WPISA-PR(18) 20745

M Wiesenberger et al.

Applications, reproducibility, numerical analysis and performance of the Feltor code and library on parallel computer architectures

Preprint of Paper to be submitted for publication in
Computer Physics Communications

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

Applications, reproducibility, numerical analysis and performance of the Feltor code and library on parallel computer architectures

Matthias Wiesenberger[a,*], Lukas Einkemmer[b,c], Markus Held[d], Albert Gutierrez-Milla[e], Xavier Sáez[e], Roman Iakymchuk[f]

[a]*Department of Physics, Technical University of Denmark (DTU), Denmark*
[b]*Department of Mathematics, University of Tübingen, Germany*
[c]*Department of Mathematics, Universität Innsbruck, Austria*
[d]*Institute for Ion Physics and Applied Physics, Universität Innsbruck, Austria*
[e]*Barcelona Supercomputing Center, Barcelona, Spain*
[f]*Department of Computational Science and Technology, Royal Institue of Technology (KTH), Sweden*

Abstract

Feltor is both a numerical library and a scientific software package built on top it. Its main target are two- and three-dimensional drift- and gyro-fluid simulations with discontinuous Galerkin methods as the main numerical discretization technique. Feltor allows developing platform independent code that runs on a variety of parallel computer architectures ranging from laptop CPUs to hybrid CPU+GPU distributed memory systems.

First, we investigate reproducibility. We observe that numerical simulations of a recently developed gyro-fluid model produces non-deterministic results in parallel computations. We show how we can restore bitwise reproducibility algorithmically and programmatically. However, we argue that in ill-conditioned physical problems numerical perturbations always grow exponentially such that convergence can fundamentally only be achieved for reduced physical quantities of interest and invariants of the system.

Furthermore, we explore important performance tuning considerations and discuss latencies and bandwidths of elementary subroutines necessary to implement the aforementioned algorithms and equations. We propose a parallel performance model that predicts the execution time of algorithms implemented in Feltor and test our model on a selection of parallel hardware architectures. We are able to predict the execution time of more complex algorithms with a relative error of less than 25% for problem sizes between 10^{-1} and 10^3MB.

Finally, we qualitatively compare a discontinuous Galerkin over the more traditional finite difference version of Arakawa's scheme for the Poisson bracket. We find equivalent results for both an external fourth order finite difference code and our own discontinuous Galerkin implementation.

1. Introduction

Simulations of phenomena in magnetized plasmas are in general highly challenging and require the use of advanced numerical algorithms and the increasing power of high-performance computers [18]. For the description of low-frequency phenomena drift-reduced Braginskii (also called *drift-fluid*) [8, 62, 42] and gyro-fluid models [7, 50, 41, 27] are efficient. Both of these approaches remove the fast time and spatial scales associated with the gyration of charged particles in the magnetic field. Compared to kinetic descriptions the reduced dimensionality in fluid models significantly lowers the computational cost. There are several codes implementing drift- and gyro-models in the literature (among others References [52, 34, 21, 49]) and Reference [15] provides an actual framework for the implementation of fluid equations.

In recent years code projects have focused on the capability to efficiently invert nonlinear elliptic equations [15, 21]. This feature is especially needed in models that relieve the so-called *Oberbeck-Boussinesq approximation* and do not distinguish between fluctuating and background quantities. For example in a gyro-fluid model we have the nonlinear elliptic equation $\nabla \cdot (N\nabla\phi) = n - N$, where n is the electron density, N is the ion gyro-center density, and ϕ the electric potential [60]. Current interest also includes the implementation of the flux-coordinate independent approach to discretize derivatives along arbitrary magnetic field lines [22, 51]. This type of scheme is particularly important if a magnetic field aligned coordinate system is unavailable. It is then challenging to resolve the inherent anisotropy of the plasma dynamics parallel and perpendicular to magnetic field lines.

Let us point out here that in the codes mentioned so far finite difference numerical methods are prevailing over more advanced schemes and the efficient use of GPUs or other accelerator cards is largely absent. We also criticize that Reference [15] is the only code that can be classified as free software in our community, which severely limits the possibility to reproduce, verify, interoperate with or reuse published results [61].

In this contribution we present our work with Feltor, a modular and free software package that we have developed particularly for the use in full-F (no Oberbeck-Boussinesq approxi-

*Corresponding author
Email address: mattwi@fysik.dtu.dk (Matthias Wiesenberger)

mation) drift- and gyro-fluid models [27, 59, 35, 26]. We use discontinuous Galerkin methods [10, 4] to spatially discretize model equations. Our efforts to enable three-dimensional simulations include the flux-coordinate independent approach within the discontinuous Galerkin framework [28], which we are the first to apply to full-F gyro-fluid models [54, 25]. Recent studies focus on numerical elliptic grid generation [57, 58]. Both are important for the efficient description of realistic magnetic field geometries.

One of the main features of the code are matrix (and in general container) free algorithms. This type of algorithm ignores the exact format or implementation of the matrix (or vector) type employed. In consequence a matrix-free implementation offers a highly flexible framework with respect to both the equations discretized and the hardware the code runs on. It allows the development of platform independent code, with the compiler choosing implementations for Nvidia GPUs using the CUDA programming language, the OpenMP parallelized version for CPUs [17], or the immediate extension to hybrid parallelization using the message passing interface (MPI).

In Section 2 of this article we give a short overview over the structure and goals of the Feltor project. Then, in the following three sections we present three different projects that involve various aspects of the library. Please note that in order to ease the reading of the article we moved the introduction to each of the discussed topics to the respective Section. In Section 3 we show how round-off errors caused by the machine precision can destroy reproducibility of a simulation. We demonstrate the implementation steps necessary to restore *bitwise* reproducibility and then debate in what ways a simulation of an ill-conditioned set of equations can be reproducible. In Section 4 we present a qualitative comparison of the Arakawa algorithm with its discontinuous Galerkin version at the example of Euler's equation in polar coordinates. Finally, in Section 5 we present results of a performance study. We briefly discuss important performance tuning methods and derive a parallel model that can predict the runtime of any algorithm in feltor on a variety of computer architectures. We present an overall discussion and conclusion of our results in Section 6.

2. Feltor overview

In this Section we give a brief overview of the structure of the Feltor project and outline its design goals and motivation. The details of how we realize these goals in code are absent in this discussion but are available in the accompanying code repository [56]. In general, we use design principles similarly found in other existing code projects (e.g. [13]) and as far as possible try to adhere to established coding practices [43, 44, 2]. The code repository includes instructions on how to compile the full user documentation, which is also available on our homepage [1]. We invite the interested reader to explore the documentation in parallel to the current section for additional information and details. We conclude this section with a short discussion of the most important implications of the project structure.

2.1. Overview

Feltor (Full-F ELectromagnetic code in TORoidal geometry) is a modular scientific software package that can be divided into six layers. Each layer defines and implements an interface that can be used by same or higher levels. This structure is depicted in Fig. 1. In the following we shortly introduce each layer and the capabilities it adds to the library.

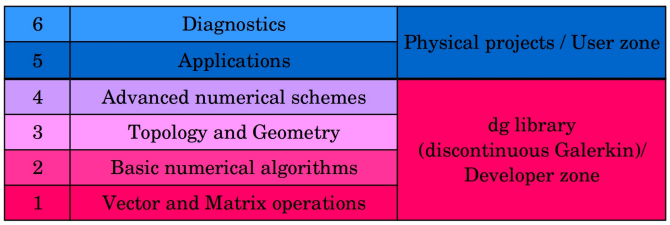


Figure 1: The structure of the project: Feltor is both a numerical library and a scientific software package built on top of that library.

User Zone A collection of actual simulation projects and diagnostic programs for two- and three-dimensional drift- and gyrofluid models

6 Diagonstics These programs are designed to analyse the output from the application programs

5 Applications Programs that execute two- and three-dimensional simulations: read in input file(s), simulate, and either write results to disc or directly visualize them on screen. Some examples led to journal publications in the past [60, 27, 38, 59].

Developper Zone The core dg library of optimized (mostly linear algebra) numerical algorithms and functions centered around discontinuous Galerkin methods on structured grids. Can be used as a standalone library.

4 Advanced algorithms Numerical schemes that are based on the existence of a geometry and/or a topology. These include e.g. the discretization of elliptic equations in arbitrary coordinates, multigrid algorithms and a semi-Lagrangian scheme to compute directional derivatives along arbitrary vector fields [28].

3 Topology and Geometry Here, we introduce data structures and functions that represent the concepts of Topology and Geometry and operations defined on them (e.g. the discontinuous Galerkin discretization of derivatives [17]). The *geometries* extension implements a large variety of grids and grid generation algorithms that can be used here [57, 58].

2 Basic algorithms Algorithms like conjugate gradient (CG) or Runge-Kutta schemes that can be implemented with basic linear algebra functions alone.

1 Vector and Matrix operations In this "hardware abstraction" level we define the interface for a set of various vector and matrix operations like additions,

multiplications, and scalar products. These functions are implemented and optimized on a variety of hardware architectures and serve as building blocks for all higher level algorithms. We study those in Section 5 of this contribution.

2.2. Design goals

The structure of Feltor is the result of an ongoing development process and subject to frequent changes. In the following we thus rather describe our goals and guidelines. These have led to the present state of the code and likely prevail in the future.

Code readability Numerical algorithms can be formulated clearly and concisely. In particular, parallelization strategies or optimization details are absent in application codes.

Ease of use We try to make our interfaces as intuitive and simple as possible. It is possible for C++ beginners to write useful, fast and reliable code with Feltor. This feature is enhanced by an exhaustive documentation.

Fast development A particular important feature from the user perspective is the possibility to quickly set up or change model equations in a minimum amount of time. We accomplish this feature by providing building blocks at Feltor's core levels, which can be freely combined or rearranged.

Speed Feltor provides specialized versions of the performance critical Level 1 functions for various target hardware architectures including for example GPUs and Intel Xeon Phis. Note that writing parallelized code is the default in Feltor. We explore and discuss performance critical issues in Section 5 of this article.

Platform independent Application code runs unchanged on a large variety of hardware ranging from a desktop environment to mid-sized compute clusters with dedicated accelerator cards. The library adapts to the resources present in the system and chooses correct implementation of functions at compile time. This is possible through a template traits dispatch system in combination with classic C-style macros at Feltor's core level. We demonstrate this feature explicitly in Section 5 of this article.

Extensibility The library is open for extensions to future hardware, new numerical algorithms and physical model equations.

