation of both serial and parallel diagonal SDC in Figure 6.6. We attempted to record another point of space-only scaling at 2240 GPUs. However, an internal NCCL error led the code to fail. The error was raised within the NCCL group in the all-to-all communication in `mpi4py-fft` (see Algorithm 7). We are unable to determine why this error occurs as the code works well for fewer GPUs, as well as more GPUs, but differently distributed. At the time of writing, we are in contact with NVIDIA about this issue, but it remains unresolved.

From experiments at lower resolution and node count, however, it can be inferred that the code is unlikely to scale well in space past the 1494 GPUs that we show in Figure 6.6. By using only 896 GPUs in space and four tasks in time, we can extend the scaling at decent parallel efficiency to essentially the whole machine.

The primary lesson that can be learned from Figure 6.6 is that diagonal SDC is an excellent tool for expanding strong scaling capabilities. Both the distributed FFTs and diagonal SDC are embarrassingly parallelizable, but the communication differs significantly. Distributed FFTs require all-to-all communication with n_{space} tasks, whereas diagonal SDC requires reduce operations with n_{time} tasks (see Algorithm 8). As discussed in section 6.1, the number of communication steps in our all-to-all implementation on GPUs scales linearly with the number of tasks. The number of communication steps in reduce operations, on the other hand, scales logarithmically with the number of tasks [15]. We find that using diagonal SDC shifts the numbers of tasks involved in each communication to patterns that require less time overall. In particular, this is what allowed us to scale to the entire JUWELS Booster machine.

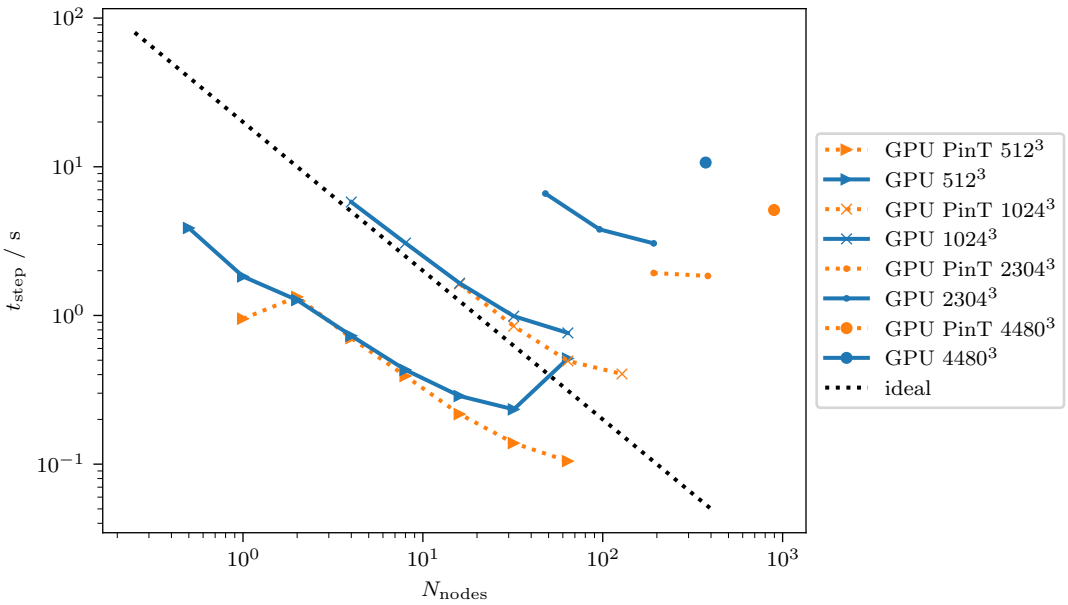


Figure 6.6: Combined strong and weak scaling plot for the 3D Gray-Scott example with serial and parallel diagonal SDC on GPUs showing wall time for a single SDC step versus the number of nodes used. "PinT" indicates that parallel diagonal SDC was used and the number denotes the spatial resolution in the legend. Lines start when the problem fits in memory and end when further scaling became poor or impossible. Diagonal SDC can extend the scaling capabilities significantly.

Another observation that we made, which is not visible in the figures, however, is that spatial parallelism is more efficient at reducing the memory demand per task than diagonal SDC. The memory demand ξ per task in serial SDC, neglecting temporary variables, is

$$\xi_{\text{serial SDC}} = (\underbrace{1}_{u_0} + \underbrace{M}_{\vec{u}} + \underbrace{M}_{F(\vec{u})} + \underbrace{1}_{u_{\text{end}}}) \times \xi_u / n, \tag{6.2}$$

with ξ_u the total non-distributed memory demand of a single solution size object and n the number of tasks in space and $\vec{u}$ here refers to the vector of solutions at collocation nodes $1, \dots, M$. The memory demand for diagonal SDC with the same number of tasks in total, is

$$\xi_{\text{diagonal SDC}} = (\underbrace{1}_{u_0} + \underbrace{1}_{u} + \underbrace{1}_{f(u)} + \underbrace{1}_{u_{\text{end}}}) \times \xi_u \times M / n. \tag{6.3}$$

The factor of M appears because fewer tasks are used in space when adding time-parallelism. The ratio of the two is

$$\frac{\xi_{\text{diagonal SDC}}}{\xi_{\text{serial SDC}}} = \frac{2M}{M+1}. \tag{6.4}$$

In our case with $M = 4$, space-time parallel diagonal SDC therefore requires 60 % more memory before considering temporary variables at a given number of tasks than space-parallel SDC. As we found the memory that is available on each GPU to be a significant limitation, this may need to be considered in some scenarios.

Parallel efficiency. In order to better determine the weak scaling capabilities, we investigate parallel efficiency in more detail. We define parallel efficiency as the throughput

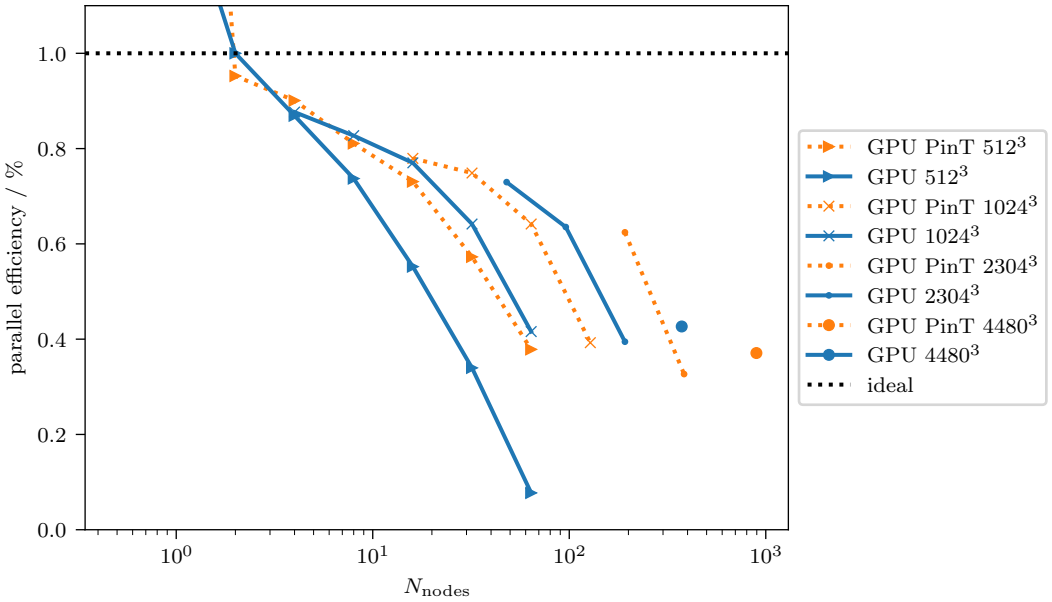


Figure 6.7: Parallel efficiency of the GPU implementation of the 3D Gray-Scott example with serial and diagonal SDC. We normalize to the throughput obtained with two nodes and space-only scaling rather than the troughput on a single GPU because the performance drop sharply when going from single to multi-node. The data is the same as in Figure 6.6.

in the time taken to solve a single SDC step per number of degrees of freedom per second per task divided by a reference throughput. Typically, one chooses the serial throughput as reference, but we do not want to capture the impact of the different intra-node (NVLINK) and inter-node (Infiniband) interconnects. The result is a sharp drop in parallel performance when moving to multi-node, which is purely a hardware feature with no algorithmic work-around. We show the resulting plot in Figure 6.7 and select values in Table 6.1. We find a decent 37 % parallel efficiency with space-time parallel SDC on the entire machine relative to the two-node base-line, compared to 43 % parallel efficiency with space-only scaling on what is less than half of the machine. This demonstrates the capability of Python to support GPUs on HPC at scale as well as the capability of diagonal SDC to extend scaling capabilities at good parallel efficiency. Note that we did not use any optimization method in `pySDC` such as CUDA graphs or multiple streams, in order to make it easy to run the code on both CPU and GPU. The only optimization we did is to take care to (implicitly) synchronize CPU and GPU as little as possible.

Note that FFTs are most efficient for highly composite numbers. Because we choose resolution as multiple of the number of available tasks for the larger resolution tests, a slight reduction in efficiency compared to the power-of-two resolutions should be expected even before considering added communication cost.

6.3.4 Other reaction diffusion problems

In this Gray-Scott implementation, we treat the diffusion part implicitly and the reaction part explicitly. Other reaction-diffusion equations can be solved using virtually the same code for the diffusion part and altering only the reaction terms in the explicit function

	$N = 512^3$		$N = 2304^3$		$N = 4480^3$	
number of nodes	1	2	96	192	374	896
space-parallel	1.38	1	0.64	-	0.43	-
space-time-parallel	2.67	0.95	-	0.62	-	0.37

Table 6.1: Parallel efficiency for the Gray-Scott example on GPUs normalized to throughput per GPU with space-only parallelism on two nodes. We show select values from Figure 6.7. Namely single-node, best performing multi-node (two nodes), the point at which efficiency starts to degrade with 2304^3 degrees of freedom, and the highest number of nodes we were able to measure with each parallelization scheme. We find significantly higher efficiency on single node due to the faster intra-node interconnect compared to inter-node. We observe that diagonal SDC can be employed to achieve almost the same efficiency at higher node count. These numbers should be taken with a grain of salt as the efficiency of computation in FFTs is slightly different for the unusual resolutions we chose here.

evaluations.

In `pySDC`, we implemented an abstract base class for reaction-diffusion equations with periodic boundary conditions. By porting this to GPU, we automatically port all derived classes with specific equations. Next to Gray-Scott, `pySDC` contains implementations for nonlinear Schrödinger, Allen-Cahn and Brusselator problems that are derived from this class. Since the reaction parts require similar computational operations for all problems, we expect similar parallel performance with all other problems as we measured for Gray-Scott.

6.4 Parallel implementation of Rayleigh-Benard convection

We now investigate GPU porting and parallel scaling of our Rayleigh-Benard implementation. While significant parts work analogously as in the Gray-Scott implementation, certain characteristics of the ultraspherical base in the vertical direction require additional attention.

6.4.1 Parallelizing the Rayleigh-Benard convection implementation

First, we discuss parallel implementation of the right hand side evaluation and then turn to the IMEX solves, where we first discuss preconditioning strategies and then linear solvers.

Evaluating the right hand side. Evaluating the right hand side of Equation 2.111 proceeds conceptually similarly to the Gray-Scott equation. We evaluate linear terms by multiplication with the L matrix from Equation 2.116 in spectral space and evaluate the nonlinear term f_{nonlin} in Equation 2.116 in physical space. As discussed in section 2.5.9, the derivative matrices along z-direction, which is discretized with an ultraspherical method, are not diagonal and we cannot split them in local matrices as in the Fourier base. However, we can build local matrices in x-direction, which is discretized with Fourier base, and then build matrices for two-dimensional discretizations via Kronecker products of local in x and global in z matrices of one-dimensional discretizations.

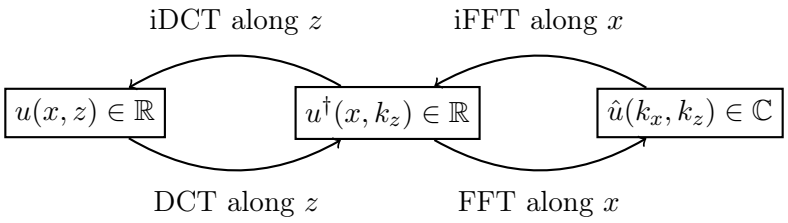


Figure 6.8: Illustration of the transforms in the RBC problem. Because our implementation of the (inverse) DCT only accepts real-valued data, we have to perform the DCT before the FFT and the inverse FFT before the inverse DCT. We denote the intermediate state where the data is transformed along z-direction but not x-direction as $u^\dagger$.

In preparation for the GPU port, we use the method by Makhoul [109] detailed in section 2.5.7 for computing the DCT along z using FFTs with `mpi4py-fft`. However, this method only works with real-valued data, which defines an order of the transforms in x- and z-directions, as illustrated in Figure 6.8. This order is sub-optimal here because it requires additional redistributions. When evaluating the linear terms in spectral space, the data needs to be aligned in z-direction and distributed in x-direction because the matrices are local in x and global in z. However, we have to perform the inverse FFT before the inverse DCT before we can evaluate the nonlinear part in physical space. Therefore, we first perform an all-to-all communication to align in x-direction for the inverse FFT and then perform another all-to-all communication to align in z-direction for the inverse DCT.

A more optimized implementation would utilize a complex-valued DCT, which would require only one redistribution and substantially reduce communication cost as the inverse DCT could be performed before the inverse FFT, but we leave this for future work as our GPU port of `mpi4py-fft` does not support any DCT as of yet, which is why we compute it via FFT in the first place.

Preconditioning in the IMEX solver. The system matrix in the IMEX solver, $(M + \Delta t \tilde{q}_{mm} L)$ (see Equation 2.116 and Equation 2.117) is sparse, but lacks patterns that can be exploited by solver libraries by default. We follow techniques used in the `Dedalus` code [22] to precondition the systems into a form that allows solver libraries to efficiently compute the LU factorization. `Dedalus` is an elaborate framework for translating plain-text strings that describe PDEs into spectral discretizations.

Preconditioning is done in two steps, which we visualize in Figure 6.9 by showing the matrices at various steps for a problem of size $N = 3 \times 5$. The first step is to render the boundary conditions sparse. Recall that the boundary conditions are obtained by evaluating the basis functions of the in-going base at the boundary. In Chebychev base, we have $T_n(\pm 1) = (\pm 1)^n$, which means the Dirichlet boundary conditions fill n_x values horizontally each. We therefore exchange these for Dirichlet polynomials $D_n(x)$, which are defined via

$$D_n(x) = \begin{cases} T_n(x), & n = 0, 1 \\ T_n(x) - T_{n-2}(x), & n \geq 2. \end{cases} \tag{6.5}$$

Notice that $D_n(\pm 1) = (\pm 1)^n$ for $n \leq 1$ and $D_n(\pm 1) = 0$ for $n \geq 2$. The change from T_n to D_n is done via a right preconditioner.

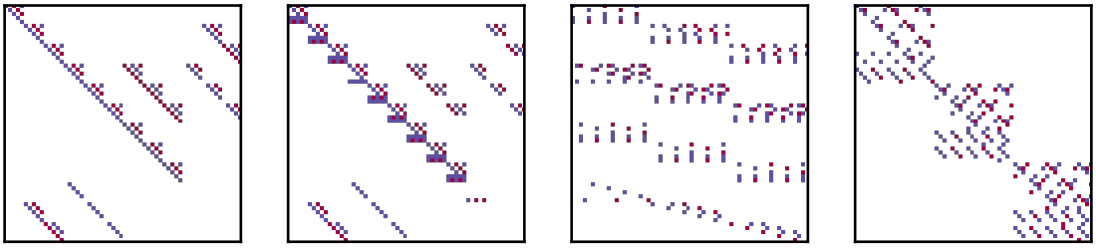


Figure 6.9: Various stages of the system matrix $M + \Delta t L$ for the Rayleigh-Benard problem with $N = 3 \times 5$ degrees of freedom. The colors mark the sign of the matrix entries, but this figure is only meant to convey the sparsity pattern. From left to right, we start with the matrix without boundary conditions, then we add the boundary conditions, apply Dirichlet preconditioning as right preconditioner and finally apply a left preconditioner, which reorders by mode rather than component, to arrive at a sparsity pattern of limited bandwidth that can be solved efficiently by solver libraries. The final left preconditioner is called "reversing the Kronecker product" in `Dedalus` [22].

The second step is called "reversing the Kronecker product" in `Dedalus` and amounts to a reordering by modes rather than components. After applying a suitable permutation matrix as a left preconditioner, the matrix has limited bandwidth and is block-diagonal. This has two advantages. First, the LU decomposition has the same (one-sided) bandwidth as the full matrix [59, Theorem 4.3.1], which means the LU decomposition of the preconditioned matrix is banded. And second, the $n_z \times n_z$ blocks can be decomposed in parallel on n_x tasks. Note that this is essentially the same concurrency as can be gained by distributing in x.

