Fachhochschule Aachen
Campus Jülich

Department: Medizintechnik und Technomathematik
Degree program: Angewandte Mathematik und Informatik

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Parareal in Julia with Singularity

Masterthesis from

Isabel Heisters

Jülich, December 2020

Declaration of independence

This work is made and written by myself. No other sources and tools have been used than those indicated.

Place and date

Signature

The thesis was supervised by:

First examiner: Prof. Dr. Johannes Grotendorst
Second examiner: Dr. Robert Speck

This thesis was done at :

Jülich Supercomputing Centre
Forschungszentrum Jülich GmbH

Abstract

In this thesis a Julia version of the parallel-in-time method Parareal is introduced. Parareal decomposes the time as an approach for parallelization. Parareal is an hierarchical, iterative algorithm that uses a coarse, cheap integrator to propagate information quickly forward in time in order to provide initial values for the parallelized original time integration scheme. The problems here used to test the Parareal algorithm are the Lorenz equation, the heat equation and the Allen-Cahn equation. Julia is a programming language that was specifically designed to be used for numerical applications and parallelization. Julia is becoming more popular due to the fact that it is easier to implement than C but has a better runtime than Python. Julia is a new language and not available on most host systems. Singularity is a container solution to create the necessities for scientific application-driven workloads. By using a container the user can configure the environment in which the application can run independently of the host system and its software specifications.

This thesis shows how a Singularity container for the Parareal algorithm implemented in Julia can be built. The container is portable to different hosts like HPC and cloud systems without having a runtime overhead in comparison to the runtime without a container.

Zusammenfassung

In dieser Arbeit wird der Parareal-Algorithmus als Werkzeug zur Parallelisierung vorgestellt. Parareal zerlegt die Zeit als Ansatz zur Parallelisierung. Parareal ist ein hierarchischer, iterativer Algorithmus, der einen günstigen Integrator verwendet, um Informationen schnell in der Zeit vorwärts zu senden, um Anfangswerte für das parallelisierte ursprüngliche Zeitintegrationsschema bereitzustellen. Die Probleme, die hier zum Testen des Parareal-Algorithmus verwendet werden, sind die Lorenz-Gleichung, die Wärmeleitungsgleichung und die Allen-Cahn-Gleichung. Julia ist eine Programmiersprache, die speziell für numerische Anwendungen und Parallelisierung entwickelt wurde. Julia wird immer beliebter, da sie einfacher zu implementieren ist als C, aber eine bessere Laufzeit als Python hat. Julia ist eine neue Sprache und auf den meisten Host-Systemen nicht verfügbar. Singularity ist eine Containerlösung, mit dem Ziel die Vorraussetzung für wissenschaftliche Projekte zu schaffen. Durch die Verwendung eines Containers kann der Benutzer die Umgebung konfigurieren, in der die Anwendung unabhängig vom Hostsystem und dessen Softwarespezifikationen laufen kann.

Diese Arbeit zeigt, wie ein Singularity-Container für den in Julia implementierten Parareal-Algorithmus gebaut werden kann. Der Container ist auf verschiedene Hosts wie HPC- und Cloud-Systeme portierbar, ohne einen Laufzeit-Overhead im Vergleich zur Laufzeit ohne Container zu haben.

Contents

List of Figures

List of Tables

Listings

1. Motivation

Classical numerical methods to solve initial value problems run serial in time, which means the time is divided into individual time steps and one step is processed after the next. To speed up solving the problems mostly parallelization in the spatial domain is used especially since supercomputers are easier to use parallelization as a tool to reduce the run time is very common. The fast development of supercomputers leads to limitations where more computing power cannot save you run time. An idea to solve this problem is a parallelization in the time domain.

Most of the time when working on parallelization C or Python is used. C is popular for achieving a high execution speed. In Python on the other hand it is easy to write code. A new language that is trying to combine these two benefits is Julia. Julia is a programming language that was specifically designed to be used for numerical applications and high-performance computing. Julia [Bez+14] is a dynamic high-performance programming language. The first stable open-source version was released in February 2012, the development of the language began in 2009. Julia is mainly used in scientific computing.
The language Julia is in contrast to other programming languages almost exclusively written in Julia itself. Only basic functions, such as integer operations, for loops, recursion or float operations, use C operations or work with C structures. Julia works with a Just-in-Time (JIT) compiler. The code is converted to machine code during compilation. Just-in-time compilation differs from so-called Ahead-of-Time (AOT) compilations in that these programs compile during run time, whereas AOT compilers compile before run time. The parser, which converts the Julia code to machine code, is implemented in Scheme [Dyb09] and is converted to optimized machine code using the LLVM compiler framework [LA04], a virtual machine. For this purpose, the framework uses the compiler backend of Clang for C/C++, which provides excellent auto-SIMD vectorization and can lead to high performance. This happens even though Julia is a dynamic language since this usually only occurs in static programming languages. Julia was also developed to make parallel implementation, whether shared memory computing or distributed memory computing, easier.