Defined scope Our focus lies on efficient discontinuous Galerkin methods on structured grids and their application to drift- and gyrofluid equations in two and three dimensions. We outsource any other operation, in particular input/output, to external libraries.

2.3. Discussion

It is possible for several groups to work independently on and with Feltor on the various levels outlined in Fig. 1. Combining the defined building blocks from lower levels a user can freely construct and explore new numerical algorithms or physical equations. At the same time any improvement or upgrade of the core level routines improves the performance of all application codes using it. Of course, the set of primitive functions also restricts the number of possible numerical algorithms or equations that can be implemented. For example direct solvers are absent in Feltor.

Another advantage is the possibility to test functions and modules separately and independent of each other. We use this feature extensively throughout the development process on all levels outlined in Fig. 1. Specifically, our tests encompass unit tests for low level subroutines, convergence studies of specific numerical algorithms as well as conservation studies of invariants in our physical models.

3. Reproducibility in numerical simulations

A paradigmatic model to study drift wave turbulence and zonal flow dynamics in the edge of magnetized fusion plasmas is the Hasegawa-Wakatani (HW) model [23, 53, 24, 47]. Recently, this model has been extended to include large relative density fluctuation amplitudes and steep density gradients within a full-F gyro-fluid approach, thus facilitating studies in the non-Oberbeck-Boussinesq regime [26]. The dimensionless modified full-F HW equations consists of continuity equations for electron particle density n, ion gyro-center density N and the polarisation equation

$$\partial_t n + \{\phi, n\} = \alpha\left(\widetilde{\phi} - \widetilde{\ln(n)}\right), \tag{1a}$$

$$\partial_t N + \left\{\phi - (\nabla\phi)^2/2, N\right\} = 0, \tag{1b}$$

$$\nabla\cdot(N\nabla_\perp\phi) = n - N, \tag{1c}$$

with electric potential ϕ, adiabaticity parameter α and Poisson bracket $\{f,g\} := \partial_x f\partial_y g - \partial_y f\partial_x g$. The Reynolds decomposition $f := \langle f\rangle + \widetilde{f}$ with Reynolds averaged part $\langle f\rangle := L_y^{-1}\int_0^{L_y} dy\, f$ and fluctuating part $\widetilde{f}$ is utilized in the parallel coupling term on the right hand side of Eq. (1a).

The initial (gyro-center) density fields $n(\vec{x},0) = N(\vec{x},0) = n_G(x)\left(1+\delta n_0(\vec{x})\right)$ consist of the reference background density profile $n_G := e^{-\kappa x}$, which is perturbed by a turbulent bath $\delta n_0(\vec{x})$. Here, κ parameterizes the constant background density gradient length. For further details to the model we refer the reader to Ref. [26].

We implemented Eqs. 1 in Feltor and now want to test the reproducibility of our parallel simulations. More precisely, we want to test if with the exact same input parameters our executable reproduces the exact same output in subsequent runs. To this end we fix a typical set of physical and numerical input parameters and run our executable twice with the exact same initial condition and parallelization strategy. In Fig. 2 we compare the output of the two runs at each time step. Initially the

relative error $\epsilon_{rel} := \|n_1 - n_2\|_2 / \|n_1\|_2$ between the two solutions vanishes. Here, n_1 and n_2 is the electron density of the first and second simulation, respectively, and $\|f\|_2$ is the L_2 norm. As time advances ϵ_{rel} rapidly increases towards $O(10^{-1})$.

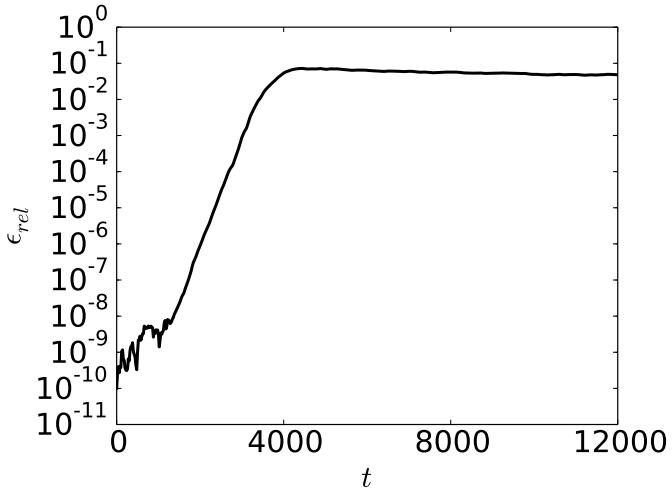


Figure 2: The relative error ϵ_{rel} as a function of time t is depicted. The relative error ϵ_{rel} between the naïve runs increases towards $O(10^{-1})$. Note, that the relative error ϵ_{rel} is biased by the constant 10^{-10}.

Although this result is very surprising at first, the possibility for two identical simulation setups to have non-identical results is readily explained. First, recall the finite nature (64-, 32-, or 16-bits) of floating-point computations that results in the non-associativity of floating-point operations [19]. For instance, let us denote $\oplus$ as the addition in `binary64` floating-point arithmetic, then $(-1 \oplus 1) \oplus 2^{-53} \neq -1 \oplus (1 \oplus 2^{-53})$ since $(-1 \oplus 1) \oplus 2^{-53} = 2^{-53}$ and $-1 \oplus (1 \oplus 2^{-53}) = 0$. Second, in a parallel environment the order of execution between threads is usually arbitrary and can vary between runs. Therefore, subsequent runs of a parallelized executable with identical input may indeed produce various binary outputs. On the other side, if the small round-off errors of machine precision lead to a large error in subsequent simulation times as seen in Fig. 2, then we are apparently faced with an ill-conditioned problem.

Both the fact that executables may produce non-deterministic results and the fact that small derivations may grow exponentially in ill-conditioned problems raise concerns about our ability to reproduce and verify our numerical simulations. In the following we view these concerns from various angles. First, we discuss reproducibility and accuracy from a purely computational and programmatic viewpoint. In Sections 3.1 and 3.2 we present how with the help of so-called *long accumulators* together with *floating-point expansions* and *error-free transformations* we can achieve *bitwise reproducibility* in our simulations. In the following Section 3.3, we then view the problem from a larger perspective and take computational, numerical and physical considerations into account. We debate the implications of finite machine precision and ill-conditioned problems on the accuracy, convergence, reproducibility, and verification of numerical simulations.

3.1. Accurate and bitwise reproducible linear algebra

In this paper, we consider the `binary64` or double-precision format of the IEEE-754-2008 standard. The standard has led to the considerable improvement in the reliability of numerical computations by rigorously specifying the properties of floating-point arithmetic. Thanks to the adaptivity of this standard by most processors, the numerical portability of applications was eased. The standard requires the basic arithmetic operations $(+, -, \times, /, \sqrt{})$ to be correctly rounded (rounding-to-nearest) [19, 29, 45]. This means that the basic operations return the closest floating-point number to the exact result, breaking ties by rounding to the floating-point number with the even significant. In this study, we assume the rounding-to-nearest rounding-mode.

Due to the finite nature of floating-point computations as well as the non-determinism of parallel executions, we develop an approach to ensure bit-wise reproducibility via ensuring correctly rounded results, whenever possible. The main idea is to keep track of both the result and the errors during the course of computations. To increase the accuracy of floating-point operations, i.e. assure their correct rounding, we rely upon the following two strategies: the first computes the result and recovers the rounding error using so-called *error-free transformations* (EFT) and stores both result and error in a *floating-point expansion* (FPE). A FPE is an unevaluated sum of p floating-point numbers whose components are ordered in magnitude with minimal overlap to cover a wide range of exponents. Typically, a FPE relies upon the use of the `TwoSum` EFT [36] for the addition and the use of the `TwoProd` EFT for the multiplication [48]. The main advantage of FPEs is that they could be fetched to the registers and reside there during the computation. However, they may not be able to guard every bit of information, which is necessary for correct rounding, for large sums or for floating-point numbers with significantly variations in magnitude.

The second strategy projects the finite range of exponents of floating-point numbers into a long vector the so-called *long (fixed-point) accumulator*. A fixed-point representation stores numbers using an integral part and a fractional part of fixed size, or equivalently a scaled integer. For instance, Kulisch [39] proposed to use a 4288-bit long accumulator for the exact dot product of two vectors composed of `binary64` numbers; however, such a large long accumulator is designed to cover all the severe cases without overflows in its highest digit. By preserving every bit of information, the long accumulator guarantees to compute the exact result of a large amount of floating-point numbers of arbitrary magnitude. However, when comparing to FPE, the long accumulator has a large memory footprint and requires roughly two times more operations to be performed.

With the aim to derive fast, accurate, and reproducible Basic Linear Algebra Subprograms (BLAS), we construct a multi-level approach for these operations that is tailored for various modern architectures with their complex multi-level memory structures. On one side, we want this approach to be fast to ensure compatible performance compared to the non-deterministic parallel versions. On the other side, we want to

preserve every bit of information before the final rounding to the desired format to assure correct-rounding and, therefore, reproducibility. To accomplish our goal, we merge together FPE and long accumulators, tune them, and efficiently implement them on various architectures, including conventional CPUS, Nvidia and AMD GPUs, and Intel Xeon Phi co-processors (for details we refer to Ref. [11]). We begin with the parallel reduction, which is in the core of many BLAS routines. We build its scalable, accurate, and reproducible version using FPEs with the `TwoSum` EFT and long accumulators. In practice, the latter is so rarely invoked that only little overhead (less than 8 %) results on summing large vectors.

The dot product of two vectors is another crucial fundamental BLAS operation. The EXDOT (exact stands for accurate and reproducible) algorithm is based on the previous EXSUM algorithm and the `TwoProd` EFT: we accumulate both the result and the error to FPEs and reduce these FPEs and long accumulators on various levels as in EXSUM. These and other routines – such as matrix-vector product (EXGEMV), triangular solve (EXTRSV), and matrix-matrix multiplication (EXGEMM) – are distributed as the Exact BLAS (ExBLAS) library [30, 31]. Thanks to the modular and hierarchical structure of linear algebra algorithms, higher level operations – such as matrix factorizations – can be entirely built on top of the fundamental kernels as those in the BLAS library. In ExBLAS, we follow this principal to construct reproducible LU factorizations with partial pivoting.