Linear solves in the IMEX scheme. We solve the linear systems in the IMEX scheme directly using LU decomposition. This has two advantages over iterative methods. First, the decompositions can be stored and reused. Second, we found the matrix to be poorly conditioned and iterative solvers to require preconditioners in order to obtain reasonable convergence.

Recall that the matrices we need to invert in the SDC sweeps are $M + \Delta t \tilde{q}_{mm} L$, with M and L as defined in Equation 2.116. Note that M here is the mass matrix and not the number of collocation nodes. We therefore need one factorization per node and only need to recompute when Δt changes.

We use `SuperLU` [102] as wrapped by `SciPy` to perform the actual decompositions on CPU. We parallelize by inverting the local system matrices that we use when evaluating the right hand side. In principle, linear solver libraries can exploit the block diagonal structure of the system matrix shown in Figure 6.9 and apply shared memory parallelisation. However, we turn this off because we found it to clash with MPI parallelism and to substantially increase the runtime.

6.4.2 Porting the Rayleigh-Benard convection implementation to GPU

We start by following the steps for GPU porting of the Gray-Scott equation in section 6.3.2, which is to say we swap `NumPy` and `SciPy` functions for `CuPy` equivalents by re-configuring

class attributes and we use the GPU port of `mpi4py-fft` from section 6.1. However, we were not able to satisfactorily port the LU decomposition.

Developing sparse solvers on GPU is not straightforward, because a lot of fine-grained memory access is required. Most approaches offload only certain operations during the factorization to GPU as opposed to the entire factorization [103, 150]. The `CuPy` equivalent of the `SciPy` function for LU factorization actually does not decompose on GPU at all, but calls the original `SciPy` function. Once the factorization is computed on CPU, the linear systems are solved using two linear solves of the lower and upper triangular parts [59, Section 3.2.8]. This is done with forward or backward substitution respectively and is run efficiently on GPU. When using fixed step size, the factorizations can be computed offline and the GPU-port runs efficiently, but with adaptive step size selection, computing the factorizations on CPU can mitigate speedup from GPUs.

There are some libraries for linear solvers on GPU, such as NVIDIA's `cuSOLVER` and `AMGX`. However, we found no straightforward way to implement them into our code because the former has no Python bindings and the latter lacks support for complex numbers. We leave implementation of a GPU-based sparse solver for future work.

Iterative solvers such as GMRES are implemented in `CuPy` and offer significant potential for GPU acceleration [100]. However, as discussed, the matrix is too poorly conditioned to use GMRES without preconditioner. Similar to complete LU decomposition, we were unable to find a GPU version of incomplete LU factorization for preconditioning that can be easily plugged into the Python code. Employing iterative solvers would likely be worthwhile as one of the advantages of SDC is that often few linear iterations are required in later SDC iterations. However, since identifying suitable preconditioners for these problems is beyond the scope of this thesis, we leave this for future work.

6.4.3 Parallel scaling

We now investigate the parallel scaling of the CPU and GPU implementations for RBC. We choose a configuration with Δt-adaptivity, four collocation nodes and four iterations. The interval under consideration is covered in twelve time steps, where the step size is changed sufficiently to require new factorizations three times. The factorizations are very expensive and dominate the run-time. We run experiments with the CPU implementation on JURECA-DC (section 2.9.2) and experiments with the GPU implementation on JUWELS booster (section 2.9.4).

Comparing CPU to GPU. We show parallel scaling of the RBC implementation on CPUs and GPUs in Figure 6.10. We find that the GPU implementation consistently performs worse than the CPU implementation for a given node count. As discussed in section 6.4.2, the factorizations are performed on CPU regardless of whether the remaining computations are performed on CPU or GPU. Therefore, the most expensive part of the computation is not accelerated by the GPUs. We do, however, employ 12 CPUs per GPU and allow thread parallelism with twelve CPUs when using GPUs, whereas we found thread parallelism to clash with the MPI parallelism when running exclusively on CPUs and therefore disabled it in that case. As shown in Figure 6.9, the matrix is block diagonal, which allows the solver to exploit concurrency in principle. Nevertheless, at a given node count, which means 64 CPUs or four GPUs, we find the CPU version to outperform the

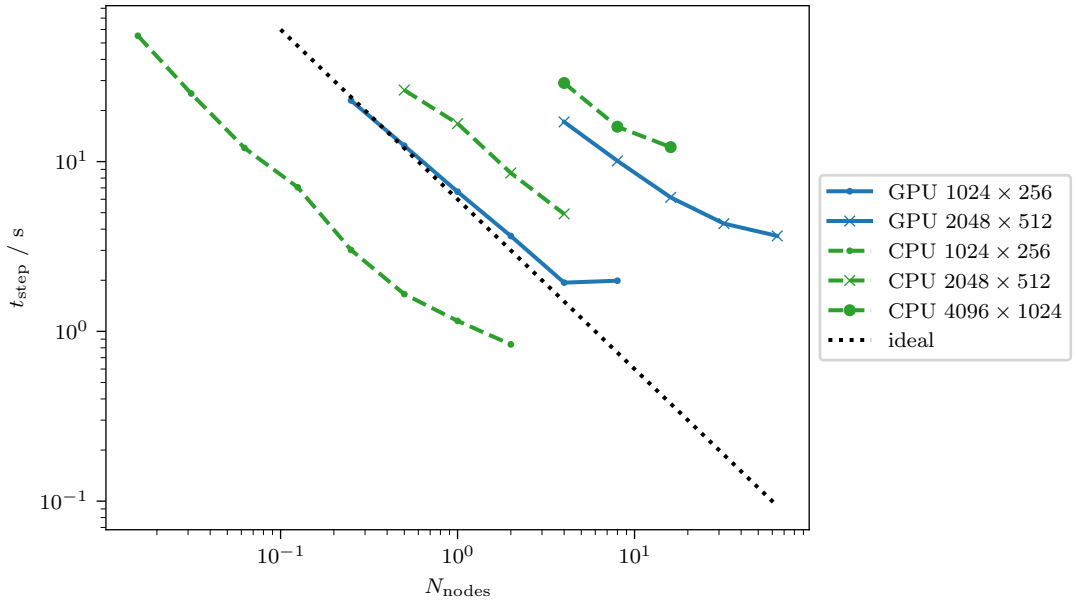


Figure 6.10: Parallel scaling of the RBC implementation on CPUs and GPUs with serial SDC via the mean time to compute one time step in SDC per node. The numbers in the legend indicate the resolution. Per node, the CPU implementation is much faster because the LU decompositions that make up a significant portion of the runtime are not accelerated by GPU.

GPU implementation by a significant amount for all resolutions we tested.

Looking at the scaling per task rather than per node in Figure 6.11, on the other hand, we find that the GPU implementation is faster than the CPU implementation per task. The CPU implementation is faster per node because the factorizations are distributed and parallelized more effectively. At the same number of tasks, the distribution is the same and the CPU implementation is not faster in the factorizations. On the other hand, other operations are accelerated on GPU, which result in the lower runtime per task on GPUs.

Next, we show data from the same scaling experiment, but the minimal time taken to solve a step rather than the average time in Figure 6.12. The time to solve a step varies greatly, depending on whether the factorizations are reused or computed from scratch. The minimal time is an example of a step where the factorization is reused and the entire computation is run on GPU. Here, the transforms are the most expensive operations and dominate the run time. We find that GPUs can solve time steps much faster than CPUs if no factorization needs to be computed. However, the space-scaling of non-factorization computations is very poor on GPUs at the resolutions we tested. This is because the computations already run in parallel on single GPUs automatically, as illustrated in Figure 6.3. Notice that the transforms scale rather well on CPU in spite of the extra transposes introduced by our implementation of DCT as FFT.

The reasons why we did not consider larger resolutions are twofold. First, the LU decompositions are already expensive and factorizing the n_z^2 blocks in the LU decomposition is a serial operation that requires $\mathcal{O}(n_z^3)$ FLOPS [59, Algorithm 3.2.1]. Second, storing the matrices of the problem discretization and the LU decompositions requires a lot of memory. For instance, when solving on a single GPU at a resolution of $N = 1024 \times 256$ less than eight matrix decompositions fit into the available 40 GB of memory next to everything else.

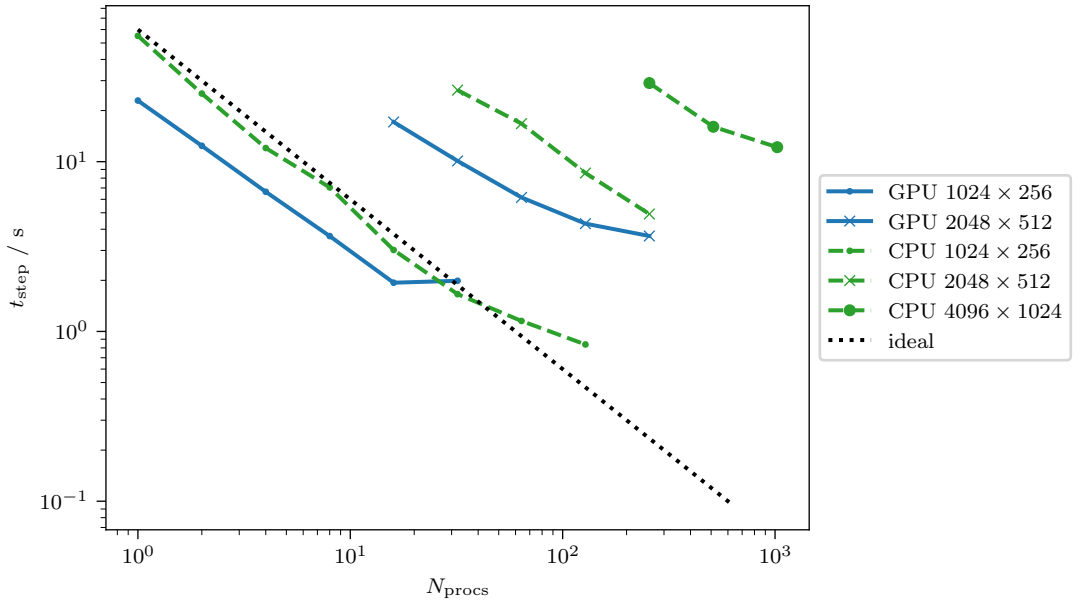


Figure 6.11: The same data as in Figure 6.10 is shown, but in mean time to compute one time step in SDC per task rather than per node. Per task, the GPU implementation is faster.

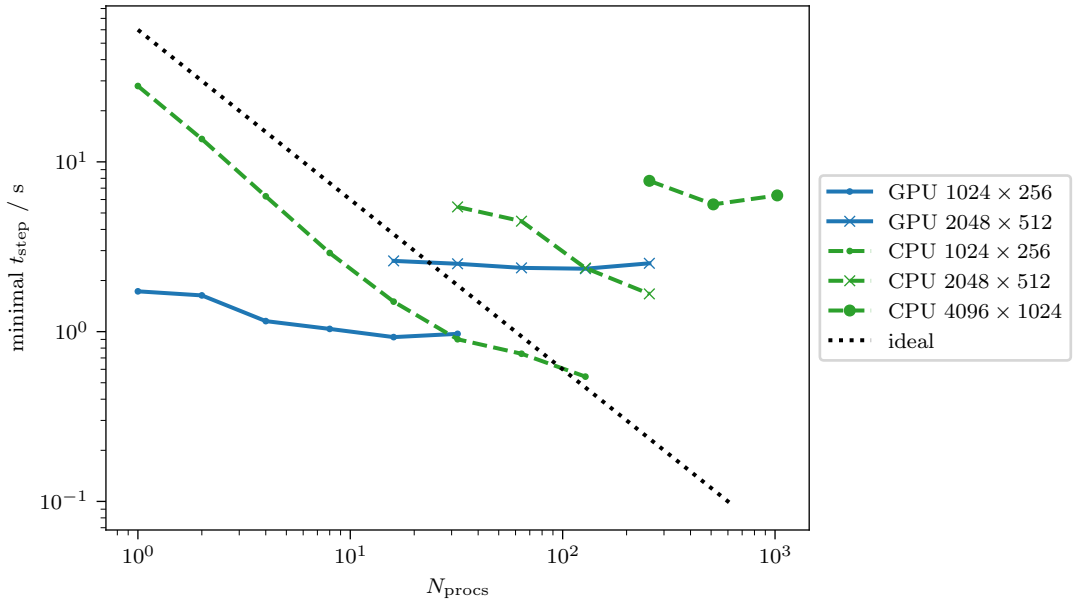


Figure 6.12: Data from the same experiment as in Figure 6.10 is shown, but instead of the average time to solve a time step with SDC, we show the minimal time over the considered interval per task. This shows how long it takes to solve a timestep when an LU factorization is reused. We find that a single GPU is much faster than a single CPU, but the GPU implementation does not scale well.

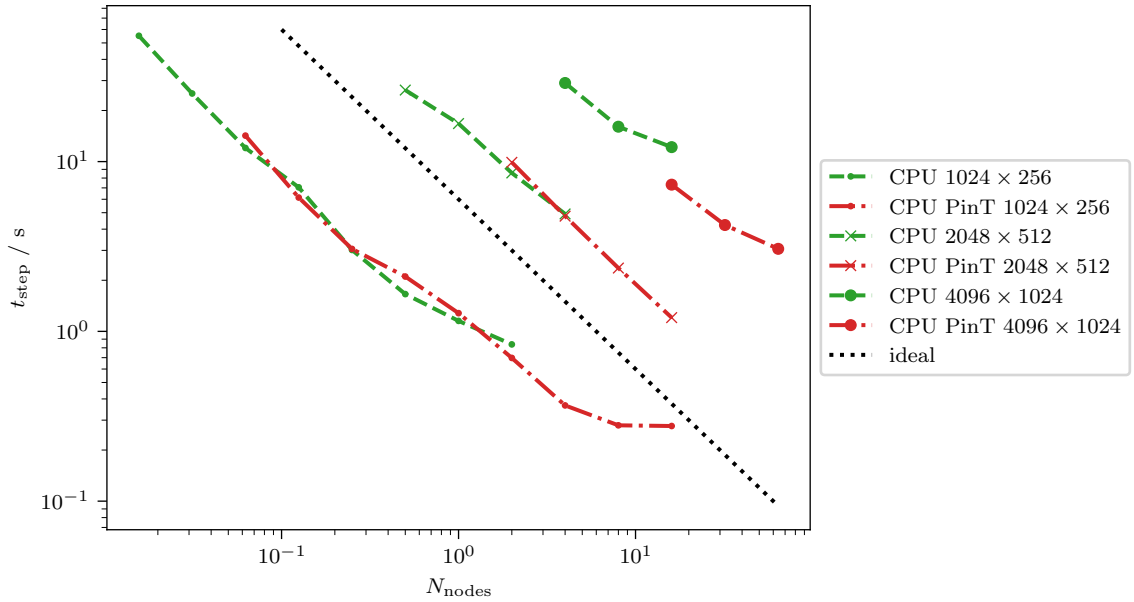


Figure 6.13: Scaling of the CPU RBC implementation with diagonal SDC and without, showing the mean time to compute one time step in SDC per node. The numbers in the legend indicate spatial resolution and "PinT" indicates that diagonal SDC was used. Adding diagonal SDC can extend the scaling capabilities substantially.

As one decomposition is needed per collocation node to do any kind of caching, this is very restrictive.

There is reason to speculate that if concurrency in the linear solves could be better exploited, the GPU implementation might outperform the CPU implementation per node as well. This could be done by splitting the local matrix into sub-problems that are solved independently and reassembled to the local solution. On CPUs, we were unable to achieve speedup with libraries mimicking thread parallelism, such as `multiprocessing`. On GPUs, on the other hand, single device parallelism is achievable in `CuPy` via multiple streams, but parallelizing the solver in this way requires a solver that runs on the device in the first place. Also, when the step size changes infrequently and the simulation is run for longer, the cost of computing the factorizations becomes small compared to the overall computational cost and running on GPUs can be far more efficient.

Diagonal SDC. Next, we investigate the capabilities of diagonal SDC to extend the scaling capabilities of both CPU and GPU implementations. We start with the CPU implementation in Figure 6.13. Just like with the Gray-Scott problem in section 6.3.3, we can extend the scaling capabilities significantly and achieve excellent speedup with space-time-parallel SDC. We confirm that this is also the case for the GPU implementation in Figure 6.14.

Recall from Figure 6.12, that the GPU implementation for operations other than the factorizations scales very poorly in space at the resolutions we tested here. As shown in Figure 6.15, diagonal SDC can provide speedup here because the operations are distributed differently. Therefore, diagonal SDC is not only a tool for extending parallelism beyond the limits of the spatial scheme, but can be beneficial even for serial-in-space implementations.

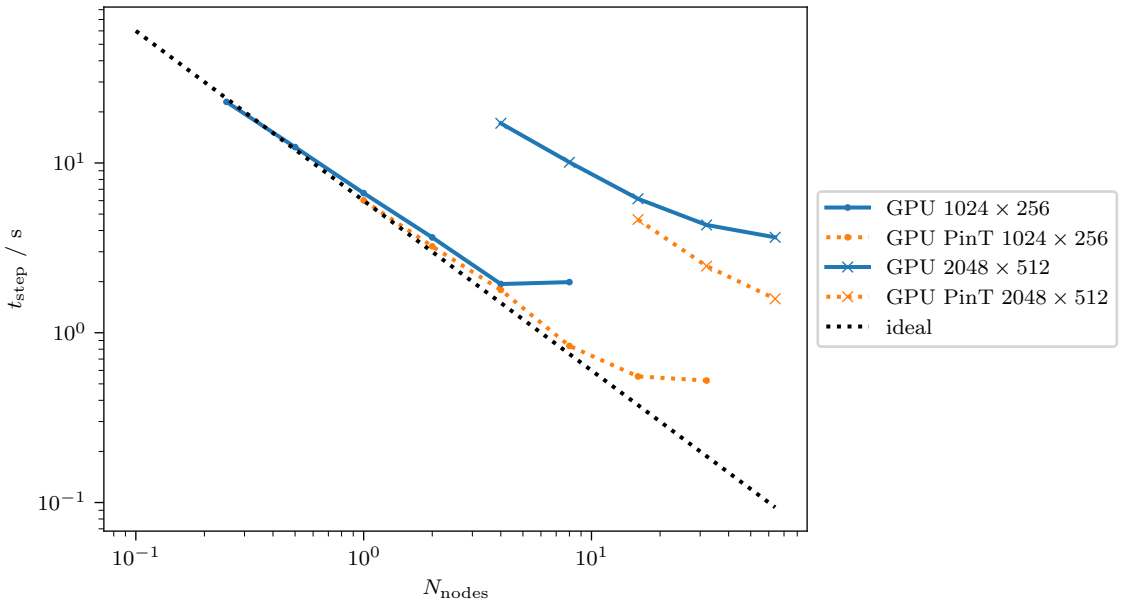


Figure 6.14: Scaling of the GPU RBC implementation with diagonal SDC and without analogous to Figure 6.13. Adding diagonal SDC extends the scaling capabilities just like in the CPU implementation.

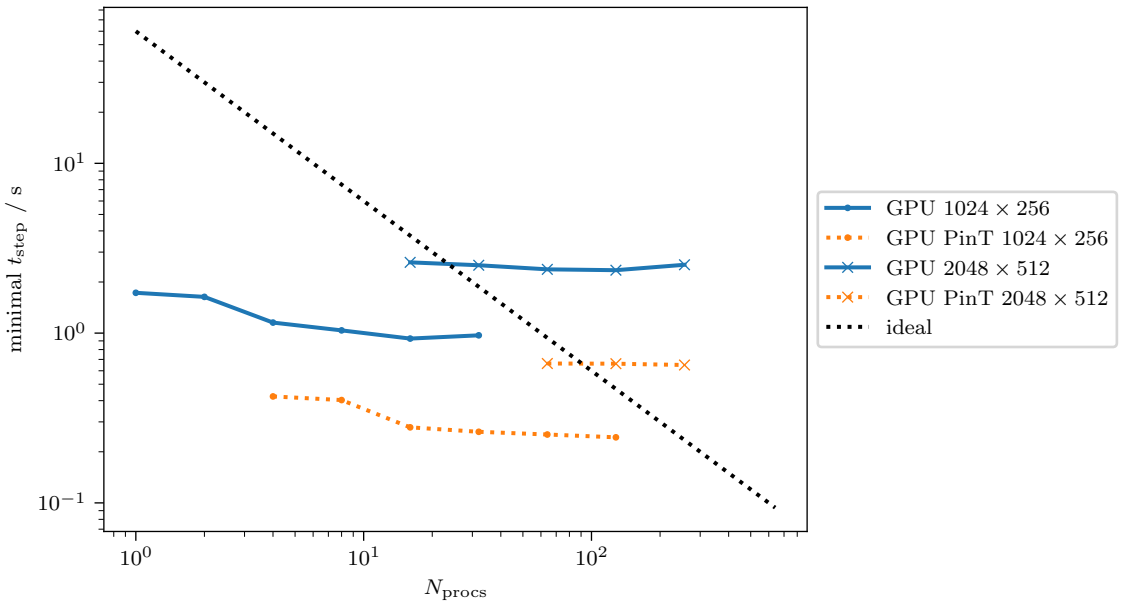


Figure 6.15: Shown is data from the same experiments as in Figure 6.14, but the minimal time to solve a step per task in order to exclude factorizations. We find that spatial scaling quickly saturates on GPUs at this resolution, but diagonal SDC can nevertheless provide speedup.

6.5 GPU agnostic code

The code we produced was written to be as GPU agnostic as possible. In many places, the exact same code is executed and the difference between CPU and GPU is purely in which numerical library is called. There are, however, a few points of note we want to discuss here.

Each kernel that is submitted to GPU incurs at least the kernel launch cost on the order of microseconds [61]. When many kernels are submitted, this cost accumulates and leads to imperfect utilization of the compute resources on the GPU. One way to mitigate this in Python is the use of CUDA graphs, as we do in section 6.1, where we found them critical to achieving competitive strong scaling with `mpi4py-fft` on GPUs. At the same time, the use of graphs makes it less straightforward to write clean GPU agnostic codes as there is no counterpart on CPU. The graphs also operate on a fixed portion of memory. Before the graph is executed, the input has to be copied to the specific part in memory. While not a large issue, practices like this contribute to cluttering the code in a fashion that is not typical of Python programming.

It is also noteworthy that the interface of `NumPy` and `CuPy` differs in a few ways. For one thing, copying any amount of data between CPU and GPU synchronizes the two, which should generally be avoided. Therefore, `CuPy` operations that return a single number, such as taking the maximum value of an array, return this number as a `CuPy` array of size one, which still resides on GPU. In a couple of places, for instance before writing such a number to file, we found it necessary to cast the numbers to floating point in order to transfer them to CPU. Again, this is not an issue per se, but contributes to cluttering of the code.

Another difference in the libraries is that `CuPy` exposes the pointers to the memory the data resides in to the user in some places. As memory management, including pointers, is done only to very limited extent by the programmer in Python, this seems an odd choice. For instance, NCCL communication functions require the user to pass the pointer. For an array named “a”, the user needs to pass “a.data.ptr”. This comes at no apparent benefit compared to the more streamlined way of passing `NumPy` arrays, which is typically done simply by passing the variable name. Perhaps we are overly critical here in pointing out minor details in the interface that do not cause significant issues in practice. However, the examples we provide are meant to convey that NVIDIA software often does not operate on the same level of maturity as CPU counterparts, making it hard to write clean, device agnostic, and fast code.

7 Large scale space-time-parallel simulations

After demonstrating that adaptive SDC as propsed in chapter 3 can be more computationally efficient than DIRK-type methods for stiff PDEs in chapter 4, can recover from a wide range of soft faults in chapter 5, and that space-time-parallel implementations of diagonal SDC can readily cater to modern HPC machines in chapter 6, we close with two "production runs" of Gray-Scott and RBC. These are meant to mimic the use of space-time-parallel adaptive diagonal SDC in practical applications, but we do not attempt to discover new physics here.

7.1 Gray-Scott

While Gray and Scott developed their equations for modelling chemical reactions, Pearson is credited with discovering the wide array of complex patterns that arise in this model when F and k are properly tuned [127]. These patterns are called Turing patterns, because Turing proposed that slight deviations and diffusion in chemical systems can lead to the formation of patterns observed in biology [156], such as the stripes of a zebra or the spots of a cheetah.

A wide range of numerical simulations of Turing patterns in the Gray-Scott equation in two-dimensions is available in the literature, e.g. [115], but we are not aware of similarly exhaustive work in three dimensions. With our production run, we show that our GPU port of `pySDC` is capable of investigating pattern formation in three dimensions at sufficient resolution by showing one specific example and discussing the observed patterns.

7.1.1 Setup

Problem setup. The initial conditions and parameters we use are a three dimensional variation of a setup shown in [155]. The parameters are

$$\nu_u = 2 \times 10^{-5}, \quad \nu_v = 10^{-5}, \quad F = 0.04, \quad k = 0.06. \tag{7.1}$$

See section 2.6.3 for the meaning of the parameters and the full equations. The initial conditions are made up of a superposition of blobs, each of which is formed as

$$\begin{aligned} B_u &= -\exp\left(-80\left((x-p_x+0.05s_x)^2+(y-p_y+0.02s_y)^2+(z-p_z+0.035s_z)^2\right)\right), \\ B_v &= +\exp\left(-80\left((x-p_x-0.05s_x)^2+(y-p_y-0.02s_y)^2+(z-p_z-0.035s_z)^2\right)\right), \end{aligned} \tag{7.2}$$

where p_i are the positions of the blobs and $s_i \in \{+1, -1\}$ are randomly determined signs, B_u is added to the u component and B_v is added to the v component. The background value is one for u and zero for v. We insert 1296 blobs in a domain of size 36^3 on a regular grid in the xy-plane and randomly in z-direction.

SDC setup. The SDC setup closely resembles the one from the scaling tests in section 6.3. We use four Gauß-Radau nodes and perform four iterations, which gives a fourth order method. We use MIN-SR-S for the implicit preconditioner and Picard for the explicit one. For step size selection, we use Δt-adaptivity with tolerance $\epsilon_{\text{TOL}} = 10^{-3}$.

Computational setup. In section 6.3, we showed that the Gray-Scott implementation gives excellent performance on GPUs. In Figure 6.7, we find excellent parallel efficiency with diagonal SDC up to 192 nodes or 768 GPUs. While the code scales beyond this at slightly reduced efficiency, we select this configuration with a resolution of $N = 2304^3$ on 192 tasks in space and four tasks in time on JUWELS booster for the production run.

7.1.2 Simulation results

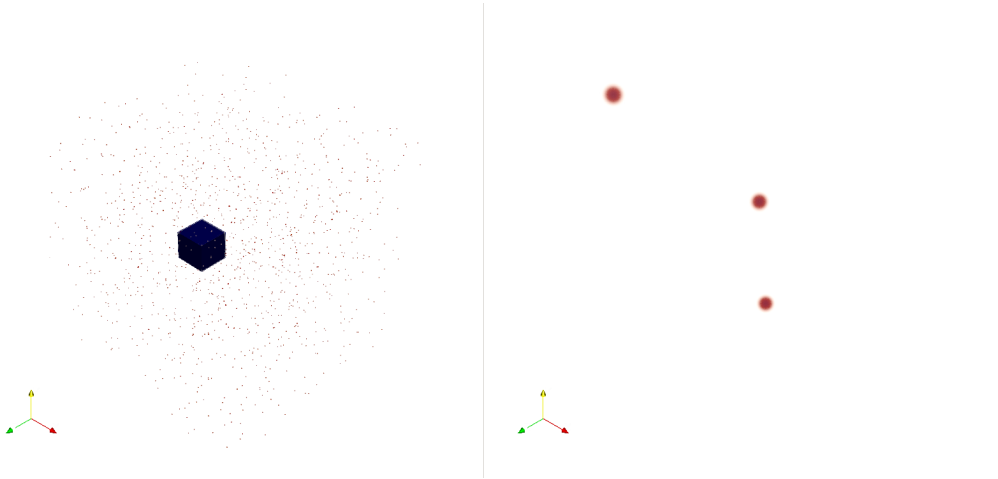
We now discuss how the solution evolves in the production run, the step size selection, and how the code ran on HPC.

Behaviour of the solution. During the course of the simulation, the blobs that make up the initial conditins expand and split up, leading to very complex formations. We visualize the initial phase of the simulation in Figure 7.1 and later points in time in Figure 7.2. For a video of the solution over time see [10, 3DGrayScottProductionRun.mp4].

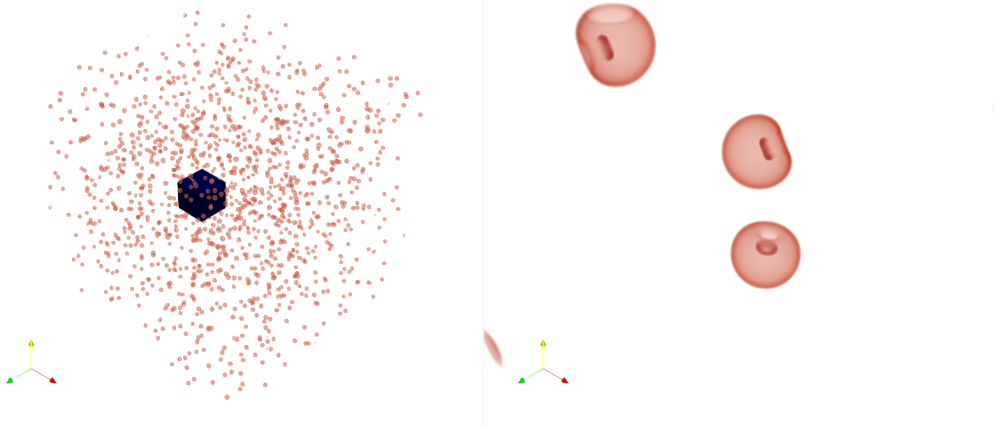
The zoomed panels in Figure 7.1 and Figure 7.2 show that the initially spherical blobs grow asymmetrically, which is due to the opposite signs in the u and v components in the initial conditions. As the spheres expand, they split off into further expanding shells and inwardly receding blobs, which remain connected via a tube. Over time, these mechanisms lead to a complex network of tubes, which spreads out over the spatial domain. Eventually, the blobs interact with neighboring blobs to join their tubes to one huge network. We show a slice in the yz-plane at the end of the simulation in order to illustrate the complex patterns that arise in Figure 7.3.

We run the simulation until we see significant interaction between the blobs. If the simulation were to be continued, the tube network would eventually extend across the entire domain.

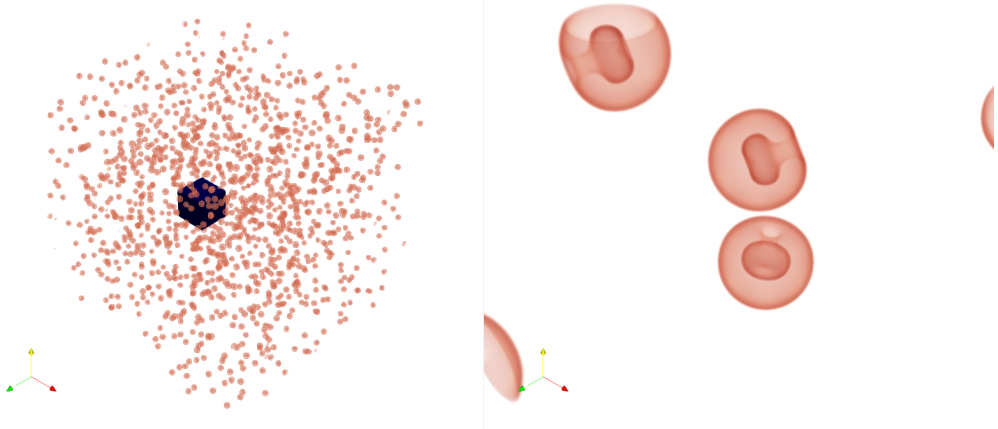
Step size selection. The timescale of the problem does not vary much. However, the initial conditions lead to rapid reactions between the two components, which requires small