3.2. *Reproducibility in* FELTOR

As outlined in Section 2 FELTOR builds its algorithms on basic primitive functions, which partly overlap with the BLAS library. Please find the exact list of functions in the documentation. Our basic assumption is that, if these elementary functions are reproducible, then all algorithms and simulations implemented with them are reproducible. This assumption follows our theoretical and practical studies [32] of the unblocked LU factorization with partial pivoting, which underneath is entirely build upon the BLAS routines. The first step to realize this goal incorporates the exactly rounded and reproducible parallel reduction from the ExBLAS library into FELTOR. In this way we can provide the exact and reproducible dot product $\sum_i x_i y_i$. Note that we also provide a function computing the weighted sum $\sum_i x_i w_i y_i$, where w represents for example the volume form of our coordinate system. This is important in numerical computations of the scalar product $\int f_1 f_2 \sqrt{g} \mathrm{d}V$ with functions f_1, f_2 and volume element $\sqrt{g}$.

In the second step we make the trivially parallel vector operations like $y \leftarrow \alpha x + \beta y$ reproducible. Unfortunately, the use of FPEs or long accumulators for these very small summations introduce too much overhead to be practical. On the other hand we do not parallelize the summation itself. We therefore use that if we can guarantee the type and order of execution to be the same on all compilers and platforms that follow the IEEE-754-2008 standard, the results are identical even though they are not exactly rounded. For performance reasons, the C++ language standard allows compilers to change the execution order of a given line of code. It even allows merging multiplications and summations with fused multiply add (FMA) instructions. These compute `a*x+b` in a single instruction with only a single rounding operation. Consider now the 'naive' implementation `y=a*x+b*y`. A compiler might translate this to two multiplications `t1=a*x` and `t2=b*y` and a subsequent summation `y=t1+t2`, or it might generate a single multiplication `t=b*y` with a subsequent FMA[1] `y=fma(a,x,t)`, which gives a slightly different result.

Our approach to solve this issue is to explicitly instruct the compiler to use FMAs together with relevant compiler flags to prevent the use of value changing optimization techniques (e.g. `-fp-model precise` for the intel `icc` compiler). The former is possible through the `std::fma` instruction added to the C++-11 language standard[2]. With this combination we avoid non-determinism in the order of operations, reduce the number of rounding errors from three to two, and, therefore, achieve binary reproducibility for this operations and even for matrix-vector multiplications $y \leftarrow \alpha M x + \beta y$. There, we take special care to secure parallel summation in our MPI implementation. The computation of boundary points can begin only after all values from other processes were communicated.

The third step towards reproducibility in FELTOR is to make the initialization of vectors reproducible. Here, the main problem lies in the use of transcendental functions like e^x, $\sin(x)$ or $\cos(x)$. Consider for example Eq. (7) in Section 4. The algorithms for computing these functions differ by compiler and the results subsequently differ if not correctly rounded[3]. A practical and portable solution to this problem is an open issue in FELTOR.

All in all, FELTOR yields reproducible results up to the compiler and the CPUs capability to compute FMAs. This means that we can reproduce simulation results bit for bit, independently of parallelization, as long as we use the same compiler and `fma` flag as the original.

3.3. *Bitwise reproducibility, accuracy, convergence and verification*

We improved FELTOR with the reproducible BLAS Level-1 subroutines and can now re-simulate Eqs. 1 and indeed obtain bitwise identical results after each run. We show our solution in Fig. 3 where we compare the radial zonal flow structure to the previous implementation. Here, the radial zonal flow structure of the naïve implementation deviates while the zonal flow structures are identical in the new implementation.

Let us now discuss the implications of what we have achieved and know up to this point.

Bitwise reproducibility We have the possibility to reproduce parallel simulation results bit-to-bit. This is particularly

[1] It may even compute `a*x` first and then use the FMA.

[2] Unfortunately, at the time of this writing the intel and microsoft compilers do not properly vectorize code involving `std::fma`. For the time being our implementation relies on `icc` and `msvc` to always translate `a*x+b` into an FMA instruction.

[3] In fact, the difference comes from the transcendental functions implementations in `libm`. Note that GNU `libm` ensures correct-rounding of these functions thanks to the GNU Multi Precision Arithmetic library. With `icc` we had to use a special flag `-fimf-arch-consistency=true` to get reproducible results across platforms.

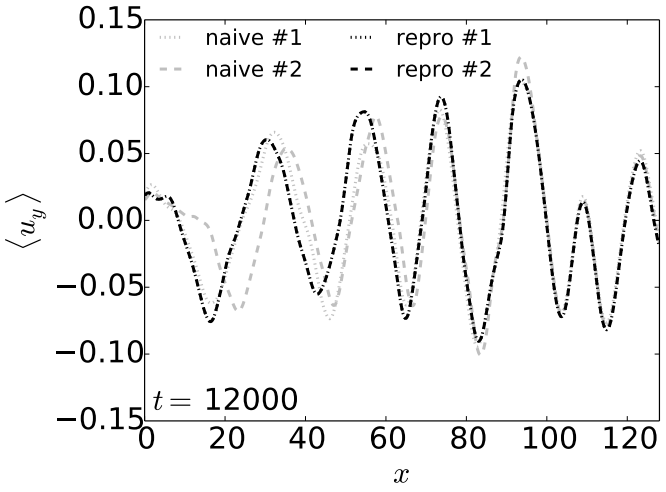


Figure 3: The radial zonal flow signature is shown. The deviation of zonal flow structure of two naïve runs with identical initial conditions is clearly visible. As opposed to this the zonal flow structure of the bitwise reproducible runs are identical.

advantageous from a programmatic point of view since alterations in the implementation or future adaptions to other parallel hardware can be rigorously checked and tested. Moreover, independent outside groups and ourselves gain the possibility to re-simulate and confirm the results. This is especially important since we usually refrain from publishing output files due to their impractically large size. Now, we have the possibility to publish the code together with input files and can expect to exactly reproduce presented results.

Accuracy It is important to mention that we not only achieved bitwise reproducibility but also increased the accuracy of our implementation, mainly in the scalar product. The problem with the previous naïve summation was the unfavourable cancellation of digits when adding small values to a large sum. Even in a tree summation algorithm the error grows with the size of the array with $\sqrt{a}$ with a being the array size. This effectively reduces the machine precision, which is a particular concern for large scale single precision computations. It is expected that the next generation of supercomputers, i.e. Exascale systems, will be composed of heterogeneous resources like CPUs and accelerators. Obtaining peak performance on these systems will, in all likelihood, require the use of single or mixed-precision simulations [5, 6, 9, 16].

Condition If we evolve the physical model Eqs. 1 over long periods of time, even small (physical) perturbations in the initial state can be amplified by many orders of magnitude. This is a fundamental property of the physical system under consideration. Consequently, this behaviour is also reflected in the numerically discretized system of equations. Recall that (numerical) perturbations are always present in this system. For example, even if the initial state is given by an analytical function its numerical representation is already inexact due to either the discretization error or the finite precision of floating point arithmetics. These (numerical) perturbations then grow over time just as their physical counterparts do.

In conclusion, we have to accept that even with the increased accuracy and reproducibility of our implementation the error[4] in our numerical solution is large after a sufficiently long simulation time. This is because any error stemming either from the numerical discretization or the finite machine precision will be amplified by the system. In particular this means that we cannot obtain convergence of our simulation. Even with infinite machine precision we would need a prohibitively fine grid to find the exact solution. On the other side the error in a single time-step or small enough time span may still be acceptable and converge with the expected order. In this context we can also expect that as long as we can maintain the machine precision in our implementation (especially for the dot product, as discussed above) using single precision gives the same physical results as does double precision. In memory bandwidth bound problems this can potentially lead to a factor two gain in performance.

Furthermore, we have to reject the notion that our bitwise reproducible solution is any more physically or numerically reasonable than the previous solution, even if the accuracy of elemental operations was increased. We only select one specific solution out of a larger class of solutions equivalent within the limits of the accuracy of the numerical discretization and the machine precision. In fact, we can also physically expect a larger class of end states that are equivalent within small (thermal) fluctuations that are present in the turbulent system described by Eqs. 1.

We therefore conclude that we need alternative methods to verify our numerical representation of Eqs. 1 than pointwise convergence. One suggestion might be to study the convergence of the actual physical quantities that we are interested in. This can for example be the zonal flow structure in Fig. 3 or turbulent spectra. Also, in the absence of convergence studies, invariants of the physical model gain importance as a consistency check of numerical methods and implementations. This on the one hand means that we should favour physical models that do provide invariants and numerical methods that conserve these invariants. Conservative numerical methods can (often) give us a good physical picture even though the L^∞ error is large. On the other hand, it is difficult to guarantee correct (physical) behaviour from symmetries of a system alone. For example energy conservation is in general not enough to guarantee physical solutions. As exemplified by the integration of the solar system in [20, Chap. 1], a numerical integrator may be locally converged and conserve energy and still produce a strikingly wrong (both qualitative and quantitative) result after a long time. Thus, we still have to conduct convergence studies applied to the physical quantities of interest.

[4] in the L^∞ or any other suitable norm

4. The Arakawa scheme

In the light of the findings in Section 3 we now discuss a numerical scheme that was specifically designed to conserve the invariants of the underlying model equations. In Section 4.1 we introduce the method while in Section 4.2 we qualitatively compare a finite difference and a discontinuous Galerkin version of the method.

4.1. Numerical method

In 1966 Arakawa [3] introduced a finite difference approximation for two-dimensional incompressible flow. That is, he devised a numerical scheme for the two-dimensional Navier–Stokes equation in vorticity formulation

$$\begin{aligned} \partial_t \omega + \{\phi, \omega\} &= 0, \\ \Delta \phi &= \omega, \end{aligned} \tag{2}$$

where the Poisson bracket is given by $\{\phi, \omega\} := \partial_x \phi \partial_y \omega - \partial_y \phi \partial_x \omega$. The function ϕ is called the potential and is obtained by solving a Poisson equation with ω as the source term. The main motivation for Arakawa's finite difference scheme is that it enables long time integration without the rapid growth of the total kinetic energy of the system (which can be observed, for example, for the standard centered difference discretization of the Poisson bracket). In fact, Arakawa's method, conserves mass as well as the two quadratic invariants kinetic energy and enstrophy up to machine precision.

The classic second order Arakawa method is constructed as follows

$$\{f, g\} \approx J_1 = \tfrac{1}{3}(J^{++} + J^{+x} + J^{x+}) \tag{3a}$$
$$J^{++} = D_x(f)D_y(g) - D_y(f)D_x(g) \tag{3b}$$
$$J^{+x} = D_x(fD_y(g)) - D_y(fD_x(g)) \tag{3c}$$
$$J^{x+} = D_y(D_x(f)g) - D_x(D_y(f)g), \tag{3d}$$

where D_x and D_y denote the standard centered finite difference operator in the x and y-directions, respectively. This numerical scheme has been found originally by Taylor series expansion. However, it can also be interpreted as an equal superposition of discretizations obtained by different formulations of the Poisson bracket (which, in the continuous case, are identical under the product rule; see [17]).