(a) $t = 0$

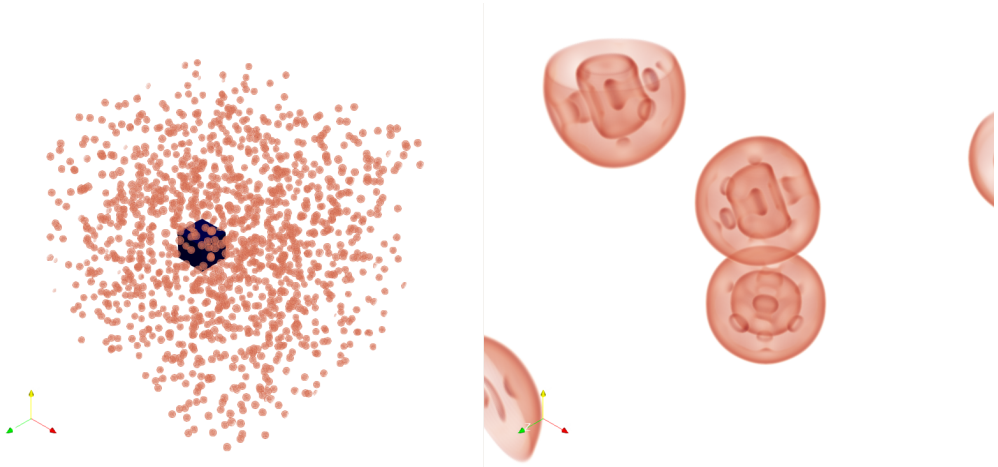


(b) $t = 553$

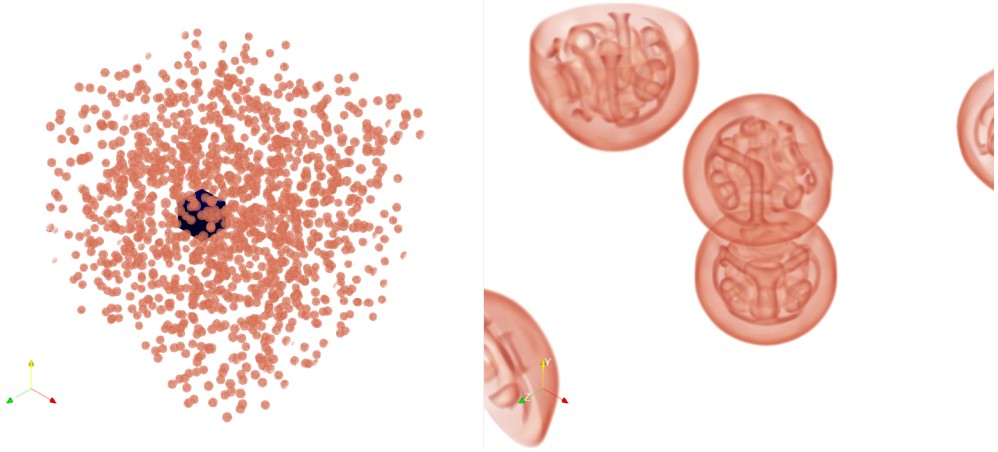


(c) $t = 1094$

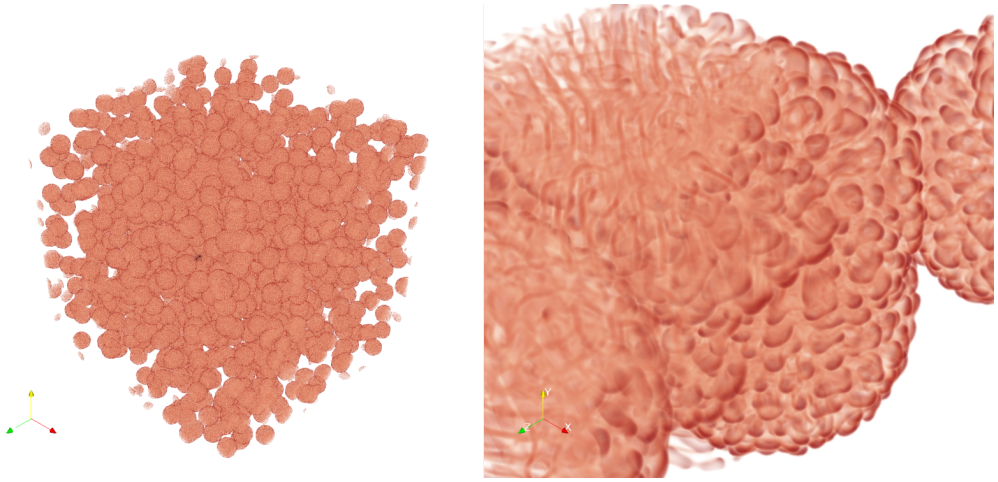
Figure 7.1: Evolution of the v component during the initial phase of the Gray-Scott production run. Shown are three points in time, with the entire spatial domain on the left side and a zoom on the right side. The zoomed area is indicated by the blue cube in the left panels. The colorscale is chosen to render all values below $v \approx 0.37$ transparent. See [10, 3DGrayScottProductionRun.mp4] for a video of the solution over time.



(a) $t = 1646$



(b) $t = 2162$



(c) $t = 4936$

Figure 7.2: Evolution of the v component in the Gray-Scott production run at later times. Shown are three points in time, with the entire spatial domain on the left side and a zoom on the right side. The zoomed area is indicated by the blue cube in the left panels. See Figure 7.1 for the preceding phase of the simulation.

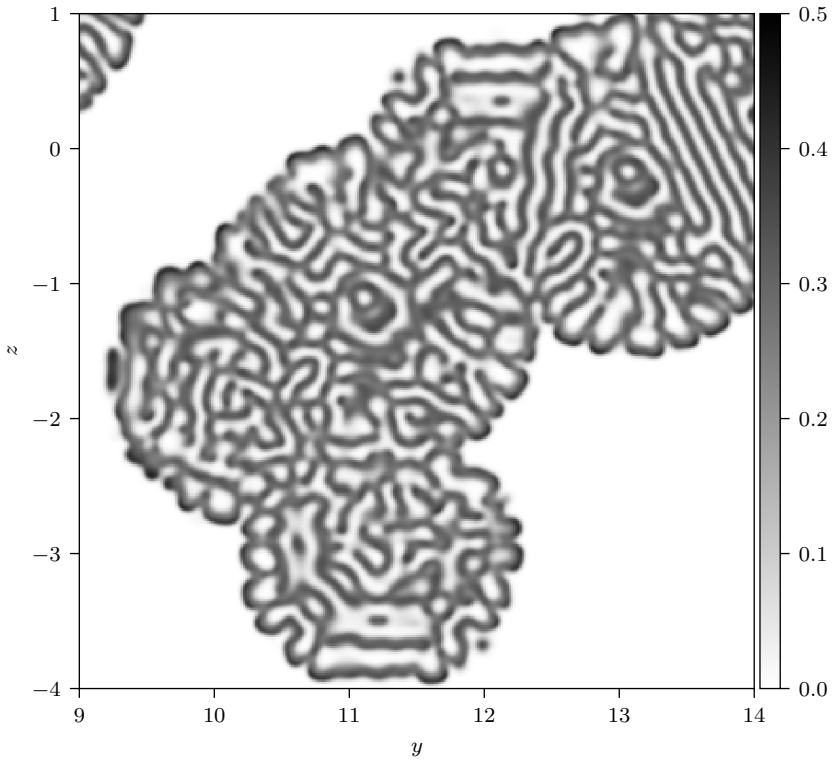


Figure 7.3: Two-dimensional slice at the end of the Gray-Scott production run. Multiple blobs interact to form a complex network of tubes.

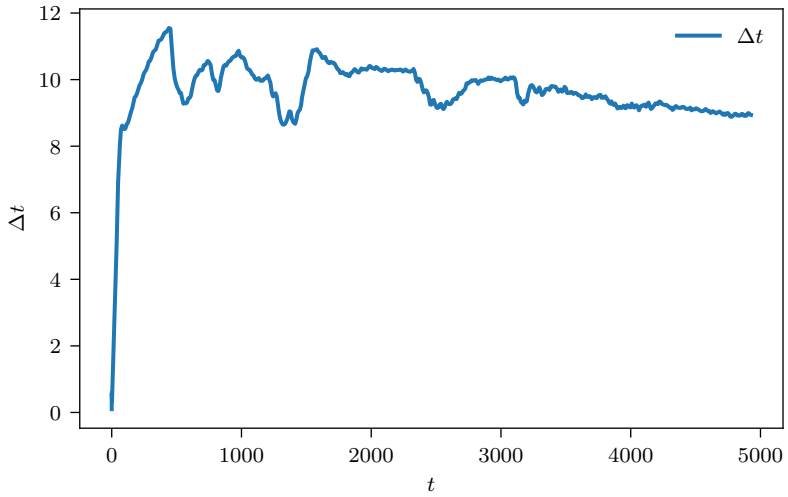


Figure 7.4: Step size distribution during the production run of Gray-Scott with Δt-adaptivity. In the beginning, step sizes on the order of 10^{-1} are required for resolving the rapid reactions in the initial conditions. However, the evolution quickly slows down as the blobs grow and the time-scale of the problem does not change dramatically from then on.

step size to resolve accurately. After a brief period of such rapid interaction, the evolution becomes more slow such that the step size is increased by one order of magnitude. We show the step sizes selected by Δt-adaptivity in Figure 7.4.

HPC capabilities. We find that `pySDC` runs similarly when using many GPUs as when using few GPUs. Loading the modules and starting the job behaves the same, no matter how many tasks are involved. We did, however, find that storing the simulation data on disk was a major bottleneck for the simulation. In our implementation, each task stores individual pickle files with its slice of the solution. Each of these files is approximately 973 MB in size, with the simulation producing a total of 17 TB of data. A parallel file format, ideally using a more efficient compression algorithm, is needed to properly run `pySDC` on HPC at scale. After the simulations were completed, infrastructure for parallel file formats using MPI was added to `pySDC` such that subsequent large simulations can run much more efficiently on HPC systems.

7.2 Rayleigh-Benard convection

While RBC is a rather simple setup, the model can be employed to study many phenomena occurring in the natural world. For instance, with large Prandl number, the setup can describe magma oceans during the formation of the Earth [83]. At low Prandl number, after adding a separate equation for magnetic fields, the equations governing RBC can be used to model the dynamo in Earth's core that causes its magnetic fields [110]. As the boundary layer in the Earth is orders of magnitude smaller than the entire domain, very large resolution is needed to properly resolve this problem.

It is not just the properties of the environment to be modelled that call for resolving structures covering multiple scales, however. While large scale features are prominent in

RBC, much of the heat transport occurs on smaller scales. With increasing amount of convection, the small scale flows become increasingly important [14]. For instance, in a three-dimensional simulation at Rayleigh number 10^8, it has been found that the large scale flow contributes only up to 30 % of the total heat transport [94]. Consequently, the resolution has to be increased in concert with the Rayleigh number in order to properly resolve all contributing flow patterns. Performing similar simulations to the ones we study here, at high resolution and massively parallel, is therefore relevant to investigating the high Rayleigh-number regime [106].

While some numerical analysis is done with low order codes, such as second order finite difference methods [148], there are also high order spectral element codes with semi-implicit time stepping in use [138]. The high-order codes in particular may benefit from (diagonal) SDC in the same fashion as our pseudo-spectral discretization.

7.2.1 Setup

We start by describing the setup we used in the production run before discussing the simulation results. Our experiments will exhibit flow patterns on multiple scales as required in actual production runs. Note that some of the simulation parameters are dependent on the behaviour during run-time, such that we already mix in some of the results in the justification for the parameters.

Problem setup. We largely use the same setup as in previous experiments with domain-size 8×2 and the same initial conditions. These are zero in all components except a linear temperature gradient, perturbed with random noise on the order of 10^{-3}. The only notable difference to previous setups is that we increase the Rayleigh number to $Ra = 3.2 \times 10^8$.

Since the noise in the initial conditions is non-smooth, the spectral methods struggle to resolve it properly. Starting out from these initial conditions leads the adaptive step size selection to choose very small step sizes and perform many restarts until the solution is sufficiently smooth. To mitigate, we smooth the initial conditions by solving five IMEX Euler steps with $\Delta t = 0.1$ before starting the time integration with step size adaptive SDC.

SDC setup. We select Δt-k adaptivity, as this proved to be well suited to combination with diagonal SDC in chapter 4. We use four Gauß-Radau nodes with up to 16 iterations, which results in an order seven method. We use $\epsilon_{\text{TOL}} = 10^{-5}$, $r_{\text{TOL}} = 5 \times 10^{-6}$ and also an increment tolerance of 5×10^{-6} for the SDC iteration stopping criterion. The preconditioners are MIN-SR-S and Picard.

We find that the IMEX solver is stable only up to $\Delta t \approx 10^{-3}$ from the point that turbulence develops. We operate very close to this limit with adaptive step size selection, as this limit is quite restrictive. We add generous rounding of Δt to the step size controller in order to keep the number of factorizations to a minimum. We round by reducing the number behind the floating point in the step size to 0 or 5. For instance, we round 2.67×10^{-3} to 2.5×10^{-3}. Otherwise, we employ the same step size controller from chapter 4, meaning we only change the step size when $\|\Delta t_{\text{opt}}/\Delta t - 1\| > 1$ or if the step is restarted and use safety factor $\beta = 0.5$.

Adding the rounding further decreases the step size. We therefore accept to perform even more right hand side evaluations than strictly needed in order to keep the number of

factorizations low. Recall from section 6.4 that the computational cost in the right hand side evaluations is dominated by the transforms, which scales with $\mathcal{O}(n \log n)$, whereas the cost in the factorizations scales with $\mathcal{O}(n_z^3)$. Hence, the larger the resolution we choose, the more we should shift the balance towards fewer factorizations and more transforms.

Computational setup. We use a resolution of $N = 4096 \times 1024$ with 4096 tasks on JURECA-DC (section 2.9.2), where we use four tasks in time and 1024 tasks in space. This is the largest resolution and most distributed configuration that we tested in the experiments on parallel scaling on CPUs in Figure 6.13.

7.2.2 Simulation results

We now describe the results from the RBC production run, in terms of the solution, the step size selection and how the code runs on HPC systems.

Behaviour of the solution. We show plots of the temperature at various points throughout the simulation in Figure 7.5. For a video of the solution over time see [10, RBCProductionRun.mp4]. The large scale behaviour is similar to the lower resolution and lower Rayleigh number setup shown in Figure 2.14. We see that the small perturbations in the temperature grow to plumes rising to the top from approximately $t = 10$ to $t = 15$. When the plumes reach the top and are pushed back down by further rising plumes, irregular flow patterns arise. Finally, the flow settles into a pseudo-stationary rolling motion. However, we stop the simulation before actually reaching the pseudo-stationary regime at $t = 26$. In contrast to the earlier example in Figure 2.14, very small scale features and eddies can be observed at this high resolution and quite complex flow patterns emerge at this high Rayleigh number.

Step size selection. In the beginning phase, before the plumes develop, we can choose large step sizes. Later on, when significant convection develops, we require smaller step size because the solution evolves faster and because of stability restrictions in the IMEX solver. We show the resulting step size distribution in Figure 7.6, where we can clearly identify the different stages of the simulation in the step size. We initially select a tolerance for adaptive step size selection of 10^{-4}, which results in large step sizes on the order of 10^{-1} up to $t \approx 10$. As the plumes develop, the step size is quickly refined until settling to $\Delta t \approx 10^{-3}$ when the flow has reached the irregular state. Afterwards, it stays there for the rest of the simulation.

We note that the stability limit of the IMEX solver is only slightly larger than 10^{-3}. The adaptive step size selection is not aware of this limit and hence attempted larger step sizes at $\epsilon_{\text{TOL}} = 10^{-4}$. Then, SDC did not converge and the step was restarted with a quarter of the step size. Adaptive SDC with these parameters is still capable of advancing in time, but the scheme is not efficient. Therefore, we stopped and restarted the simulation with decreased $\epsilon_{\text{TOL}} = 10^{-5}$, which results in similar step sizes but fewer restarts. Stability limits can be easily incorporated into adaptive step size selection, if they are precisely known, which is not the case here.