Despite the advantages of Arakawa's method, it was not appreciated in many physical applications because it is difficult to generalize it to the three dimensional case (recently an approach [37, Chap. B.2] based on Nambu brackets [46] has been put forward). However, in recent years it has received increasing attention from the plasma physics community. This is due to the fact that in magnetized plasmas the dynamics parallel and perpendicular to the magnetic field can often be separated. Thus, yielding an essentially two-dimensional problem perpendicular to the magnetic field.

Recently, the interpretation of Arakawa's method outlined above has allowed its extension to a discontinuous Galerkin space discretization [17]. This approach takes the same form as (3), except that D_x and D_y are now computed by a discontinuous Galerkin approximation. For simplicity we will illustrate this in a single dimension (the extension to two dimensions is straightforward; for more details we refer the reader to [17]). To start we divide our computational domain into a number of cells, where the nth cell is defined as $C_n = [x_{n-1/2}, x_{n+1/2}]$. The numerical approximation of a function f is then given by

$$f(x) \approx f_h(x) = \sum_{ni} f^{ni} p_{ni}(x),$$

where h is the grid spacing, f^{ni} are the degrees of freedom that need to be stored in the algorithm, and p_{ni} is the Legendre polynomial of degree i scaled and translated to the nth cell. The derivative is then given by

$$D_x(f) = \sum_{ni} f^{ni}_{h;x} p_{ni}(x),$$

where

$$f^{ni}_{h;x} = \hat{f} p_{ni}|^{x_{n+1/2}}_{x_{n-1/2}} - \int_{C_n} f_h(x) \partial_x p_{ni} \, \mathrm{d}x.$$

The quantity $\hat{f}$ is the numerical flux and needs to be specified. It turns out that if we choose the centered flux, i.e. we set

$$\hat{f}(x) = \tfrac{1}{2} \lim_{\epsilon \to 0, \epsilon > 0} f_h(x + \epsilon) + \tfrac{1}{2} \lim_{\epsilon \to 0, \epsilon > 0} f_h(x - \epsilon)$$

the favorable properties of Arakawa's method are preserved by this discontinuous Galerkin scheme. In particular, mass, kinetic energy, and enstrophy are conserved up to machine precision. Moreover, this approach has the advantage that numerical methods of arbitrary order can be constructed easily. Also, discontinuous Galerkin methods, in general, are advantageous for parallelization as the coupling between different cells is mediated by a single value only (the numerical flux at the cell interface). We will explore this feature in Section 5 of this paper.

Before proceeding, let us remark that even though the second order Arakawa method has been widely used in the plasma physics literature, the corresponding fourth order method is significantly less well known. This numerical method will be part of the comparison in the next section and is given by [3]

$$\{f, g\} \approx 2J_1 - J_2$$

with

$$J_2 = \frac{1}{3}(J_2^{xx} + J_2^{+x} + J_2^{x+}) \tag{4a}$$
$$\begin{aligned} d(J_2^{xx})_{ij} = {} & (f_{i+1,j+1} - f_{i-1,j-1})(g_{i-1,j+1} - g_{i+1,j-1}) \\ & - (f_{i-1,j+1} - f_{i+1,j-1})(g_{i+1,j+1} - g_{i-1,j-1}) \end{aligned} \tag{4b}$$
$$\begin{aligned} d(J_2^{+x})_{ij} = {} & f_{i+2,j}(g_{i+1,j+1} - g_{i+1;j-1}) - f_{i-2,j}(g_{i-1,j+1} - g_{i-1;j-1}) \\ & - f_{i,j+2}(g_{i+1,j+1} - g_{i-1,j+1}) + f_{i,j-2}(g_{i+1,j-1} - g_{i-1,j-1}) \end{aligned} \tag{4c}$$
$$\begin{aligned} d(J_2^{x+})_{ij} = {} & f_{i+1,j+1}(g_{i,j+2} - g_{i+2,j}) + f_{i-1,j-1}(g_{i-2,j} - g_{i,j-2}) \\ & - f_{i-1,j+1}(g_{i,j+2} - g_{i-2,j}) + f_{i+1,j-1}(g_{i+2,j} - g_{i,j-2}), \end{aligned} \tag{4d}$$

where $d = 8h_x h_y$. The grid spacing in the x-direction is denoted by h_x and the grid spacing in the y-direction is denoted by h_y.

Similar to the second order Arakawa scheme this method conserves mass, kinetic energy, and enstrophy and has been found by Taylor series expansion.

Strictly speaking these invariants are conserved up to machine precision only for periodic boundary conditions. This is equally true for both the second and fourth order Arakawa methods and the discontinuous Galerkin schemes. For Dirichlet and Neumann boundary conditions a (usually small) error is incurred at the boundary. This is discussed in some detail in [12].

4.2. Numerical comparison

The order of the discontinuous Galerkin scheme is determined by the polynomial degree used to approximate the solution in each cell. Choosing a piecewise constant approximant actually yields a second order method that, for discretizing the Poisson bracket, is identical to the arakawa2 scheme. Choosing a piecewise cubic approximant results in a fourth order discontinuous Galerkin scheme (henceforth denoted by dG4) that has been found to yield good results in plasma simulations ([60]). The primary goal of this section is thus to compare the dG4 method to the arakawa2 and arakawa4 scheme.

For the former we use the Feltor code [56], while for the latter the implementation described in [12] is used. There are important differences of these two implementations, in addition to the fact that different numerical methods are used to discretize the Poisson bracket. Most notably that the Feltor code is more general in that it can handle arbitrary structured grids (see, for example, [57]). On the other hand, the code implementing the Arakawa method is tailored to polar coordinates. In particular, the Poisson equation is solved using a spectral method. In addition, a range of optimizations have been conducted to increase the performance for that specific case. In this section, we will use the implementation [12] as a reference to determine the quality of the dG method. Since it uses a different numerical approach (for both the Poisson bracket as well as for solving the Poisson equation), any bias introduced would be in favor of the Arakawa scheme and thus would make the results obtained in this section even more favorable towards the discontinuous Galerkin approach.

Our goal in this section is to perform a comparison between the discontinuous Galerkin approach, as described in the previous section, and the Arakawa finite difference approach. Since the code implementing Arakawa's method can only handle polar coordinates (r, θ) we will restrict ourselves to this case. In the following we will consider a simple yet interesting model from plasma physics (see, for example, [40]). More specifically, the guiding center model in polar coordinates is given by

$$\partial_t \rho - \frac{1}{r}\{\phi, \rho\} = 0, \tag{5}$$

where r is the volume element and the potential is determined by the Poisson equation in polar coordinates

$$-\partial_r^2\phi - \frac{1}{r}\partial_r\phi - \frac{1}{r^2}\partial_\theta^2\phi = \rho. \tag{6}$$

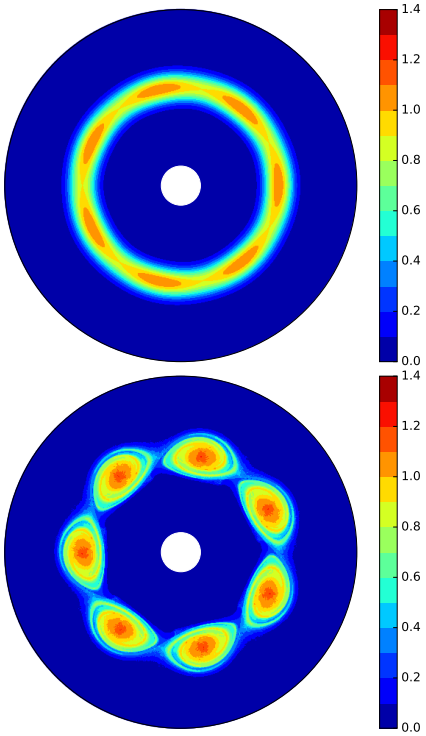


Figure 4: A numerical solution of the guiding center model at $t = 0$ (top) and $t = 30$ (bottom) is shown. The fourth order Arakawa scheme, 1024 grid points per direction, and a time step size of $\tau = 0.01$ has been used.

As a side remark let us comment on the definition of the Poisson bracket in an arbitrary coordinate system (ζ, η). The interested reader will note that the volume form in two dimensions can be interpreted as a symplectic two-form and thus is, in fact, the defining factor for the Poisson bracket $\{f, g\} := (\partial_\zeta f \partial_\eta g - f_\eta g_\zeta)/\sqrt{g}$. For the purposes of this work the definition below Eq. 2 without volume element is sufficient.

The sought-after quantity is $\rho(t, x, y)$ which is defined on the domain $\Omega = [r_{\min}, r_{\max}] \times [0, 2\pi]$. We consider the initial value

$$\rho(0, x, y) = (1 + \epsilon \cos(\ell\theta))\mathrm{e}^{-2(r-r_c)^2} \tag{7}$$

with $\epsilon = 0.1$, $\ell = 7$, $r_c = 6$, $r_{\min} = 1$, and $r_{\max} = 10$. We fix N_r and N_θ as the number of grid points in the r and θ direction for the Arakawa scheme and the product of cell number and polynomial coefficients in each cell for the dG scheme. Equation (5) is then integrated in time. A plot of the numerical solution at $t = 30$ is shown in Figure 4. We can clearly see the development of small scale structures. Thus, we are mostly interested in the qualitative properties of the solution; as opposed to say the L^∞ error in ρ. This is in fact the situation for which Arakawa's method has been originally designed and in which the improved conservation properties of the method are most important.

To perform the comparison we show numerical results for the dG4, arakawa2, and arakawa4 scheme in Figure 5 (where, in order to facility a direct comparison, we have plotted only a single vortex). Among these methods the arakawa2 scheme clearly performs worst. For both the arakawa4 scheme and, in particular, the dG4 scheme (that is implemented in FELTOR)

we see significant improvements.

5. Performance and runtime prediction

In this Section we present a performance study of the FELTOR library. This study includes a second dataset [55] to this article, which provides the complete raw data in csv format as well as the `ipython` notebooks used for the data analysis and plot generation. The interested reader is invited to inspect these notebooks in parallel to reading this section for additional information and details.

We begin this section with a discussion of important performance optimization techniques for memory bandwidth bound algorithms 5.1. We then shortly describe the hardware, the configuration and the program that we used to generate the performance data 5.2. Having measured and discussed the performance of FELTOR's building blocks in Section 5.3 we suggest a performance model that predicts the runtime of any constructed algorithm in Section 5.4. We discuss strong and weak scaling in Section 5.5 and conclude with a critical discussion in Section 5.6.

5.1. Optimization techniques for low-level FELTOR *routines*

As mentioned in Section 2, the Level 1 algorithms implemented in FELTOR include basic algebra routines that build the dg library code. Besides trivially parallel vector operations like addition or pointwise multiplication, we implemented the scalar product with long accumulators (Section 3) and a sparse matrix-vector multiplication. Optimizing these operations is a key task in order to increase the overall performance of any higher level algorithm or application using FELTOR.

Note, that we devised our own sparse block matrix format, which specifically saves storage on redundant blocks and thus potentially fits into small and fast memory caches of the target architecture. It is used for the computation of the simple discontinuous Galerkin derivative in x and y on product spaces (see Section 4 and Reference [17] for more details). Many algorithms, including the Arakawa scheme, build on those derivatives. An optimization of the corresponding matrix-vector product will thus greatly contribute to reducing their execution times.

In general, vector additions, sparse-matrix-vector multiplications and scalar products require a similar amount of memory and arithmetic operations. This means that on all modern hardware architectures these routines are memory bound. However, this conclusion assumes an efficient implementation. In particular, it assumes that our code is able to exploit the parallelism present on these architectures in order to saturate the available bandwidth. In addition, to achieve optimal performance, memory has to be read in a sequential (coalesced) manner. This is especially true for the Intel Xeon Phi "Knights Landing" accelerator card (KNL) and GPUs.

The easiest option to optimize a code in a new architecture such as KNL is to recompile it with the proper flags (discussed for KNL further below) and thus get an instantaneous benefit. However, achieving a full and efficient use of a new architecture requires an analysis using available profiling tools and an optimization effort, which is reflected normally in code modifications.

The strategy to optimize a code for a given architecture involves different levels, beginning from the core level to the outer levels of the hardware, since all the optimizations introduced in any level automatically benefits its upper levels.

Most modern processors have so-called *vector units* that allow it to execute a single instruction on multiple data (SIMD) per cycle. For example, each KNL core has two 512-bit vector units that enable it to compute 16 double precision operations concurrently. The usage of these SIMD (or vector) instructions in a loop is called *vectorization*.

Most compilers may vectorize loop structures automatically to take advantage of vector units if they are called with the proper options. For the KNL, the intel compiler provides `-xMIC-AVX512` to enable AVX-512 vector instruction set [33], `-fma` to generate fused multiply-add (FMA) instructions and `-align` to use aligned load or store vector instructions.

However, the vectorization report generated by the compiler typically shows that not all loops can be vectorized. The compiler only vectorizes when it considers this process a) safe and b) improves the performance. This means that in order to achieve a good performance sometimes we have to help the compiler to vectorize loops initially discarded by it. For example, when the compiler believes that two pointers in a loop may reference a common memory region implying likely data dependencies among iterations the compiler refrains from vectorization. This situation can be solved using the keyword `restrict` for a pointer argument in a C/ C++ function, which indicates that the pointer argument provides exclusive access to the memory referenced in the function and no other pointer can access it.

Another example is when the compiler does not vectorize a loop because an efficiency heuristics predicts that this vectorization will lead to a worsening of the performance, such as the presence of many unaligned data accesses. This time it can be solved introducing the OpenMP-4 extension `#pragma omp simd`, which explicitly tells the compiler that it is safe to use SIMD instructions in the following loop.

In general we observe that vectorization significantly improves the performance of the scalar product with long accumulators, our sparse matrix-vector multiplication and to a lesser extent also the vector additions.

Continuing with the higher hardware level, a KNL node contains 68 cores, so a good thread scalability is mandatory to take advantage of them. In the case of the sparse-matrix vector multiplication, the previous code contained three consecutive OpenMP parallel regions that were merged into one to give all threads more work reducing idle time and overhead costs, such as thread management and synchronization. Besides, KNL offers hyperthreading, which means that each core supports up to 4 threads, leading to the possibility of using up to 272 threads per KNL node. As hyperthreading may improve performance when memory access latency limits the execution, some performed experiments suggested to run at least 2 threads in order to increase the full core usage and so improve performance.

Finally, we observe that making the number of polynomial

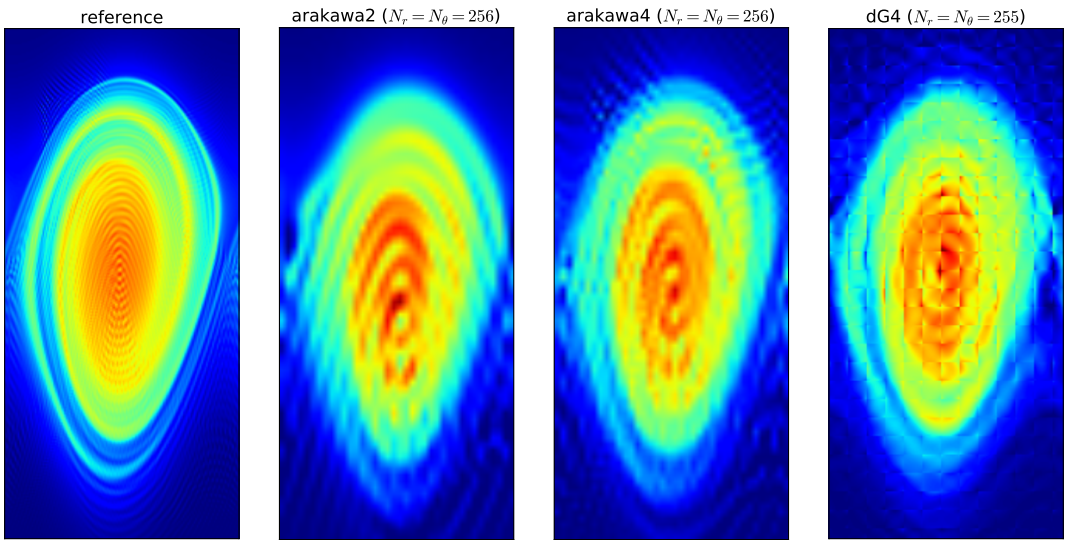


Figure 5: The numerical solution, restricted to $[4.6, 7.5] \times [0, 0.95]$ for the guiding center model at $t = 30$ is shown. The same colors as in Figure 4 are used. The reference solution is computed using the fourth order Arakawa scheme and 1024 grid points per direction.

coefficients a compile time constant (a template parameter) resulted in another significant improvement of runtime in the matrix-vector multiplication. The coefficient fixes the size of the blocks in the sparse matrix format and thus the size of the tight inner loops of the routine.

Note that all the optimizations that have been performed for the KNL have a positive effect on the regular CPU performance as well. On GPUs we observe similar performance improvements when we add the `restrict` keyword to pointer arguments in the corresponding kernels and use template arguments as well. We can avoid warp divergence since if-clauses are absent in our implementations.

5.2. Configuration

We use the program `feltor/inc/dg/cluster_mpib.cu` that is contained in [56], together with suitable submit scripts in [55], for generating the performance data. Essentially, we gather the average run times of a variety of primitive functions, the Arakawa algorithm and a conjugate gradient iteration. Let us note here that the results from different architectures are bitwise identical as long as we only compare results from the same compiler (see Section 3). We vary problem sizes and number of compute nodes on a selection of representative hardware architectures, which includes a current consumer grade desktop CPU and GPU, as well as dedicated high performance compute hardware from Intel and Nvidia. Please find a short description of the configuration in Table 1 and more details in the dataset [55]. We refer to the documentation of the `dg::Timer` class in [56] for details of how we measure the time on the various architectures involved.

5.3. Performance measurements

From the measured runtime t and the array size S we compute the memory bandwidth b of an algorithm or function

$$b = \frac{mS}{t} \tag{8}$$

where m is the number of memory loads and stores. We follow the STREAM conventions in counting memory operations, which means that we separately count each read and each write of a memory location. For example the vector addition `axpby`, which computes the operation $y \leftarrow \alpha x + \beta y$, counts as $m = 3$ times the vector size since we have to read both x and y and then write into y. The `dot` product $x \cdot y$ counts as $m = 2$ times the vector size.

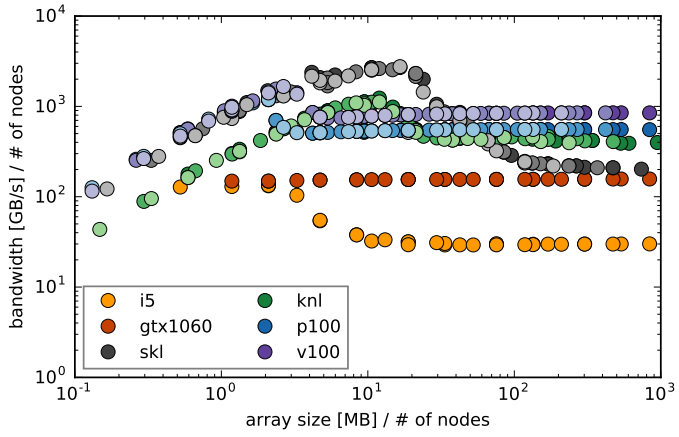
In Fig. 6 we plot the average bandwidth b for various hardware architectures and problem sizes S. We normalize the plot to the number of nodes n, b/n and S/n, such that each point represents the performance of a single node.

First, we note that in both, Fig. 6a and Fig. 6b the lightly colored points from multi-node runs lie almost exactly on top of their single-node counterparts. This is especially true for the P100 and V100 GPUs and the Skylake nodes but is not so well fulfilled for the Xeon Phi. The feature indicates a high *weak scaling efficiency* of both `axpby` and `dot`, which means that the achievable bandwidth of a given node is given solely by the problem size on the node itself.