We show the resulting work in number of matrix factorizations and right hand side evaluations in Figure 7.7. Less than 100 factorizations are required to cover the diffusion

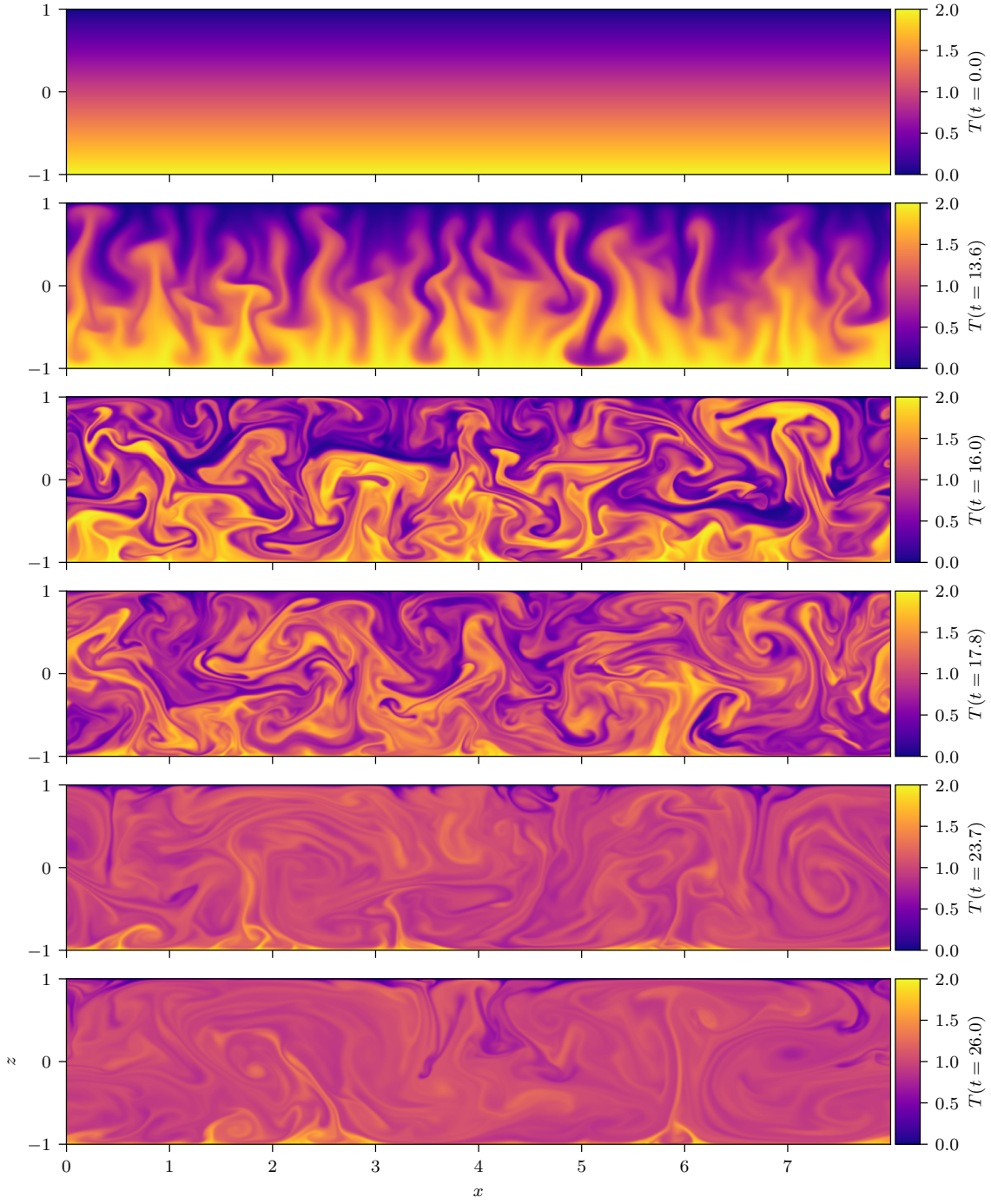


Figure 7.5: Temperature profile in the Rayleigh-Benard convection production run with Δt-k-adaptivity over time. With the large resolution of $N = 4096 \times 1024$, small eddies are visible in the turbulent flow. We use diagonal SDC with four tasks in time and 1024 tasks in space for a total of 4096 CPUs on JURECA. See [10, RBCProductionRun.mp4] for a video of the solution over time.

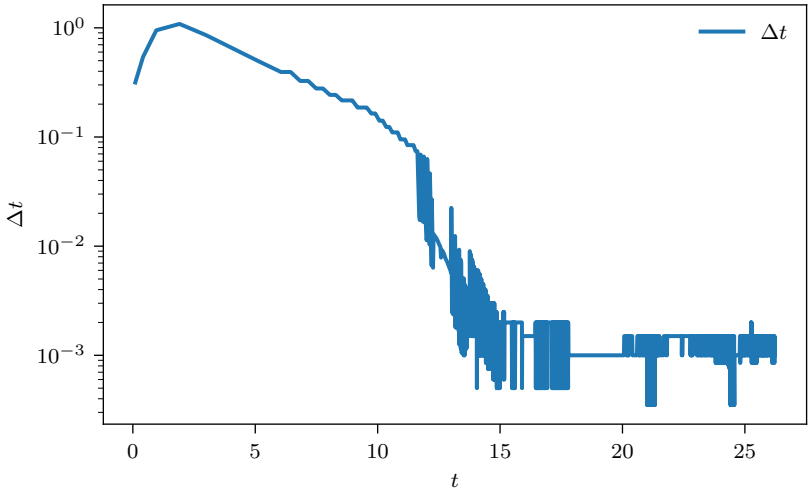


Figure 7.6: Step size distribution during the production run of RBC with Δt-k adaptivity. Step sizes of restarted steps are not shown. Up to $t \approx 10$, the flow is slow and the step sizes are relatively large. As turbulence develops, the step size is refined by more than two orders of magnitude. When turbulence is properly under way, the IMEX solver is unstable for Δt larger than approximately 5×10^{-3} for the remainder of the simulation.

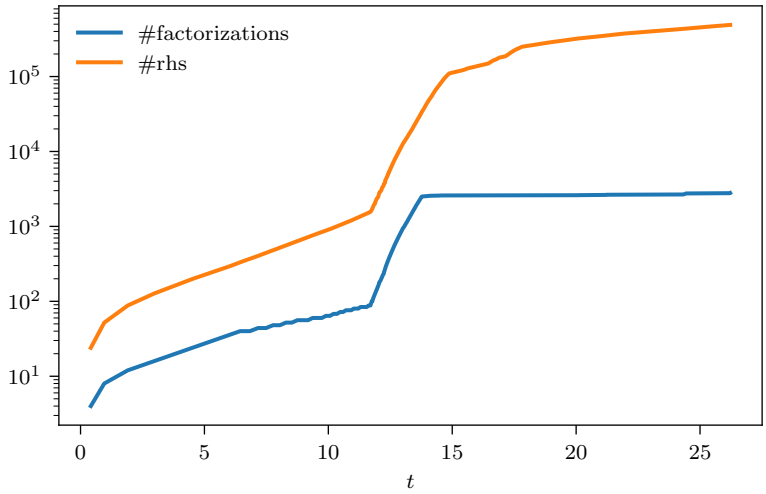


Figure 7.7: Work required to achieve a certain point in time in the RBC production run. After a few large time steps, the step size is decreased during the development of turbulence. Afterwards, the stability limit of the IMEX solver keeps the step size within a narrow range, such that the LU decompositions are efficiently reused.

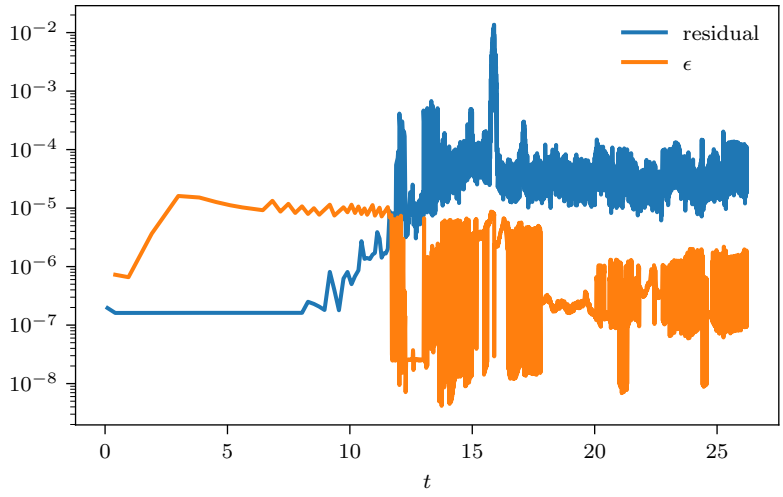


Figure 7.8: SDC residual and error estimate used in step size selection throughout the RBC production run. The large gradients at the boundary prevent the SDC residual from reaching arbitrarily small values. We reduced the tolerance for adaptive step size selection from 10^{-4} to 10^{-5} after the development of turbulence in order to keep the step size below the stability limit without restarts. The error estimate is significantly below the tolerance due to the step size controller and safety factor $\beta = 0.5$.

dominated initial phase. As turbulence develops, the step size is refined which requires a lot of LU decompositions. During this phase, approximately 2000 factorizations are computed. Once the step size reaches the relatively stable regime with $\Delta t \approx 10^{-3}$ from $t \approx 15$, we require virtually no new factorizations because the step size controller returns step sizes for which factorizations are already available most of the time. The number of right hand side evaluations evolves similarly, but it rises continuously in the regime where the step size has settled.

Finally, we show the error estimate used in step size selection and the residual of the collocation problem during the run in Figure 7.8. We find error estimates significantly below the tolerance value due to the elaborate step size controller, which selects smaller than optimal step sizes in order to keep the number of matrix factorizations at bay. In particular, we use a small safety factor of $\beta = 0.5$ in order to reduce the number of restarts. Since we only change the step size when the relative change is large, the error estimate oscillates heavily when the step size does change. While this results in unnecessarily high resolution in some time steps, we chose to accept this in order to reuse the matrix factorizations as much as possible, which is crucial to obtaining an efficient scheme.

As discussed in section 2.5.11, the SDC residual does not reach arbitrarily small values when the perturbations due to the treatment of the boundary conditions are non-negligible. We verify that the perturbations are small relative to the order of magnitude of the solution by showing the residual in Figure 7.8. During most of the simulation, the residual oscillates around 10^{-4}, with only occasional larger values.

HPC capabilities. We showed that our code is capable of high-resolution, massively parallel simulations. Both large and small scale features are resolved, which is crucial in actual production runs. However, the simulation is relatively expensive and the code is

limited in its HPC capabilities.

We have discussed various shortcomings of our implementation in section 6.4, such as manually expressing the DCT via FFT and performing more transposes than theoretically needed. The major limitation, however, is solving the linear systems in the ultraspherical discretization using a direct solver, which requires $\mathcal{O}(n_z^3)$ floating point operations. This heavily limits the practicability of Chebychev or ultraspherical discretizations at high resolution. It can be resolved, for instance, by transitioning to spectral element methods, where the domain is split into multiple cells. Then, smaller sub-problems are solved concurrently with spectral methods and the global solution is obtained by enforcing coupling conditions in a separate step, which is serial but inexpensive [18, Chapter 22].

Alternatively, iterative solvers may converge quickly if suitably preconditioned, even for large resolution. As iterative solvers have been successfully accelerated by GPUs within `pySDC` for other problems [100], and SDC is known to provide good initial guesses for iterative solvers between iterations [147], they could likely be used to obtain a very efficient implementation. We leave implementation of iterative solvers or of a spectral element discretization for future work. We do expect, however, that the concurrency afforded by diagonal SDC is useful, regardless of these details of space-discretization.

Note that all previously published demonstrations of PinT or space-time speedup that we are aware of use only tens of tasks, employ more simple diffusion dominated or linear problems, or exhibit poor parallel efficiency when scaling in time [37, 53, 116, 117, 124, 165]. Our implementations of diagonal SDC and spectral methods, on the other hand, exhibit excellent parallel scaling in both space and time for a complex and practically relevant benchmark problem up to thousands of tasks.

8 Conclusion

8.1 Summary

The numerical simulation of time-dependent problems on massively parallel machines will eventually require parallelisation in the time direction in order to make efficient use of the ever growing hardware. Spectral deferred correction (SDC) is a time-integration method that offers multiple routes for time-parallelism, which can make it preferable to the similar and more common diagonally implicit Runge-Kutta methods (RKM). In this thesis, we have developed generic adaptive extensions for SDC, implemented complex benchmark problems parallel in space and time, and measured performance in a range of experiments. Here, we repeat the key results that we obtained in the process.

Developed step size adaptive extensions for SDC. In chapter 3, we developed two adaptive extensions for SDC along the lines of embedded RKM. The first we call Δt-adaptivity, where the iteration number is kept constant, but the step size is adapted based on an estimate of the local error. The error estimate is obtained by subtracting two consecutive iterates and requires the order to increase by one with each iteration. This is typically the case in SDC, but has to be confirmed numerically for a given combination of preconditioner and problem. The second algorithm, Δt-k-adaptivity, selects both the step size and the iteration number adaptively and employs an estimate of the error of the converged collocation problem. This makes it more flexible with respect to SDC preconditioners and inexactness.

Demonstrated computational efficiency of adaptive SDC. In chapter 4, we measured computational efficiency of adaptive SDC via work-precision diagrams, relating wall-time to achieved accuracy for a range of problems and SDC schemes.

First, we confirmed that both step size adaptive algorithms finely adjust the accuracy to the requirements of the problem during runtime and can increase computational efficiency over fixed step size SDC schemes. Afterwards, we measured parallel-in-time (PinT) speedup of adaptive SDC with diagonal SDC, Block Gauß-Seidel SDC (BGSSDC) and a combination of both. We found excellent parallel performance with diagonal SDC with both Δt-adaptivity and Δt-k-adaptivity for the PDE benchmarks Gray-Scott and

Rayleigh-Benard convection (RBC). BGSSDC also gave good speedup for the Gray-Scott equation with Δt-adaptivity, but not with Δt-k-adaptivity or for RBC.

Then, we compared performance of parallel-in-time adaptive SDC against state-of-the-art RKM. For the Gray-Scott example, BGSSDC with Δt-adaptivity was the most efficient scheme when coarse accuracy is desired, whereas Δt-k-adaptive diagonal SDC was the most efficient scheme for achieving high accuracy. Notably, the embedded additive RKM we compared against was never more efficient than SDC. In RBC, the treatment of the incompressibility constraint requires RKM to be stiffly accurate. Few suitable stiffly accurate additive RKM are available in the literature and the best performing method we could find was a third order additive RKM. Third order SDC outperformed this method, with Δt-k-adaptive diagonal SDC giving the best performance for any accuracy that we tested. However, the flexibility of SDC allows to easily increase the order and fifth order SDC outperformed the additive RKM by wide margins. On the other hand, for the ODE benchmarks van der Pol and Lorenz attractor, we found the RKM reference methods to be more computationally efficient, suggesting that SDC is primarily a good method for PDEs.

Demonstrated resilience capabilities of adaptive SDC. In chapter 5, we measured the ability of adaptive SDC to recover from faults in the form of transient bit flips. Adaptive methods measure the accuracy during runtime and increase the accuracy if needed. If the accuracy is insufficient due to a fault, adaptive schemes will automatically attempt to correct the fault by restarting or by continued iteration. We also developed an SDC specific version of the dedicated resilience strategy Hot Rod for comparison.

We performed experiments where we flipped random bits in the solution variable and accepted the fault as recovered only if the global error at the end of the simulation was not increased beyond 10 %. Using the chaotic Lorenz attractor problem, we investigated the ability of adaptive SDC to correct faults depending on where in the SDC algorithm they occur. Adaptive SDC could protect against soft faults almost perfectly and on par with Hot Rod, except for faults targeting the initial conditions, which require further protection. Unlike Hot Rod, adaptivity does not add overhead, but is actually a mechanism for reducing computational overhead, making it an attractive resilience strategy.

Measured space-time parallel scaling of adaptive SDC on HPC. In chapter 6, we discussed space-time-parallel implementations of the Gray-Scott and RBC benchmarks in Python within the prototyping library `pySDC`. We focused on GPU implementations as these account for the majority of compute power in modern HPC machines. For time-parallelism, we used diagonal SDC, since this performed very well in chapter 4.

We discretized both problems in space using Fourier and ultraspherical spectral methods, which rely on fast Fourier transforms (FFTs). Therefore, we first ported the library `mpi4py-fft` for distributed FFTs to GPUs. In our tests, `mpi4py-fft` performed significantly better on GPUs compared to CPUs and performed almost as well as the compiled and optimised alternative `cuFFTMp` on GPUs, which does not have Python bindings.

We then ported the Gray-Scott and RBC implementations within `pySDC` to GPU. With Gray-Scott, we found excellent performance on GPUs and were able to scale up to the entire JUWELS booster machine at 37 % parallel efficiency when combining parallelism in space and time. Diagonal SDC was crucial in extending scaling beyond the limits of

space-parallelism. Our RBC implementation relies on a sparse LU decomposition, which we were unable to efficiently port to GPU. However, we found good parallel scaling and again used diagonal SDC successfully to scale beyond the limits of spatial parallelism. Demonstrations of space-time-parallel speedup on this scale for such complex problems is unprecedented in the literature to the best of our knowledge.

Finally, we showed large space-time-parallel simulations of Gray-Scott and RBC that resemble production runs in chapter 7. These serve to show that the adaptive extensions and diagonal SDC are useful in practice and not mere theoretical considerations.

8.2 Future work

In preparing this thesis, many further questions arose which exceed the scope of this work. We now outline some directions for future research.

Use adaptive SDC in PFASST. The parallel full approximation scheme in space and time (PFASST) is an algorithm that can run massively parallel by solving blocks of time steps in parallel. In this thesis, we have used adaptive SDC in Block Gauß-Seidel SDC (BGSSDC), which is a special case of PFASST. Therefore, we expect the step size selection mechanisms to work with more elaborate PFASST schemes as well, which may be able to provide more speedup than BGSSDC.

Furthermore, in our experiments with BGSSDC, we only used the same step size across all steps in the block. By estimating the local error on all steps in the block, it is possible to make a guess for the evolution of the ideal step size in the next block as well. However, it is unclear if this can improve computational efficiency and performance is likely very problem dependent.