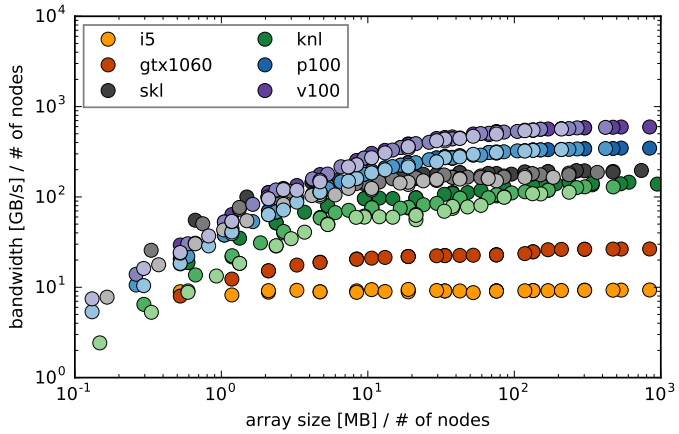
Next, we note that in Fig. 6a the bandwidth for small to medium sized problems (1MB $< S/n <$ 10MB) is significantly higher than for large problems ($S/n >$ 100MB) for all architectures (note that the lowest sized point of the 'gtx1060' is hidden beneath a 'skl' point). This is especially pronounced for the Skylake architecture. We explain this by the cache level hierarchy. The problem fits entirely into the cache such that its higher speed becomes visible. In fact, the peaks roughly coincide with the relevant cache sizes (see [55] for exact cache

	device description	single-node configuration
i5	Intel Core i5-6600 @ 3.30GHz (2x 8GB DDR4, 4 cores)	1 MPI task x 4 OpenMP threads (1 per core)
skl	2x Intel Xeon 8160 (Skylake) at 2.10 GHz (12x 16GB DDR4, 2x 24 cores)	2 MPI tasks (1 per socket) x 24 OpenMP threads (1 per core)
knl	Intel Xeon Phi 7250 (Knights Landing) at 1.40 GHz (16GB MCDRAM, 68 cores)	1 MPI task x 136 OpenMP hyperthreads (2 per core)
gtx1060	Nvidia GeForce GTX 1060 (6GB global memory)	1 MPI task per GPU
p100	Nvidia Tesla P100-PCIe (16GB global memory)	1 MPI task per GPU
v100	Nvidia Tesla V100-PCIe (16GB global memory)	1 MPI task per GPU

Table 1: Description of the tested compute nodes: device description and total RAM size as well as the corresponding distribution of MPI tasks and OpenMP threads/GPU contexts. The configuration of multiple nodes scales the number of MPI tasks; for example 4 skl nodes involve 8 MPI tasks each spawning 24 OpenMP threads running on 192 physical cores.



(a) axpby



(b) dot

Figure 6: Single node memory bandwidth plot of the trivially parallel vector addition (a) and the exact dot product (b) on various hardware architectures on one node (full saturation color), on two nodes (medium saturation) and on four nodes (low saturation). We normalize to the number of nodes, such that the single node performance becomes visible.

sizes). This feature is absent in Fig. 6b, which is most likely due to the high number of 64-bit operations per memory load in the long accumulator scalar product.

For the multi-node architectures we identify a linear regime for small array sizes ($S/n < 2$MB) in both Fig. 6a and 6b. Here, the bandwidth increases linearly with the array size, which indicates a size-independent runtime T_{lat}, called the *latency*. Note that the multi-node results in Fig. 6b indicate an increasing latency for multiple nodes. We explain this by the necessary global communication between nodes due to the reduction, which is absent in the `axpby` algorithm. On the other side, for large array sizes ($S/n > 100$MB) we identify a regime with constant bandwidth B in both Fig. 6a and 6b independent of node number.

We determine B by taking the average bandwidth of the largest array sizes and estimate an error with the standard deviation. This is in general a very robust method and yields small errors in our experience. The correct determination of the latency T_{lat} is more involved with the available data. We differentiate between single-node and multi-node latency. As a first approximation we simply identify the minimum average runtime with T_{lat} and again use the standard deviation as an error estimate. If we now assume that the runtime is given by

$$t = T_{\text{lat}}(n) + \frac{mS}{nB}, \tag{9}$$

then we can correct the minimum runtime t_{min} by $-mS/B$ with the previously measured B to obtain a better approximation to T_{lat}. However, with the exception of the Knights landing architecture the single-node latencies for `axpby` are so small that the values become negative. In this case we replace the value by 0. In Table 2 we give numerical values of the bandwidths together with the latencies as well as the peak bandwidth according to to the vendors. Within the error the `axpby` latencies can be neglected altogether except for the Knights Landing architecture.

We note that the GPUs and the Xeon Phi have the highest latencies in the `dot` algorithm. The high GPU latency is the result of the slow PCIe lanes since the result has to be sent back to the host CPU, which entails communication. As already evident in Figure 6b the latencies on multiple nodes significantly increase for the Xeon Phi and the Skylake architectures. This indicates a possible long latency of the internode connection. For the P100 and V100 GPUs the latency seems to be dominated by

	peak bandwidth [GB/s]	axpby bandwidth [GB/s]	$T_{lat}(1)$ [μs]	$T_{lat}(4)$ [μs]	dot bandwidth [GB/s]	$T_{lat}(1)$ [μs]	$T_{lat}(4)$ [μs]
i5	34	30 ± 01	00 ± 02	n/a	10 ± 01	05 ± 01	n/a
gtx1060	192	158 ± 01	00 ± 01	n/a	27 ± 01	93 ± 09	n/a
skl	256	207 ± 06	00 ± 01	00 ± 01	193 ± 19	18 ± 03	38 ± 05
knl	>400	394 ± 23	06 ± 01	10 ± 01	142 ± 07	55 ± 02	120 ± 06
p100	732	554 ± 01	01 ± 01	03 ± 01	347 ± 02	49 ± 01	49 ± 01
v100	898	849 ± 01	02 ± 01	03 ± 01	594 ± 03	34 ± 02	35 ± 01

Table 2: Measured single node bandwidth and latency on a single node and four nodes of `apxby` and `dot`. The peak bandwidth is the theoretical RAM bandwidth according to the vendors. The exact peak bandwidth of the MCDRAM on knl was not disclosed to us.

the communication between GPU and host CPU.

Finally, we note that all architectures reach only 75% (p100) to 95% (v100) of their theoretical peak bandwidth in the `axpby` function. This is in line with previous observations of the STREAM benchmark [14]. We do not know of any practical method to overcome this performance degradation programmatically and consider the measured bandwidth $B^{\texttt{axpby}}$ as the maximum bandwidth any memory bound algorithm can achieve on the given architecture. In this sense, the Skylake architecture reaches almost 100% efficiency for the `dot` algorithm, followed by the Tesla cards P100 and V100. The GTX 1060 has the lowest efficiency, which is most likely due to the drastic reduction of double precision performance on the gaming GPU (a factor 32 compared to single precision), which is absent in the Tesla GPUs.

For the matrix-vector product we perform the same analysis and show the results In Table 3. There, we present the average bandwidth between a dG derivative in the x-direction and the y-direction, which we call `dxdy`. Due to our efficient format the matrix itself does not contribute to the memory loads and stores. We count two loads and one store for the $y \leftarrow \alpha Mx + \beta y$ operation ($m = 3$). However, we do differentiate between various polynomial orders, which determines the *stencil* of the operation. A higher polynomial order increases the registry pressure and thus decrease the efficiency of the implementation. The latency should not be influenced by the polynomial order and we provide the latencies for the $P = 2$ case. We observe the highest latencies for the multi-node configurations. Here, the algorithm involves communication between neighboring processes. This is particularly unfavourable for the GPUs since these have to communicate via the host CPU across the PCIe lanes.

Concerning the single node bandwidths B we overall observe the highest values for the Tesla GPUs. It is noteworthy that the GTX 1060 has a very low latency and reaches almost 70% of the bandwidth of the much more expensive Skylake and Xeon Phi nodes.

5.4. Performance prediction model

Any one of the primitive subroutines in Level 1 in Feltor falls into one of the categories 'trivially parallel' (`axpby`), 'nearest neighbor communication' (`dxdy`) and 'global communication' (`dot`). We specifically measured the bandwidths and latencies of the three operations `axpby`, `dxdy` and `dot` in Tables 2 and 3. For the following discussion we assume that these values accurately represent the bandwidths and latencies of the whole respective class of functions. In fact, we use these values to predict the runtime of any algorithm that is implemented in terms of Level 1 subroutines. For a given architecture and node number n we predict a runtime t depending on the array size S and the number of polynomial coefficients P

$$t(P,S,n) = \sum_q \sum_{i=0}^{N_q-1} t_i^q(P,S,n) = \sum_q \left[N_q T^q_{\text{lat}}(n) + \frac{M^q S}{n B^q(P)} \right]$$
$$=: N \left[T_{\text{lat}}(n) + \frac{M}{N} \frac{S}{n B(P)} \right] \tag{10}$$

$$T_{\text{lat}}(n) := \frac{1}{N} \sum_q N_q T^q_{\text{lat}}(n) \tag{11}$$

$$\frac{1}{B(P)} := \frac{1}{M} \sum_q \frac{M^q}{B^q(P)} \tag{12}$$

with the function type $q \in$(`axpby`,`dot`,`dxdy`), i iterates over all occurrences of function type q, N_q is the total number of occurrences of all functions of type q, M^q is the total number of memory loads and stores among functions of type q, $B^q(P)$ is the single node memory bandwidth of function type q and $T^q_{\text{lat}}(n)$ is the latency depending on the number of nodes used. In Eqs. (11) and (12) we defined the average latency and weighted average single node bandwidth, where $N := \sum_q N_q$ and $M := \sum_q M^q$. The values for $B^q(P)$ and $T^q_{\text{lat}}(n)$ are in Table 2 and 3. We present an average over a conjugate gradient iteration and the Arakawa algorithm in Table 4. These two algorithms represent a typical mixture of primitive functions used in a Feltor simulation project. In fact, we get a first approximation of the runtime of any algorithm by counting the total number of function calls N and using Eq. (10), Table 4 and $M/N \approx 3.3$.

In Fig. 7 we compare the result of the prediction in Eq. (10) with the measured runtime for the `arakawa` algorithm and one `cg` iteration. The plots depict the relative error of the prediction. If a point lies below 1, the execution was faster than predicted. Especially for large array sizes $S/n >$ 30MB our prediction is accurate for all architectures. We note in both Fig. 7a and Fig. 7b that the measured runtime for the Xeon Phi card on multiple nodes for sizes $S/n <$ 20MB is systematically over-

	B(P=2) [GB/s]	B(P=3) [GB/s]	B(P=4) [GB/s]	B(P=5) [GB/s]	$T_{lat}(1)$ [μs]	$T_{lat}(4)$ [μs]
i5	28 ± 03	30 ± 03	26 ± 02	22 ± 02	00 ± 02	n/a
gtx1060	131 ± 01	112 ± 02	84 ± 14	70 ± 18	00 ± 01	n/a
skl	182 ± 36	162 ± 13	119 ± 19	111 ± 09	23 ± 03	29 ± 03
knl	240 ± 18	173 ± 27	127 ± 19	102 ± 15	10 ± 01	53 ± 04
p100	288 ± 03	238 ± 04	201 ± 02	166 ± 15	02 ± 01	64 ± 01
v100	802 ± 17	713 ± 20	650 ± 16	536 ± 49	04 ± 01	67 ± 02

Table 3: Single node bandwidth of a dG matrix-vector multiplication for various polynomial coefficients P. Latencies on a single node/card and four nodes/cards.

	B(P=2) [GB/s]	B(P=3) [GB/s]	B(P=4) [GB/s]	B(P=5) [GB/s]	$T_{lat}(1)$ [μs]	$T_{lat}(4)$ [μs]
i5	26 ± 02	27 ± 02	26 ± 01	23 ± 02	01 ± 01	n/a
gtx1060	116 ± 01	108 ± 01	94 ± 09	85 ± 12	09 ± 01	n/a
skl	194 ± 20	183 ± 09	153 ± 15	147 ± 07	14 ± 02	19 ± 02
knl	281 ± 13	232 ± 24	188 ± 20	160 ± 18	13 ± 01	42 ± 02
p100	377 ± 02	333 ± 04	297 ± 02	259 ± 17	06 ± 01	39 ± 01
v100	808 ± 09	763 ± 11	727 ± 10	653 ± 35	06 ± 01	39 ± 01

Table 4: Average single node bandwidths B for various polynomial coefficients P as well as average single-node and multi-node latencies according to Eq. (10). We use Table 2 and 3, $(N^{\texttt{axpby}}, N^{\texttt{dot}}, N^{\texttt{dxdy}}) = (9, 2, 12)$ and $(M^{\texttt{axpby}}, M^{\texttt{dot}}, M^{\texttt{dxdy}}) = (36, 4, 36)$, $N = 23$, $M = 76$ and a ratio of $M/N = 3.30$. This corresponds to the average between a conjugate gradient iteration and the Arakawa algorithm.

estimated. On the other side the Skylake architecture and the Intel i5 CPU for array sizes $S/n < 30$MB run up to a factor 2 faster than predicted. We explain this by the very fast execution of the trivially parallel part of the algorithms in the fast cache as is evident in Fig. 6a. This effect is not included in our parallel model. On the other hand, the measured run times T^{meas} for the Tesla GPUs and our desktop system are remarkably well predicted by our model and with only few exceptions lie within an interval $(3/4)T^{\text{pred}} < T^{\text{meas}} < (4/3)T^{\text{pred}}$, with T^{pred} given by Eq. 10.

5.5. Strong and weak scaling

Equation (10) enables us to discuss the strong and weak scaling of an arbitrary algorithm. The strong scaling of a problem with total array size S, polynomial coefficients P and number of nodes n is defined as

$$\varepsilon(P, S, n) := \frac{t(P, S, 1)}{nt(P, S, n)} = \frac{T_{\text{lat}}(1) + (M/N)(S/B(P))}{nT_{\text{lat}}(n) + (M/N)(S/B(P))} \quad (13)$$

The weak scaling efficiency relates run times with equal array size per node $s = S/n$ as

$$\gamma(P, s, n) := \frac{t(P, s, 1)}{t(P, ns, n)} = \frac{T_{\text{lat}}(1) + (M/N)(s/B(P))}{T_{\text{lat}}(n) + (M/N)(s/B(P))} \quad (14)$$

We immediately see that the efficiency ε tends to 0 for large number of nodes n, while the explicit n dependency in γ vanishes. The only remaining dependence on n is in the latency $T_{\text{lat}}(n)$. We argue that the dependence on n should vanish in the latencies for `axpby`, since there is no communication at all. For the `dxdy` algorithm the latencies should also become independent of n for large n since communication happens only between nearest neighbors. Only for the `dot` product the latency should increase with n due to the global communication.

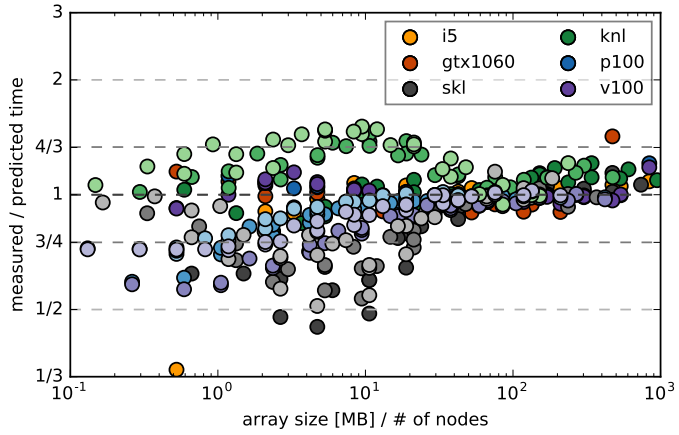
Both the strong and the weak scaling tend to unity if $s = S/n \gg (N/M)T_{\text{lat}}(n)B(P)$. The value of the product

$$(S/n)_{\min} \approx 0.3T_{\text{lat}}(n)B(P) \quad (15)$$

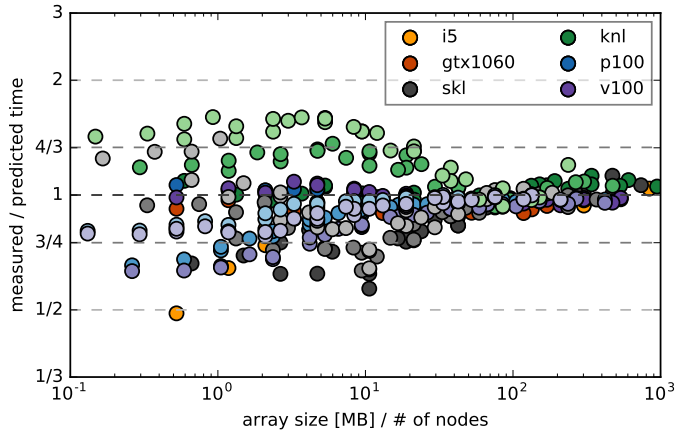
can thus be taken as minimum size per node for an efficient Feltor simulation. For the values presented in Table 4 the minimum array size per node typically lies between 1 and 10MB.

5.6. Discussion

From Eq. (10) it is clear that the runtime t is low if the average latency T_{lat} is low and the bandwidth B is high. On the other side, performance can also be gained by reducing the number of function calls N, or the number of memory operations M. Apparently, the fastest possible implementation is to implement the whole algorithm in a single function, with $N = 1$ and a minimum number of memory operations $M_{\min}$. For example, our current `arakawa` implementation has $M = 34$ and $N = 9$, which compares unfavourably to a possible $N_{\min} = 1$ and $M_{\min} = 4$. The drawback of implementing and optimizing every algorithm or equation separately is the increased maintenance and performance tuning cost. Furthermore, this approach would not be easily extensible or modifiable and violates our design goals presented in Section 2. Still, we estimate the performance we loose due to the Feltor design between a factor 2 and 5 depending on the algorithm at hand. In an effort to mitigate the problem we introduced new primitive functions with increased workload, for example the vector operation $z \leftarrow \alpha x_1 y_1 + \beta x_2 y_2 + \gamma z$, where $x_1 y_1$ is a point-wise multiplication. We currently also explore template parameter packs in the C++-11 standard as a promising candidate to increase the workload of Level 1 functions in Feltor.



(a) arakawa



(b) cg

Figure 7: Comparison of predicted runtime Eq. (10) to measured time for the Arakawa algorithm (a) and a single conjugate gradient (cg) iteration (b). The plot highlights the deviation from the predicted time. Points below 1 mean faster execution than predicted.

6. Discussion and Conclusion

With Table 4 and Eq. 10 in Section 5 we have a powerful tool to judge the performance of various hardware architectures available to us. We are able to estimate the runtime of a Feltor simulation for given problem size, hardware and node count. From a user perspective the possibility to predict performance is certainly a valuable feature since available resources can be used more effectively and the performance of future hardware can be estimated from its theoretical bandwidth.

Let us here discuss performance also in the light of the simulations we eventually want to carry out. In fact, an increase/decrease in performance of the implementation may lead to an only marginal improvement/deterioration of the numerical simulations itself. Consider for example a three dimensional problem and an available runtime T (set by cluster policies, allocated resources or simply personal preferences). Accounting for the reduced/increased time step due to the CFL condition a factor 2 increase/decrease in performance leads to only a factor $2^{1/4} \approx 1.19$ increase/decrease in the number of grid points per dimension. This justifies our design goals laid out in Section 2. We do strife for performance but when faced with possible trade-off scenarios we put equal value on other goals as well.

Of course, the choice of the physical model and the numerical methods employed ultimately set the limit of what an implementation can achieve in terms of performance. As discussed in Section 4 in discontinuous Galerkin methods the order of the method is a free parameter. In Section 5 we argue that a higher order method executes slower than a lower order method with the same number of degrees of freedom due to the increased stencil. At the same time, the high order method may also require less points overall to achieve the same resolution and accuracy as the lower order method. The minimum requirements of what a simulation has to resolve is eventually given by the spatial and temporal scales in the physical dynamics.

Another consideration is the question of when a simulation is "converged". As we argue in Section 3 this question can be difficult to answer. In ill-conditioned problems only reduced physical quantities of interest or invariants may be valid indicators. Pointwise convergence may be lost altogether and reproducibility has to be reconsidered. We show that we algorithmically and programmatically achieve bitwise reproducibility of results in Section 3 by ensuring deterministic execution of elementary subroutines in parallel environments. Again, in Section 5 we show that these changes, which may be viewed as restrictions, still allow high performant simulations on various architectures.

This discussion hopefully provides the reader with the tools necessary to justify an appropriate setup for a numerical simulation and although this article was primarily written on Feltor we think that our arguments holds for any similar simulation framework as well.

Acknowledgements

We thank Harald Servat, from the Intel Corporation, for the help provided in optimizing Feltor on the Intel Xeon

Phi "Knights landing" and Skylake architectures. We thank Siegfried Höfinger from the Technical University of Vienna for mediating the contact to Nvidia. We acknowledge the support of Nvidia PSG Cluster with the donation of the Tesla P100 and V100 GPUs used for this research.

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement no. 713683 (COFUNDfellowsDTU). This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

References

[1] Feltor homepage. `https://feltor-dev.github.io`.

[2] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[3] A. Arakawa. Computational design for long-term numerical integration of the equations of fluid motion: Two-dimensional incompressible flow. Part I. *J. Comput. Phys.*, 1(1):119–143, 1966.

[4] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified analysis of discontinuous galerkin methods for elliptic problems. *SIAM J. Numer. Anal.*, 39(5):1749–1779, January 2002.