Use adaptive SDC for DAEs. Differential algebraic equations (DAEs) often cause similar issues for numerical integration as stiff PDEs and order reduction is sometimes observed [6]. This can pose challenges for the adaptive step size selection we developed in this thesis, which requires to know the order of accuracy. Therefore, adaptations to the error estimates may be needed when applying the adaptive algorithms to complex DAEs.

Note that RBC is a DAE, because of the incompressibility constraint. Here, we were able to treat the algebraic equations simply by using a quadrature rule that includes the end-point of the interval and we did not observe order reduction. However, problems with more complicated algebraic constraints can require more elaborate treatment [77].

Use more elaborate step size controllers and real-world applications. For the RBC problem, we found that always taking the largest possible step size is not necessarily the most efficient scheme. This is also the case for many real-world problems. It would be interesting to benchmark the performance of adaptive SDC with PID controllers, for instance, and within an actual production code.

Combine resilience strategies and use more realistic fault insertion. We found that adaptive schemes can recover from many types of faults very efficiently while simultaneously reducing computational cost, but identified certain classes of faults that they

cannot correct. Other resilience schemes can protect even against these kinds of faults, but often often add significant overhead in terms of computation or memory. It would be interesting to combine various resilience schemes that each selectively protect against certain types of fault and that add as little overhead as possible.

Ideally, such strategies would be investigated with a more realistic fault insertion mechanism as well. We restricted the discussion to transient bit flips in solution variables, but more realistic experiments would also consider, for instance, bit flips in operators. This is another type of fault that adaptive SDC cannot protect against. However, SDC, and particularly BGSSDC, has some redundancy that can be leveraged for resilience, e.g. [144].

Use other space-discretizations in space-time-parallel experiments. The global spectral methods we used for spatial discretizations of PDEs are not without limitations. In particular, the linear solves in the RBC example were very restrictive for scaling the problem. Other discretizations such as finite element or spectral element offer more potential for parallelization in space. Combining these with time-parallel SDC could enable to scale to even more tasks than in our experiments.

Also, we found that space-time-parallelism with diagonal SDC can improve parallel efficiency over space-only-parallelism with global spectral space discretizations because the communication pattern is shifted. As other methods use different communication patterns, it would be interesting if similar effects can be observed.

Further GPU optimisation. While performance considerations often take a back seat in Python programming, codes can be optimised for GPUs to some degree. Most importantly, synchronisation between GPU and CPU should be avoided as much as possible, which can be done easily. Also, one should take care to keep the number of individual kernels low, as every kernel incurs a launch cost, which accumulates when many small kernels are launched. In Python, this is less straightforward to do, but not impossible. We have discussed the use of CUDA graphs within `mpi4py-fft`, which can be used to launch multiple operations within the same kernel. However, these make device-agnostic programming more difficult and cannot be stacked. For instance, we could not have recorded a CUDA graph for an SDC step that uses the CUDA graph within `mpi4py-fft`. Users can write CUDA kernels and use them within `CuPy`, but this requires to write actual CUDA code, rather than Python code. For lack of a straightforward way to reduce the number of kernel launches in a device-agnostic way, we did not substantially optimize `pySDC` in this regard. It would be interesting to investigate GPU optimisation and measure further performance gains, particularly with vendor-portable methods.

8.3 Concluding remarks

SDC remains a niche method for time-integration with other RKM being a lot more common in production codes. However, this thesis has demonstrated that SDC can outperform state-of-the-art RKM for some problems. If the problem at hand is a complicated stiff PDE, SDC should certainly be considered. SDC affords great flexibility and schemes can be tailored to the problem by choosing appropriate splitting, inexactness, or time-parallelization, for instance. Finding the SDC scheme that is most efficient for a given problem is best done with numerical experiments, for which the prototyping library `pySDC` proved to be very

convenient. We hope this work and our implementations can encourage experimentation with SDC and aid the development of future space-time-parallel codes.

Reproducing the results. For instructions on how to reproduce the results shown in this thesis, please consult the project sections in `pySDC` version 5.6 [143]. Experiments from chapter 4 and chapter 5 are part of the resilience project at `pySDC/projects/Resilience/README.rst`. Experiments from chapter 6 and chapter 7 are part of the GPU project at `pySDC/projects/GPU/README.rst`.

Acknowledgements

I am enormously grateful to my main supervisor Robert Speck who taught me a lot about research software engineering, a bit about maths, who was always supportive and made time, and who sent me on many exciting trips. As I came into this project with virtually no experience and knowledge about programming or numerical methods, a lot of patience was required of him, which, luckily, there was ample of. Robert allowed me great freedom within his code `pySDC` and I learned a lot during all the mistakes I made in the process. Also, I owe great gratitude to my second supervisor Daniel Ruprecht, who attempted to teach me about setting clear goals and writing concisely, even though I still have a lot of practicing to do. His frank comments were always very helpful and provided good guidance. Daniel and Robert's leadership style played a major role in making my Ph.D. experience enjoyable and I know it cannot be taken for granted.

Furthermore, I am very grateful for the discussions with Thibaut Lunet and Sebastian Götschel, who contributed a lot to the contents of this thesis. I particularly enjoy indefinitely ongoing quarrels about camel case and vim with Thibaut. I found the experience of being part of two groups very positive because it increased the number of discussions somewhat and I always enjoyed my visits to Hamburg. I want to thank the NVIDIA Application Lab at Jülich, specifically Markus Hrywniak, for help on programming GPUs.

I also want to thank my office mates in Jülich for making my time here very fun by sharing bike rides in the commute, playing ping pong in the basement, and standing in line at Seecasino together. Special thanks to Sina for spending as much time on making the hat as I did on writing the thesis. I want to thank my friends and family, especially Max, Malte, Hanno, Eddi, Christine, and Sara, for all the support and help they provided, my wife Julia for making the difficult times that are inevitably part of writing a thesis bearable, and my daughter Pia for moral support during the defence and for enriching my life away from keyboard in countless ways.

Funding. We gratefully acknowledge computing time granted on JUSUF, JURECA, JUWELS and JUWELS Booster through the CSTMA project at Jülich Supercomputing Centre. We thankfully acknowledge funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955701. The JU receives support

from the European Union's Horizon 2020 research and innovation programme and Belgium, France, Germany, and Switzerland.

Bibliography

[1] Agullo, E., Altenbernd, M., Anzt, H., Bautista-Gomez, L., Benacchio, T., Bonaventura, L., Bungartz, H.J., Chatterjee, S., Ciorba, F.M., DeBardeleben, N., Drzisga, D., Eibl, S., Engelmann, C., Gansterer, W.N., Giraud, L., Göddeke, D., Heisig, M., Jézéquel, F., Kohl, N., Li, X.S., Lion, R., Mehl, M., Mycek, P., Obersteiner, M., Quintana-Ortí, E.S., Rizzi, F., Rüde, U., Schulz, M., Fung, F., Speck, R., Stals, L., Teranishi, K., Thibault, S., Thönnes, D., Wagner, A., Wohlmuth, B.: Resiliency in numerical algorithm design for extreme scale simulations. The International Journal of High Performance Computing Applications **36**(2), 251–285 (2022). DOI 10.1177/10943420211055188. URL `https://doi.org/10.1177/10943420211055188`

[2] Ajitsaria, A.: What is the Python global interpreter lock (GIL). `https://realpython.com/python-gil/` [Accessed 13.01.2025]

[3] Allen, S.M., Cahn, J.W.: A microscopic theory for antiphase boundary motion and its application to antiphase domain coarsening. Acta Metallurgica **27**(6), 1085–1095 (1979). DOI https://doi.org/10.1016/0001-6160(79)90196-2. URL `https://www.sciencedirect.com/science/article/pii/0001616079901962`

[4] Alvarez, D.: JUWELS Cluster and Booster: Exascale Pathfinder with Modular Supercomputing Architecture at Juelich Supercomputing Centre. Journal of large-scale research facilities JLSRF **7**, A183 (2021). DOI 10.17815/jlsrf-7-183. URL `https://jlsrf.org/index.php/lsf/article/view/183`

[5] Anzt, H., Dongarra, J., Quintana-Ortí, E.S.: Tuning stationary iterative solvers for fault resilience. In: Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA '15. Association for Computing Machinery, New York, NY, USA (2015). DOI 10.1145/2832080.2832081. URL `https://doi.org/10.1145/2832080.2832081`

[6] Ascher, U.M.: Computer methods for ordinary differential equations and differential-algebraic equations. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), Philadelphia, Pa (1998)

[7] Ascher, U.M., Ruuth, S.J., Spiteri, R.J.: Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. Applied Numerical Mathematics **25**(2), 151–167 (1997). DOI https://doi.org/10.1016/S0168-9274(97)00056-1. URL https://www.sciencedirect.com/science/article/pii/S0168927497000561. Special Issue on Time Integration

[8] Atkinson, K.E., Stewart, D., Han, W.: Numerical solution of ordinary differential equations. Pure and applied mathematics. Wiley, Hoboken, N.J (2009). DOI 10.1002/9781118164495

[9] Ayala, A., Tomov, S., Haidar, A., Dongarra, J.: heFFTe: Highly efficient FFT for exascale. In: V.V. Krzhizhanovskaya, G. Závodszky, M.H. Lees, J.J. Dongarra, P.M.A. Sloot, S. Brissos, J. Teixeira (eds.) Computational Science – ICCS 2020, pp. 262–275. Springer International Publishing, Cham (2020)

[10] Baumann, T.: Videos accompanying PhD thesis Efficient Massively Space-Time-Parallel Simulations with Adaptive Spectral Deferred Correction (2025). DOI 10.5281/ZENODO.15308621. URL https://zenodo.org/doi/10.5281/zenodo.15308621

[11] Baumann, T., Götschel, S., Lunet, T., Ruprecht, D., Speck, R.: Adaptive time step selection for spectral deferred correction. Numerical Algorithms (2024). DOI 10.1007/s11075-024-01964-z. URL https://doi.org/10.1007/s11075-024-01964-z

[12] Baumann, T., Götschel, S., Lunet, T., Ruprecht, D., Speck, R.: Resilience against soft faults through adaptivity in spectral deferred correction (2024). URL https://arxiv.org/abs/2412.00529

[13] Benson, A.R., Schmit, S., Schreiber, R.: Silent error detection in numerical time-stepping schemes. The International Journal of High Performance Computing Applications **29**(4), 403–421 (2015). DOI 10.1177/1094342014532297. URL https://doi.org/10.1177/1094342014532297

[14] Berghout, P., Baars, W.J., Krug, D.: The large-scale footprint in small-scale Rayleigh-Bénard turbulence. Journal of Fluid Mechanics **911**, A62 (2021). DOI 10.1017/jfm.2020.1097

[15] Bernaschi, M., Iannello, G., Lauria, M.: Efficient implementation of reduce-scatter in MPI. Journal of Systems Architecture **49**(3), 89–108 (2003). DOI https://doi.org/10.1016/S1383-7621(03)00059-6. URL https://www.sciencedirect.com/science/article/pii/S1383762103000596. Parallel, Distributed and Network-based Processing - selected papers from the 10th Euromicro Workshop

[16] Berrut, J.P., Trefethen, L.N.: Barycentric Lagrange interpolation. SIAM Review **46**(3), 501–517 (2004). DOI 10.1137/S0036144502417715. URL https://doi.org/10.1137/S0036144502417715

[17] Bluestein, L.: A linear filtering approach to the computation of discrete Fourier transform. IEEE Transactions on Audio and Electroacoustics **18**(4), 451–455 (1970). DOI 10.1109/TAU.1970.1162132

[18] Boyd, J.P.: Chebyshev & Fourier spectral methods. No. 49 in Lecture notes in engineering. Springer, Berlin Heidelberg New York London Paris Tokyo Hong Kong (1989)

[19] Brandt, A.: Multi-level adaptive solutions to boundary-value problems. Mathematics of computation **31**(138), 333–390 (1977)

[20] Brauer, F., Castillo-Chavez, C.: Mathematical Models in Population Biology and Epidemiology, *Texts in Applied Mathematics*, vol. 40. Springer New York, New York, NY (2012). DOI 10.1007/978-1-4614-1686-9. URL `https://link.springer.com/10.1007/978-1-4614-1686-9`

[21] Bruck, J., Ho, C.T., Kipnis, S., Upfal, E., Weathersby, D.: Efficient algorithms for all-to-all communications in multiport message-passing systems. IEEE Transactions on Parallel and Distributed Systems **8**(11), 1143–1156 (1997). DOI 10.1109/71.642949

[22] Burns, K.J., Vasil, G.M., Oishi, J.S., Lecoanet, D., Brown, B.P.: Dedalus: A flexible framework for numerical simulations with spectral methods. Physical Review Research **2**(2), 023068 (2020). DOI 10.1103/PhysRevResearch.2.023068

[23] Burrage, K.: Parallel methods for ODEs. Advances in Computational Mathematics **7**(1), 1–31 (1997). DOI 10.1023/A:1018997130884. URL `https://doi.org/10.1023/A:1018997130884`

[24] Butcher, J.C.: Numerical methods for ordinary differential equations, 2. aufl edn. Wiley, Chicester (2008)

[25] Čaklović, G., Lunet, T., Götschel, S., Ruprecht, D.: Improving efficiency of parallel across the method spectral deferred corrections. SIAM Journal on Scientific Computing **47**(1), A430–A453 (2025). DOI 10.1137/24M1649800. URL `https://doi.org/10.1137/24M1649800`

[26] Cambier, L., Pan, D., Ligowski, L.: Multinode multi-GPU: Using NVIDIA cuFFTMp FFTs at scale. `https://developer.nvidia.com/blog/multinode-multi-gpu-using-nvidia-cufftmp-ffts-at-scale/` [Accessed: 13.10.24] (2022)

[27] Casanova, H., Robert, Y., Vivien, F., Zaidouni, D.: On the impact of process replication on executions of large-scale parallel applications with coordinated checkpointing. Future Generation Computer Systems **51**, 7–19 (2015). DOI https://doi.org/10.1016/j.future.2015.04.003. URL `https://www.sciencedirect.com/science/article/pii/S0167739X15000953`. Special Section: A Note on New Trends in Data-Aware Scheduling and Resource Provisioning in Modern HPC Systems

[28] Cash, J.R., Karp, A.H.: A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides. ACM Transactions on Mathematical Software **16**(3), 201–222 (1990). DOI 10.1145/79505.79507. URL `https://dl.acm.org/doi/10.1145/79505.79507`

[29] Causley, M., Seal, D.: On the convergence of spectral deferred correction methods. Communications in Applied Mathematics and Computational Science **14**(1), 33–64 (2019). DOI 10.2140/camcos.2019.14.33. URL https://msp.org/camcos/2019/14-1/p02.xhtml

[30] Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing OpenSHMEM: SHMEM for the PGAS community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS '10. Association for Computing Machinery, New York, NY, USA (2010). DOI 10.1145/2020373.2020375. URL https://doi.org/10.1145/2020373.2020375

[31] Chegini, F., Steinke, T., Weiser, M.: Efficient adaptivity for simulating cardiac electrophysiology with spectral deferred correction methods. arXiv e-prints arXiv:2311.07206 (2023). DOI 10.48550/arXiv.2311.07206

[32] Chen, C.L., Hsiao, M.Y.: Error-correcting codes for semiconductor memory applications: A state-of-the-art review. IBM Journal of Research and Development **28**(2), 124–134 (1984). DOI 10.1147/rd.282.0124

[33] Chorin, A.J.: Numerical solution of the Navier-Stokes equations. Mathematics of Computation **22**(104), 745–762 (1968). DOI 10.1090/S0025-5718-1968-0242392-2. URL https://www.ams.org/mcom/1968-22-104/S0025-5718-1968-0242392-2/

[34] Christlieb, A., Ong, B., Qiu, J.M.: Comments on high-order integrators embedded within integral deferred correction methods. Communications in Applied Mathematics and Computational Science **4**(1), 27–56 (2009). DOI 10.2140/camcos.2009.4.27. URL http://msp.org/camcos/2009/4-1/p02.xhtml

[35] Christlieb, A., Ong, B., Qiu, J.M.: Integral deferred correction methods constructed with high order Runge-Kutta integrators. Mathematics of Computation **79**(270), 761–783 (2009). DOI 10.1090/S0025-5718-09-02276-5. URL http://www.ams.org/journal-getitem?pii=S0025-5718-09-02276-5

[36] Cieszelski, R.: A Case Study of Rayleigh-Bénard Convection with Clouds. Boundary-Layer Meteorology **88**(2), 211–237 (1998). DOI 10.1023/A:1001145803614. URL https://doi.org/10.1023/A:1001145803614

[37] Clarke, A., Davies, C., Ruprecht, D., Tobias, S., Oishi, J.S.: Performance of parallel-in-time integration for Rayleigh-Bénard convection. Computing and Visualization in Science **23**(1), 10 (2020). DOI 10.1007/s00791-020-00332-3. URL https://doi.org/10.1007/s00791-020-00332-3