[5] S. Ashby et al. The opportunities and challenges of exascale computing. *Report of the ASCAC Subcommittee on Exascale Computing*, 2010.

[6] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Comput. Phys. Commun.*, 180(12):2526–2533, 2009.

[7] M. A. Beer and G. W. Hammett. Toroidal gyrofluid equations for simulations of tokamak turbulence. *Phys. Plasmas*, 3(11):4046, 1996.

[8] Braginskii. Transport processes in a plasma. *Rev. Plasma Phys.*, Vol. 1, 1965.

[9] G. Chen, L. Chacón, and D. C. Barnes. An efficient mixed-precision, hybrid CPU–GPU implementation of a nonlinearly implicit one-dimensional particle-in-cell algorithm. *J. Comput. Phys.*, 231(16):5374–5388, 2012.

[10] B. Cockburn and C. W. Shu. The local discontinuous Galerkin method for time-dependent convection-diffusion systems. *SIAM J. Numer. Anal.*, 35(6):2440–2463, November 1998.

[11] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk. Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Parallel Computing*, 49:83–97, 2015.

[12] N. Crouseilles, L. Einkemmer, and M. Prugger. An exponential integrator for the drift-kinetic model. *Comput. Phys. Commun.*, 2017.

[13] S. Dalton, N. Bell, L. Olson, and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. Version 0.5.0.

[14] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. GPU-stream v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. *High Performance Computing, Isc High Performance 2016 International Workshops*, 9945:489–507, 2016.

[15] B. D. Dudson, A. Allen, G. Breyiannis, E. Brugger, J. Buchanan, L. Easy, S. Farley, I. Joseph, M. Kim, A. D. McGann, J. T. Omotani, M. V. Umansky, N. R. Walkden, T. Xia, and X. Q. Xu. Bout plus plus : Recent and current developments. *J. Plasma Phys.*, 81:365810104, January 2015.

[16] L. Einkemmer. A mixed precision semi-Lagrangian algorithm and its performance on accelerators. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*, pages 74–80, 2016.

[17] L. Einkemmer and M. Wiesenberger. A conservative discontinuous Galerkin scheme for the 2D incompressible Navier–Stokes equations. *Comput. Phys. Commun.*, 185(11):2865–2873, 2014.

[18] A. Fasoli, S. Brunner, W. A. Cooper, J. P. Graves, P. Ricci, O. Sauter, and L. Villard. Computational challenges in magnetic-confinement fusion physics. *Nature Physics*, 12(5):411–423, May 2016.

[19] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, 23(1):5–48, March 1991.

[20] E. Hairer, C. Lubich, and G. Wanner. *Geometric numerical integration: structure-preserving algorithms for ordinary differential equations*, volume 31. Springer, 2006.

[21] F. D. Halpern, P. Ricci, S. Jolliet, J. Loizu, J. Morales, A. Mosetto, F. Musil, F. Riva, T. M. Tran, and C. Wersal. The gbs code for tokamak scrape-off layer simulations. *J. Comput. Phys.*, 315:388–408, June 2016.

[22] F Hariri and M Ottaviani. A flux-coordinate independent field-aligned approach to plasma turbulence simulations. *Comput. Phys. Commun.*, 184(11):2419–2429, 2013.

[23] A. Hasegawa and M. Wakatani. Plasma edge turbulence. *Phys. Rev. Lett.*, 50:682, 1983.

[24] A. Hasegawa and M. Wakatani. Self-organization of electrostatic turbulence in a cylindrical plasma. *Phys. Rev. Lett.*, 59:1581–1584, October 1987.

[25] M. Held. *Full-F gyro-fluid modelling of the tokamak edge and scrape-off layer*. PhD thesis, 2016.

[26] M. Held, M. Wiesenberger, R. Kube, and A. Kendl. Non-oberbeck-boussinesq zonal flow generation. *Nucl. Fusion*, (under review), 2018.

[27] M. Held, M. Wiesenberger, J. Madsen, and A. Kendl. The influence of temperature dynamics and dynamic finite ion larmor radius effects on seeded high amplitude plasma blobs. *Nucl. Fusion*, 56(12):126005, 2016.

[28] M. Held, M. Wiesenberger, and A. Stegmeir. Three discontinuous galerkin schemes for the anisotropic heat conduction equation on non-aligned grids. *Comput. Phys. Commun.*, 199:29–39, February 2016.

[29] Nicholas J. Higham. *Accuracy and stability of numerical algorithms, second ed.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002.

[30] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat. ExBLAS: Reproducible and accurate BLAS library. In *Proceedings of the Numerical Reproducibility at Exascale (NRE2015) workshop held as part of the Supercomputing Conference (SC15). Austin, TX, USA, November 15-20, 2015*, October 2015.

[31] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat. ExBLAS (Exact BLAS) library. Available on the WWW, `https://exblas.lip6.fr/`, 2018. Accessed 10-MAR-2018.

[32] R. Iakymchuk, S. Graillat, D. Defour, and E. S. Quintana-Ortí. Hierarchical Approach for Deriving a Reproducible LU factorization. Technical report, 2018. Available on the WWW, `https://hal.archives-ouvertes.fr/hal-01419813`. Accessed 14-June-2018, HAL ID: hal-01419813.

[33] Intel. Intel® architecture instruction set extensions programming reference. Available on the WWW, https://software.intel.com/en-us/intel-architecture-instruction-set-extensions-programming-reference, 2018. Accessed 7-MAY-2018.

[34] A. Kendl. Modelling of turbulent impurity transport in fusion edge plasmas using measured and calculated ionization cross sections. *International Journal of Mass Spectrometry*, 365:106–113, May 2014.

[35] A. Kendl, G. Danler, M. Wiesenberger, and M. Held. Interchange instability and transport in matter-antimatter plasmas. *Phys. Rev. Lett.*, 118(23):235001, June 2017.

[36] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, third ed.* Addison-Wesley, 1997.

[37] M. Kraus. *Variational Integrators in Plasma Physics*. PhD thesis, Technische Universität München, 2013.

[38] R. Kube, O. E. Garcia, and M. Wiesenberger. Amplitude and size scaling for interchange motions of plasma filaments. *Phys. Plasmas*, 23(12):122302, 2016.

[39] U. Kulisch and V. Snyder. The Exact Dot Product As Basic Tool for Long Interval Arithmetic. *Computing*, 91(3):307–313, March 2011.

[40] E. Madaule, S.A. Hirstoaga, M. Mehrenberger, and J. Pétri. Semi-Lagrangian simulations of the diocotron instability. *hal preprint, hal-00841504*, 2013.

[41] J. Madsen. Full-f gyrofluid model. *Phys. Plasmas*, 20(7):072301, July 2013.

[42] J. Madsen, V. Naulin, A. H. Nielsen, and J. J. Rasmussen. Collisional transport across the magnetic field in drift-fluid models. *Phys. Plasmas*, 23(3):032306, March 2016.

[43] S. Meyers. *Effective C++, Third Edition*. Addison-Wesley Professional,

2005.
[44] S. Meyers. *Effective Modern C++*. O'Reilly Media, 2014.
[45] J. M. Muller, N. Brisebarre, F. de Dinechin, C. P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.
[46] Yoichiro Nambu. Generalized hamiltonian dynamics. In *Broken Symmetry: Selected Papers of Y Nambu*, pages 302–309. World Scientific, 1995.
[47] R. Numata, R. Ball, and R. L. Dewar. Bifurcation in electrostatic resistive drift wave turbulence. *Phys. Plasmas*, 14:102312, 2007.
[48] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput*, 26, 2005.
[49] J. J. Rasmussen, A. H. Nielsen, J. Madsen, V. Naulin, and G. S. Xu. Numerical modeling of the transition from low to high confinement in magnetically confined plasma. *Plasma Phys. Control. Fusion*, 58(1):014031, January 2016.
[50] B. Scott. Derivation via free energy conservation constraints of gyrofluid equations with finite-gyroradius electromagnetic nonlinearities. *Phys. Plasmas*, 17(10):102306, October 2010.
[51] A. Stegmeir, D. Coster, O. Maj, and K. Lackner. Numerical methods for 3D tokamak simulations using a flux-surface independent grid. *Contrib. Plasma Phys.*, 54(4-6):549–554, June 2014.
[52] P. Tamain, P. Ghendrih, E. Tsitrone, V. Grandgirard, X. Garbet, Y. Sarazin, E. Serre, G. Ciraolo, and G. Chiavassa. Tokam-3D: A 3D fluid code for transport and turbulence in the edge plasma of tokamaks. *J. Comput. Phys.*, 229(2):361–378, January 2010.
[53] M. Wakatani and A. Hasegawa. A collisional drift wave description of plasma edge turbulence. *Phys. Fluids*, 27:611, 1984.
[54] M. Wiesenberger. *Gyrofluid computations of filament dynamics in tokamak scrape-off layers*. PhD thesis, 2014.
[55] M. Wiesenberger. Feltor Performance Dataset. *Zenodo* `http://doi.org/10.5281/zenodo.1290274`, 2018.
[56] M. Wiesenberger and M. Held. Feltor v5.0. *Zenodo* `http://doi.org/10.5281/zenodo.1290270`, 2018.
[57] M. Wiesenberger, M. Held, and L. Einkemmer. Streamline integration as a method for two-dimensional elliptic grid generation. *J. Comput. Phys.*, 340:435–450, 2017.
[58] M. Wiesenberger, M. Held, L. Einkemmer, and A. Kendl. Streamline integration as a method for structured grid generation in x-point geometry. *J. Comput. Phys.*, (under review), 2018.
[59] M. Wiesenberger, M. Held, R. Kube, and O E Garcia. Unified transport scaling laws for plasma blobs and depletions. *Phys. Plasmas*, 24(6):064502, 2017.
[60] M. Wiesenberger, J. Madsen, and A. Kendl. Radial convection of finite ion temperature, high amplitude plasma blobs. *Phys. Plasmas*, 21:092301, 2014.
[61] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J. W. Boiten, L. B. D. Santos, P. E. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. T. Evelo, R. Finkers, A. Gonzalez-Beltran, A. J. G. Gray, P. Groth, C. Goble, J. S. Grethe, J. Heringa, P. A. C. 't Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S. J. Lusher, M. E. Martone, A. Mons, A. L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S. A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, and B. Mons. Comment: The fair guiding principles for scientific data management and stewardship. *Scientific Data*, 3:UNSP 160018, March 2016.
[62] A. Zeiler, J. F. Drake, and B. Rogers. Nonlinear reduced Braginskii equations with ion thermal dynamics in toroidal plasma. *Phys. Plasmas*, 4(6):2134–2138, 1997.