[38] Colombo, A., Crivellini, A., Ghidoni, A., Massa, F., Noventa, G.: p-adaptive discontinuous Galerkin solution of transonic viscous flows with variable time step-size. Computers & Fluids **282**, 106,392 (2024). DOI https://doi.org/10.1016/j.compfluid.2024.106392. URL https://www.sciencedirect.com/science/article/pii/S004579302400224X

[39] Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Mathematics of computation **19**(90), 297–301 (1965)

[40] Courant, R., Friedrichs, K., Lewy, H.: Über die partiellen Differenzengleichungen der mathematischen Physik. Mathematische Annalen **100**(1), 32–74 (1928). DOI 10.1007/BF01448839. URL http://link.springer.com/10.1007/BF01448839

[41] Crago, S.P., Kang, D.I., Kang, M., Kost, R., Singh, K., Suh, J., Walters, J.P.: Programming models and development software for a space-based many-core processor. In: Proceedings of the 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology, SMC-IT '11, p. 95–102. IEEE Computer Society, USA (2011). DOI 10.1109/SMC-IT.2011.29. URL https://doi.org/10.1109/SMC-IT.2011.29

[42] Dalcín, L., Mortensen, M., Keyes, D.E.: Fast parallel multidimensional FFT using advanced MPI. Journal of Parallel and Distributed Computing (2019). DOI 10.1016/j.jpdc.2019.02.006

[43] Dalcín, L., Paz, R., Storti, M.: MPI for Python. Journal of Parallel and Distributed Computing **65**(9), 1108–1115 (2005). DOI https://doi.org/10.1016/j.jpdc.2005.03.010. URL https://www.sciencedirect.com/science/article/pii/S0743731505000560

[44] Dongarra, J., Geist, A.: Report on the Oak Ridge National Laboratory's Frontier system. Tech. Rep. Tech Report No. ICL-UT-22-05 (2022)

[45] Dongarra, J., Hittinger, J., Bell, J., Chacon, L., Falgout, R., Heroux, M., Hovland, P., Ng, E., Webster, C., Wild, S.: Applied mathematics research for exascale computing. Tech. rep., Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States) (2014). DOI 10.2172/1149042. URL https://www.osti.gov/biblio/1149042

[46] Dormand, J., Prince, P.: A family of embedded Runge-Kutta formulae. Journal of Computational and Applied Mathematics **6**(1), 19–26 (1980). DOI https://doi.org/10.1016/0771-050X(80)90013-3. URL https://www.sciencedirect.com/science/article/pii/0771050X80900133

[47] Dutt, A., Greengard, L., Rokhlin, V.: Spectral deferred correction methods for ordinary differential equations. BIT Numerical Mathematics **40**(2), 241–266 (2000). DOI 10.1023/A:1022338906936. URL https://doi.org/10.1023/A:1022338906936

[48] Easterbrook, S.: Computing the climate: how we know what we know about climate change. Cambridge University Press, Cambridge New York, NY Port Melbourne, VIC New Delhi Singapore (2023)

[49] Einkemmer, L.: An adaptive step size controller for iterative implicit methods. Applied Numerical Mathematics **132**, 182–204 (2018). DOI https://doi.org/10.1016/j.apnum.2018.06.002. URL https://www.sciencedirect.com/science/article/pii/S0168927418301387

[50] Emmett, M., Minion, M.: Toward an efficient parallel in time method for partial differential equations. Communications in Applied Mathematics and Computational Science **7**(1), 105–132 (2012). DOI 10.2140/camcos.2012.7.105. URL http://msp.org/camcos/2012/7-1/p04.xhtml

[51] Entschev, P.A.: Single-GPU CuPy speedups. https://medium.com/rapids-ai/single-gpu-cupy-speedups-ea99cbbb0cbb [Accessed 12.01.2025] (2019)

[52] Fiala, D., Mueller, F., Ferreira, K.B.: Flipsphere: A software-based DRAM error detection and correction library for HPC. In: 2016 IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT), pp. 19–28 (2016). DOI 10.1109/DS-RT.2016.27

[53] Freese, P., Götschel, S., Lunet, T., Ruprecht, D., Schreiber, M.: Parallel performance of shared memory parallel spectral deferred corrections (2024). URL https://arxiv.org/abs/2403.20135

[54] Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE **93**(2), 216–231 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation"

[55] Fulford, G., Forrester, P., Jones, A.: Modelling with Differential and Difference Equations. Australian Mathematical Society Lecture Series. Cambridge University Press (1997)

[56] Gander, M.J.: 50 years of time parallel time integration. In: T. Carraro, M. Geiger, S. Körkel, R. Rannacher (eds.) Multiple Shooting and Time Domain Decomposition Methods, vol. 9, pp. 69–113. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-23321-5_3. Series Title: Contributions in Mathematical and Computational Sciences

[57] Gander, M.J., Lunet, T., Ruprecht, D., Speck, R.: A unified analysis framework for iterative parallel-in-time algorithms. SIAM Journal on Scientific Computing **45**(5), A2275–A2303 (2023). DOI 10.1137/22M1487163. URL https://doi.org/10.1137/22M1487163

[58] Glosli, J.N., Richards, D.F., Caspersen, K.J., Rudd, R.E., Gunnels, J.A., Streitz, F.H.: Extending stability beyond CPU millennium: A micron-scale atomistic simulation of Kelvin-Helmholtz instability. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC '07. Association for Computing Machinery, New York, NY, USA (2007). DOI 10.1145/1362622.1362700. URL https://doi.org/10.1145/1362622.1362700

[59] Golub, G.H., Van Loan, C.F.: Matrix computations, fourth edition edn. Johns Hopkins studies in the mathematical sciences. The Johns Hopkins University Press, Baltimore (2013)

[60] Götschel, S., Minion, M., Ruprecht, D., Speck, R.: Twelve ways to fool the masses when giving parallel-in-time results. In: B. Ong, J. Schroder, J. Shipton, S. Friedhoff (eds.) Parallel-in-Time Integration Methods, pp. 81–94. Springer International Publishing, Cham (2021)

[61] Gray, A.: Getting started with CUDA graphs. https://developer.nvidia.com/blog/cuda-graphs/ (2019). Accessed: 2024-12-13

[62] Gray, P., Scott, S.: Autocatalytic reactions in the isothermal, continuous stirred tank reactor: Isolas and other forms of multistability. Chemical Engineering Science **38**(1), 29–43 (1983). DOI https://doi.org/10.1016/0009-2509(83)80132-8. URL `https://www.sciencedirect.com/science/article/pii/0009250983801328`

[63] Grout, R., Kolla, H., Minion, M., Bell, J.: Achieving algorithmic resilience for temporal integration through spectral deferred corrections. Communications in Applied Mathematics and Computational Science **12**(1), 25–50 (2017). DOI 10.2140/camcos.2017.12.25. URL `http://msp.org/camcos/2017/12-1/p02.xhtml`

[64] Guesmi, M., Grotteschi, M., Stiller, J.: Assessment of high-order IMEX methods for incompressible flow. International Journal for Numerical Methods in Fluids **95**(6), 954–978 (2023). DOI https://doi.org/10.1002/fld.5177. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.5177`

[65] Guhur, P.L., Zhang, H., Peterka, T., Constantinescu, E., Cappello, F.: Lightweight and accurate silent data corruption detection in ordinary differential equation solvers. In: P.F. Dutot, D. Trystram (eds.) Euro-Par 2016: Parallel Processing, pp. 644–656. Springer International Publishing, Cham (2016)

[66] Guibert, D., Tromeur-Dervout, D.: Parallel deferred correction method for CFD problems. In: J. Kwon, A. Ecer, N. Satofuka, J. Periaux, P. Fox (eds.) Parallel Computational Fluid Dynamics 2006, pp. 131–138. Elsevier Science B.V., Amsterdam (2007). DOI https://doi.org/10.1016/B978-044453035-6/50019-5

[67] Guo, R., Xu, Y.: A high order adaptive time-stepping strategy and local discontinuous Galerkin method for the modified phase field crystal equation. Communications in Computational Physics **24**(1), 123–151 (2018). DOI https://doi.org/10.4208/cicp.OA-2017-0074. URL `http://global-sci.org/intro/article_detail/cicp/10931.html`

[68] Guo, R., Xu, Y.: High order numerical simulations for the binary fluid–surfactant system using the discontinuous Galerkin and spectral deferred correction methods. SIAM Journal on Scientific Computing **42**(2), B353–B378 (2020). DOI 10.1137/18M1235405. URL `https://doi.org/10.1137/18M1235405`

[69] Gustafson, J.L.: Moore's Law, pp. 1177–1184. Springer US, Boston, MA (2011). DOI 10.1007/978-0-387-09766-4_81. URL `https://doi.org/10.1007/978-0-387-09766-4_81`

[70] Göddeke, D., Altenbernd, M., Ribbrock, D.: Fault-tolerant finite-element multigrid algorithms with hierarchically compressed asynchronous checkpointing. Parallel Computing **49**, 117–135 (2015). DOI https://doi.org/10.1016/j.parco.2015.07.003. URL `https://www.sciencedirect.com/science/article/pii/S0167819115001064`

[71] Hairer, E., Wanner, G.: Solving Ordinary Differential Equations II, *Springer Series in Computational Mathematics*, vol. 14. Springer Berlin Heidelberg, Berlin, Heidelberg (1996). DOI 10.1007/978-3-642-05221-7. URL `http://link.springer.com/10.1007/978-3-642-05221-7`

[72] Hairer, E., Wanner, G., Nørsett, S.P.: Solving Ordinary Differential Equations I, *Springer Series in Computational Mathematics*, vol. 8. Springer Berlin Heidelberg, Berlin, Heidelberg (1993). DOI 10.1007/978-3-540-78862-1

[73] Hansen, A.C., Strain, J.: On the order of deferred correction. Applied Numerical Mathematics **61**(8), 961–973 (2011). DOI https://doi.org/10.1016/j.apnum.2011.04.001. URL https://www.sciencedirect.com/science/article/pii/S0168927411000687

[74] Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (BLCR) for Linux clusters. Journal of Physics: Conference Series **46**(1), 494 (2006). DOI 10.1088/1742-6596/46/1/067. URL https://dx.doi.org/10.1088/1742-6596/46/1/067

[75] Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with NumPy. Nature **585**(7825), 357–362 (2020). DOI 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2

[76] Heideman, M.T., Johnson, D.H., Burrus, C.S.: Gauss and the history of the fast Fourier transform. Archive for History of Exact Sciences **34**(3), 265–277 (1985). DOI 10.1007/BF00348431. URL https://doi.org/10.1007/BF00348431

[77] Ho, K.: On the application of spectral deferred corrections to differential algebraic equations. Master's thesis, Karlsruhe Institute of Technology (2023)

[78] van der Houwen, P.J., Sommeijer, B.P.: Parallel iteration of high-order Runge-Kutta methods with stepsize control. Journal of Computational and Applied Mathematics **29**(1), 111–127 (1990). DOI https://doi.org/10.1016/0377-0427(90)90200-J. URL https://www.sciencedirect.com/science/article/pii/037704279090200J

[79] van der Houwen, P.J., Sommeijer, B.P.: Iterated Runge-Kutta methods on parallel computers. SIAM Journal on Scientific and Statistical Computing **12**(5), 1000–1028 (1991). DOI 10.1137/0912054. URL https://doi.org/10.1137/0912054

[80] van der Houwen, P.J., Sommeijer, B.P., Couzy, W.: Embedded diagonally implicit Runge-Kutta algorithms on parallel computers. Mathematics of Computation **58**(197), 135–159 (1992)

[81] Hsu, C.H., Imam, N., Langer, A., Potluri, S., Newburn, C.J.: An initial assessment of NVSHMEM for high performance computing. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1–10 (2020). DOI 10.1109/IPDPSW50202.2020.00104

[82] Huang, J., Jia, J., Minion, M.: Accelerating the convergence of spectral deferred correction methods. Journal of Computational Physics **214**(2), 633–656 (2006). DOI https://doi.org/10.1016/j.jcp.2005.10.004. URL https://www.sciencedirect.com/science/article/pii/S0021999105004663

[83] Höink, T., Schmalzl, J., Hansen, U.: Dynamics of metal-silicate separation in a terrestrial magma ocean. Geochemistry, Geophysics, Geosystems **7**(9) (2006). DOI https://doi.org/10.1029/2006GC001268. URL `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2006GC001268`

[84] Jaradat, M.K., Sik Yang, W.: An adaptive time-stepping control algorithm for molten salt reactor transient analyses. Annals of Nuclear Energy **190**, 109,880 (2023). DOI https://doi.org/10.1016/j.anucene.2023.109880. URL `https://www.sciencedirect.com/science/article/pii/S0306454923001998`

[85] John, C.M.: MPI as API: Using UCC's NCCL Backend for MPI's Allreduce. `https://x-dev.pages.jsc.fz-juelich.de/2023/07/18/mpi-ucc-nccl.html` [Accessed: 14.10.24] (2023). DOI 10.34732/XDVBLG-TVBJIG

[86] Johnson, K.G.: Reaching for the Moon: the autobiography of NASA mathematician Katherine Johnson, first edition edn. Atheneum Books for Young Readers, New York (2019)

[87] Johnson, S.S.: Sirens of Mars: searching for life on another world. Penguin Books, S.l. (2021). OCLC: 1198976118

[88] Kennedy, C.A., Carpenter, M.H.: Diagonally implicit Runge-Kutta methods for ordinary differential equations. a review (2016). URL `https://ntrs.nasa.gov/api/citations/20160005923/downloads/20160005923.pdf`

[89] Kennedy, C.A., Carpenter, M.H.: Higher-order additive Runge-Kutta schemes for ordinary differential equations. Applied Numerical Mathematics **136**, 183–205 (2019). DOI https://doi.org/10.1016/j.apnum.2018.10.007. URL `https://www.sciencedirect.com/science/article/pii/S0168927418302332`

[90] Kesselheim, S., Herten, A., Krajsek, K., Ebert, J., Jitsev, J., Cherti, M., Langguth, M., Gong, B., Stadtler, S., Mozaffari, A., Cavallaro, G., Sedona, R., Schug, A., Strube, A., Kamath, R., Schultz, M.G., Riedel, M., Lippert, T.: JUWELS Booster – a supercomputer for large-scale AI research. In: H. Jagode, H. Anzt, H. Ltaief, P. Luszczek (eds.) High Performance Computing, pp. 453–468. Springer International Publishing, Cham (2021)

[91] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture, ISCA '14, p. 361–372. IEEE Press (2014)

[92] Krause, D.: JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing Centre. Journal of large-scale research facilities JLSRF **5**, A135 (2019). DOI 10.17815/jlsrf-5-171. URL `https://jlsrf.org/index.php/lsf/article/view/171`

[93] Kremling, G., Speck, R.: Convergence of multilevel spectral deferred corrections. Communications in Applied Mathematics and Computational Science **16**(2), 227–265 (2021)

[94] Krug, D., Lohse, D., Stevens, R.J.A.M.: Coherence of temperature and velocity superstructures in turbulent Rayleigh-Bénard flow. Journal of Fluid Mechanics **887**, A2 (2020). DOI 10.1017/jfm.2019.1054

[95] Kumar, G.G., Sahoo, S.K., Meher, P.K.: 50 Years of FFT Algorithms and Applications. Circuits, Systems, and Signal Processing **38**(12), 5665–5698 (2019). DOI 10.1007/s00034-019-01136-8. URL `http://link.springer.com/10.1007/s00034-019-01136-8`

[96] Kumar, R., Mamidala, A., Panda, D.K.: Scaling alltoall collective on multi-core systems. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–8 (2008). DOI 10.1109/IPDPS.2008.4536141

[97] Lapinsky, S.E., Easty, A.C.: Electromagnetic interference in critical care. Journal of Critical Care **21**(3), 267–270 (2006). DOI https://doi.org/10.1016/j.jcrc.2006.03.010. URL `https://www.sciencedirect.com/science/article/pii/S0883944106000499`

[98] Layton, A.T., Minion, M.L.: Implications of the choice of quadrature nodes for Picard integral deferred corrections methods for ordinary differential equations. BIT Numerical Mathematics **45**(2), 341–373 (2005). DOI 10.1007/s10543-005-0016-1. URL `https://doi.org/10.1007/s10543-005-0016-1`

[99] Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's plenty of room at the top: What will drive computer performance after Moore's law? Science **368**(6495), eaam9744 (2020). DOI 10.1126/science.aam9744. URL `https://www.science.org/doi/abs/10.1126/science.aam9744`

[100] Lenz, T.: GPU-Beschleunigung von pySDC. Master's thesis, Fachhochschule Aachen (2022)

[101] Leppänen, T.: Computational studies of pattern formation in Turing systems. Ph.D. thesis, Helsinki University of Technology (2004)

[102] Li, X.S., Demmel, J.W.: SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Trans. Math. Softw. **29**(2), 110–140 (2003). DOI 10.1145/779359.779361. URL `https://doi.org/10.1145/779359.779361`

[103] Li, X.S., Lin, P., Liu, Y., Sao, P.: Newly released capabilities in the distributed-memory SuperLU sparse direct solver. ACM Trans. Math. Softw. **49**(1) (2023). DOI 10.1145/3577197. URL `https://doi.org/10.1145/3577197`

[104] Liang, X., Chen, J., Tao, D., Li, S., Wu, P., Li, H., Ouyang, K., Liu, Y., Song, F., Chen, Z.: Correcting soft errors online in fast Fourier transform. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3126908.3126915. URL `https://doi.org/10.1145/3126908.3126915`

[105] Liu, H., Zou, J.: Some new additive Runge-Kutta methods and their applications. Journal of Computational and Applied Mathematics **190**(1), 74–98 (2006). DOI https://doi.org/10.1016/j.cam.2005.02.020. URL https://www.sciencedirect.com/science/article/pii/S0377042705002220. Special Issue: International Conference on Mathematics and its Application

[106] Lohse, D., Shishkina, O.: Ultimate Rayleigh-Bénard turbulence. Rev. Mod. Phys. **96**, 035,001 (2024). DOI 10.1103/RevModPhys.96.035001. URL https://link.aps.org/doi/10.1103/RevModPhys.96.035001

[107] Lorenz, E.N.: Deterministic Nonperiodic Flow. Journal of the Atmospheric Sciences **20**(2), 130–141 (1963). DOI 10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2. URL http://journals.ametsoc.org/doi/10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2

[108] Lunet, T., Baumann, T., Speck, R.: Parallel-in-Time/qmat (2025). DOI 10.5281/ZENODO.11956478. URL https://zenodo.org/doi/10.5281/zenodo.11956478

[109] Makhoul, J.: A fast cosine transform in one and two dimensions. IEEE Transactions on Acoustics, Speech, and Signal Processing **28**(1), 27–34 (1980). DOI 10.1109/TASSP.1980.1163351

[110] Matsui, H., Heien, E., Aubert, J., Aurnou, J.M., Avery, M., Brown, B., Buffett, B.A., Busse, F., Christensen, U.R., Davies, C.J., Featherstone, N., Gastine, T., Glatzmaier, G.A., Gubbins, D., Guermond, J.L., Hayashi, Y.Y., Hollerbach, R., Hwang, L.J., Jackson, A., Jones, C.A., Jiang, W., Kellogg, L.H., Kuang, W., Landeau, M., Marti, P., Olson, P., Ribeiro, A., Sasaki, Y., Schaeffer, N., Simitev, R.D., Sheyko, A., Silva, L., Stanley, S., Takahashi, F., Takehiro, S.i., Wicht, J., Willis, A.P.: Performance benchmarks for a next generation numerical dynamo model. Geochemistry, Geophysics, Geosystems **17**(5), 1586–1607 (2016). DOI https://doi.org/10.1002/2015GC006159. URL https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2015GC006159

[111] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.1 (2023). URL https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[112] Miller, C.: Chip war: the fight for the world's most critical technology, first scribner hardcover edition edn. Scribner, an imprint of Simon & Schuster, New York (2022). OCLC: on1296942188

[113] Minion, M.L.: Semi-implicit spectral deferred correction methods for ordinary differential equations. Communications in Mathematical Sciences **1**(3), 471–500 (2003). DOI 10.4310/CMS.2003.v1.n3.a6. URL http://www.intlpress.com/site/pub/pages/journals/items/cms/content/vols/0001/0003/a006/

[114] Moon, T.K.: Error correction coding: mathematical methods and algorithms. John Wiley & Sons (2020)

[115] Munafo, R.P.: Stable localized moving patterns in the 2-d Gray-Scott model (2014). URL https://arxiv.org/abs/1501.01990

[116] Munch, P., Dravins, I., Kronbichler, M., Neytcheva, M.: Stage-parallel fully implicit Runge–Kutta implementations with optimal multilevel preconditioners at the scaling limit. SIAM Journal on Scientific Computing pp. S71–S96 (2023). DOI 10.1137/22m1503270. URL https://doi.org/10.1137%2F22m1503270

[117] Muralikrishnan, S., Speck, R.: ParaPIF: A Parareal approach for parallel-in-time integration of particle-in-Fourier schemes (2024). URL https://arxiv.org/abs/2407.00485

[118] Mycek, P., Rizzi, F., Maître, O.L., Sargsyan, K., Morris, K., Safta, C., Debusschere, B., Knio, O.: Discrete a priori bounds for the detection of corrupted PDE solutions in exascale computations. SIAM Journal on Scientific Computing **39**(1), C1–C28 (2017). DOI 10.1137/15M1051786. URL https://doi.org/10.1137/15M1051786

[119] NumPy Developers: Universal functions. https://numpy.org/doc/stable/reference/ufuncs.html#ufuncs (2024). Accessed: 2025-04-14

[120] NVIDIA Corporation: NVIDIA collective communication library (NCCL) documentation. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html (2020). Accessed: 2025-01-18

[121] NVIDIA Corporation and affiliates: CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html [Accessed 18.01.2025] (2024)

[122] Okuta, R., Unno, Y., Nishino, D., Hido, S., Loomis, C.: CuPy: A NumPy-compatible library for NVIDIA GPU calculations. In: Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS) (2017). URL http://learningsys.org/nips17/assets/papers/paper_16.pdf

[123] Olver, S., Townsend, A.: A fast and well-conditioned spectral method. SIAM Review **55**(3), 462–489 (2013). DOI 10.1137/120865458. URL https://doi.org/10.1137/120865458

[124] Ong, B.W., Schroder, J.B.: Applications of time parallelization. Computing and Visualization in Science **23**(1-4) (2020). DOI 10.1007/s00791-020-00331-4. URL https://doi.org/10.1007/s00791-020-00331-4

[125] Ong, B.W., Spiteri, R.J.: Deferred Correction Methods for Ordinary Differential Equations. Journal of Scientific Computing **83**(3), 60 (2020). DOI 10.1007/s10915-020-01235-8. URL https://doi.org/10.1007/s10915-020-01235-8

[126] Pareschi, L., Russo, G.: Implicit-explicit Runge-Kutta schemes and applications to hyperbolic systems with relaxation. Journal of Scientific Computing **25**(1-2), 129–155 (2005). DOI 10.1007/BF02728986. URL http://link.springer.com/10.1007/BF02728986

[127] Pearson, J.E.: Complex patterns in a simple system. Science **261**(5118), 189–192 (1993). DOI 10.1126/science.261.5118.189. URL https://www.science.org/doi/abs/10.1126/science.261.5118.189

[128] van der Pol, B.: LXXXVIII. *On "relaxation-oscillations"*. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science **2**(11), 978–992 (1926). DOI 10.1080/14786442608564127. URL http://www.tandfonline.com/doi/abs/10.1080/14786442608564127

[129] van der Pol, B.: The nonlinear theory of electric oscillations. Proceedings of the Institute of Radio Engineers **22**(9), 1051–1086 (1934). DOI 10.1109/JRPROC.1934.226781

[130] Quaife, B., Biros, G.: Adaptive time stepping for vesicle suspensions. Journal of Computational Physics **306**, 478–499 (2016). DOI https://doi.org/10.1016/j.jcp.2015.11.050. URL https://www.sciencedirect.com/science/article/pii/S0021999115007901

[131] Ranocha, H., Winters, A.R., Castro, H.G., Dalcin, L., Schlottke-Lakemper, M., Gassner, G.J., Parsani, M.: On Error-Based Step Size Control for Discontinuous Galerkin Methods for Compressible Fluid Dynamics. Communications on Applied Mathematics and Computation **7**(1), 3–39 (2025). DOI 10.1007/s42967-023-00264-y. URL https://link.springer.com/10.1007/s42967-023-00264-y

[132] Reid, T.R.: The chip: how two Americans invented the microchip and launched a revolution, rev. ed edn. Random House, New York (2001)

[133] Richardson, L.F.: Weather Prediction by Numerical Process, 2 edn. Cambridge Mathematical Library. Cambridge University Press (2007)

[134] Rupp, K.: Microprocessor trend data. https://github.com/karlrupp/microprocessor-trend-data [Accessed 20.10.24] (2022)

[135] Ruprecht, D., Speck, R.: Spectral deferred corrections with fast-wave slow-wave splitting. SIAM Journal on Scientific Computing **38**(4), A2535–A2557 (2016). DOI 10.1137/16M1060078. URL https://doi.org/10.1137/16M1060078

[136] Ruprecht, D., Speck, R., Emmett, M., Bolten, M., Krause, R.: Poster: Extreme-scale space-time parallelism. In: Proceedings of the 2013 Conference on High Performance Computing Networking, Storage and Analysis Companion, SC '13 Companion (2013). URL http://sc13.supercomputing.org/sites/default/files/PostersArchive/tech_posters/post148s2-file3.pdf

[137] Salsa, S.: Partial differential equations in action: from modelling to theory. Universitext. Springer, Milan (2008)

[138] Samuel, R.J., Bode, M., Scheel, J.D., Sreenivasan, K.R., Schumacher, J.: No sustained mean velocity in the boundary region of plane thermal convection. Journal of Fluid Mechanics **996**, A49 (2024). DOI 10.1017/jfm.2024.853

[139] Schroeder, B., Pinheiro, E., Weber, W.D.: DRAM errors in the wild: A large-scale field study. Commun. ACM **54**(2), 100–107 (2011). DOI 10.1145/1897816.1897844. URL https://doi.org/10.1145/1897816.1897844

[140] Söderlind, G.: Digital filters in adaptive time-stepping. ACM Trans. Math. Softw. **29**(1), 1–26 (2003). DOI 10.1145/641876.641877. URL https://doi.org/10.1145/641876.641877

[141] Speck, R.: Parallelizing spectral deferred corrections across the method. Computing and Visualization in Science **19**(3), 75–83 (2018). DOI 10.1007/s00791-018-0298-x. URL https://doi.org/10.1007/s00791-018-0298-x

[142] Speck, R.: Algorithm 997: pySDC-Prototyping spectral deferred corrections. ACM Trans. Math. Softw. **45**(3) (2019). DOI 10.1145/3310410. URL https://doi.org/10.1145/3310410

[143] Speck, R., Lunet, T., Baumann, T., Wimmer, L., Akramov, I., Rosilho de Souza, G., Fritz, J., Shipton, J.: Parallel-in-Time/pySDC (2025). DOI 10.5281/ZENODO.15196003. URL https://zenodo.org/doi/10.5281/zenodo.15196003

[144] Speck, R., Ruprecht, D.: Toward fault-tolerant parallel-in-time integration with PFASST. Parallel Computing **62**, 20–37 (2017). DOI https://doi.org/10.1016/j.parco.2016.12.001. URL https://www.sciencedirect.com/science/article/pii/S0167819116301338

[145] Speck, R., Ruprecht, D., Emmett, M., Minion, M., Bolten, M., Krause, R.: A multi-level spectral deferred correction method. BIT Numerical Mathematics **55**(3), 843–867 (2015). DOI 10.1007/s10543-014-0517-x. URL https://doi.org/10.1007/s10543-014-0517-x

[146] Speck, R., Ruprecht, D., Krause, R., Emmett, M., Minion, M.L., Winkel, M., Gibbon, P.: A massively space-time parallel N-body solver. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pp. 92:1–92:11. IEEE Computer Society Press, Los Alamitos, CA, USA (2012). DOI 10.1109/SC.2012.6. URL http://dx.doi.org/10.1109/SC.2012.6

[147] Speck, R., Ruprecht, D., Minion, M., Emmett, M., Krause, R.: Inexact spectral deferred corrections. In: T. Dickopf, M.J. Gander, L. Halpern, R. Krause, L.F. Pavarino (eds.) Domain Decomposition Methods in Science and Engineering XXII, pp. 389–396. Springer International Publishing, Cham (2016)

[148] Stevens, R.J.A.M., Blass, A., Zhu, X., Verzicco, R., Lohse, D.: Turbulent thermal superstructures in Rayleigh-Bénard convection. Phys. Rev. Fluids **3**, 041,501 (2018). DOI 10.1103/PhysRevFluids.3.041501. URL https://link.aps.org/doi/10.1103/PhysRevFluids.3.041501

[149] Strauss, W.A.: Partial differential equations: an introduction, 2nd ed edn. Wiley, New York (2009)

[150] Świrydowicz, K., Koukpaizan, N., Ribizel, T., Göbel, F., Abhyankar, S., Anzt, H., Peleš, S.: GPU-resident sparse direct linear solvers for alternating current optimal power flow analysis. International Journal of Electrical Power & Energy Systems **155**, 109,517 (2024). DOI https://doi.org/10.1016/j.ijepes.2023.109517. URL https://www.sciencedirect.com/science/article/pii/S0142061523005744

[151] Tang, T., Xie, H., Yin, X.: High-order convergence of spectral deferred correction methods on general quadrature nodes. J. Sci. Comput. **56**(1), 1–13 (2013). DOI 10.1007/s10915-012-9657-9. URL https://doi.org/10.1007/s10915-012-9657-9

[152] Thörnig, P.: JURECA: Data centric and booster modules implementing the modular supercomputing architecture at Jülich supercomputing centre. Journal of large-scale research facilities JLSRF **7**, A182–A182 (2021). DOI 10.17815/jlsrf-7-182. URL https://doi.org/10.17815/jlsrf-7-182

[153] Top500.org: Top500. https://top500.org (2025). Accessed: 2025-04-24

[154] Trefethen, L.N.: Spectral Methods in MATLAB. Society for Industrial and Applied Mathematics (2000). DOI 10.1137/1.9780898719598. URL https://epubs.siam.org/doi/abs/10.1137/1.9780898719598

[155] Trefethen, L.N.: Gray-Scott equations in 2d. https://www.chebfun.org/examples/pde/GrayScott.html [Accessed 7.12.24] (2016)

[156] Turing, A.M.: The chemical basis of morphogenesis. Bulletin of mathematical biology **52**, 153–197 (1990)

[157] Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S.J., Brett, M., Wilson, J., Millman, K.J., Mayorov, N., Nelson, A.R.J., Jones, E., Kern, R., Larson, E., Carey, C.J., Polat, İ., Feng, Y., Moore, E.W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E.A., Harris, C.R., Archibald, A.M., Ribeiro, A.H., Pedregosa, F., van Mulbregt, P., SciPy 1.0 Contributors: SciPy 1.0: Fundamental algorithms for scientific computing in Python. Nature Methods **17**, 261–272 (2020). DOI 10.1038/s41592-019-0686-2

[158] VMware: Exploring the GPU architecture. https://www.vmware.com/docs/exploring-the-gpu-architecture [Accessed 18.01.2025]

[159] Von St. Vieth, B.: JUSUF: Modular tier-2 supercomputing and cloud infrastructure at Jülich Supercomputing Centre. Journal of large-scale research facilities JLSRF **7**, A179 (2021). DOI 10.17815/jlsrf-7-179. URL http://jlsrf.org/index.php/lsf/article/view/179

[160] Wei, J., Thomas, A., Li, G., Pattabiraman, K.: Quantifying the accuracy of high-level fault injection techniques for hardware faults. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 375–382 (2014). DOI 10.1109/DSN.2014.2

[161] Weiser, M.: Faster SDC convergence on non-equidistant grids by DIRK sweeps. BIT Numerical Mathematics **55**(4), 1219–1241 (2015). DOI 10.1007/s10543-014-0540-y. URL `https://doi.org/10.1007/s10543-014-0540-y`

[162] Weiser, M., Ghosh, S.: Theoretically optimal inexact spectral deferred correction methods. Communications in Applied Mathematics and Computational Science **13**(1), 53–86 (2018). DOI 10.2140/camcos.2018.13.53. URL `https://msp.org/camcos/2018/13-1/p03.xhtml`

[163] Wu, P., Guan, Q., DeBardeleben, N., Blanchard, S., Tao, D., Liang, X., Chen, J., Chen, Z.: Towards practical algorithm based fault tolerance in dense linear algebra. In: Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC '16, p. 31–42. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2907294.2907315. URL `https://doi.org/10.1145/2907294.2907315`

[164] XIE, C., Chen, J., Firoz, J., Li, J., Song, S.L., Barker, K., Raugas, M., Li, A.: Fast and scalable sparse triangular solver for multi-GPU based HPC architectures. In: Proceedings of the 50th International Conference on Parallel Processing, ICPP '21. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3472456.3472478. URL `https://doi.org/10.1145/3472456.3472478`

[165] Zhen, M., Liu, X., Ding, X., Cai, J.: High-order space–time parallel computing of the Navier-Stokes equations. Computer Methods in Applied Mechanics and Engineering **423**, 116,880 (2024). DOI 10.1016/j.cma.2024.116880. URL `http://dx.doi.org/10.1016/j.cma.2024.116880`