Julia programs can be executed from the command line and Julia as a language offers an interactive session. In this session, commands can be tested, programs and normal shell commands can be executed. The interactive session can also be used to manage Julia's packages. Julia can be developed

in different environments. One possibility is to use the development environment Juno, which is specially designed for Julia. It is based on Atom [Git], an open-source text editor developed by the project hosting service GitHub for Windows, macOS, and Linux. It is also possible to write Julia in Jupyter notebooks [Jup], which are based on the principle of the IPython notebook. This means that the Jupyter notebook can not only work with Julia programs but also Markdown and up to 40 other programming languages.

Julia is a new programming language and not available on all host systems. Therefor Singularity can be used as a tool to create an environment in which the application can run. Singularity is a container solution with the goal to create the necessities for scientific application driven workloads. Inside a container an independent environment which can differ from the host can be executed. This can be of an advantage, because Julia is a new programming language and is not configured on all host systems. Additionally, by using a container the user can configure all packages for example in Julia independently and is not dependent on the host systme like a supercomputer. The goal is that the run time of the application with Singularity should be approximately the same as without the container.

The goal of this work is to have a working Parareal algorithm which can be applied to different problems written in Julia that runs on the Supercomputer Jureca with MPI inside a Singularity container without a significant timing overhead. The container should be portable to different hosts as well. The implementation should be easy to adjust for different problems and different solvers.

2. The Parareal algorithm

In this chapter, the Parareal algorithm is introduced. First, a review is given on what was published in the past. After that, the algorithm is presented and analyzed. Finally the parallelization strategy is introduced as well as a review of how efficient the parallelization is.

2.1. Classification in literature

The following introduction is based on [GV07; GH08]. The time-parallel algorithm Parareal was published in 2001 by Lions et al. [LMT01], to solve initial value problems parallel not in space but in "real time". This motivation led to the name for the algorithm, Parareal, as a composition of parallel and real time.

The first idea for parallelizing initial value problems in the time domain was published in 1964 by Nievergelt [Nie64]. In his algorithm, starting with an estimate, more exact approximations can be calculated in parallel on each timestep and these approximations are finally combined in serial. This approach of Nievergelt has been further developed by Bellen and Zennaro and was published in 1989 [BZ89]. Bellen and Zennaro's method is based on iteratively solving the fixed-point equation with the Steffens-Algorithm on a discrete level. After that Chartier and Phillipe [CP93] and Saha et. al. [SST97] published in 1993 and 1996 algorithms which are also working by iteratively solving the fixed-point equation on a continuous level. The Parareal algorithm can be considered as a special variant of the method of Saha et. al.. To use Parareal, the time has to be decomposed into time intervals. The idea is to calculate for every time interval a solution for an initial value problem in parallel. The required starting value for the initial value problem is calculated in serial with a fast but inaccurate solver. With the starting value, the initial value problem can be solved for each time interval in parallel with a time intensive but more accurate solver. With an iteration rule, Parareal corrects the starting value with a more accurate computed value.

The convergence analysis of Parareal was published in 2008 in [GH08]. They showed the superlinear convergence of the algorithm for a system of nonlinear differential equations mathematical and also with the help of different examples, such as the Lorenz equation, numerically. The stability of Parareal for autonomous systems was investigated in [SR05].

In the following, some current applications of Parareal are shown to highlight the relevance of the algorithm. Ariel. et. al. [GT15] successfully used Parareal for the solution of strongly oscillating differential equations by combining it with solvers for multi-scale calculations. Baudron et. al. [Bau+14] used Parareal to parallelize a solver for the neutron diffusion equation. In [KR14] Parareal was combined with a space-parallel, time-serial solver for the solution of the Burger's equation. It has been shown that the speedup of the combined space and time parallel solver is better than the space parallel version. In the publication by Ruprecht et. al. [RSK16] the algorithm was applied to diffusion problems, such as the heat equation. Good convergence properties were shown due to the variation of the problem coefficients. Clarke et.al. [Cla+19] used the Parareal algorithm to speed up the simulations on the natural dynamos in the Earth and Sun.

2.2. Mathematical approach

A problem to solve with the Parareal algorithm is for example a time-dependent initial value problem of the type

$$q_t(t) = -f(q), q(0) = q_0, t \in [0, T]. \tag{2.1}$$

Parareal as a method to parallelize the time needs a decomposition of the time $[0, T]$ into n_c intervals $[t_i, t_{i+1}]$ with equal step size $\Delta t = t_{i+1} - t_i, i \in 0, .., n_g$ with $0 = t_0 < ... < t_i < t_{i+1} < ... < t_n = T$. The algorithm involves as described before in section 2.1 two solvers. An inaccurate one referred to as $\mathcal{G}$ and an more accurate one referred to as $\mathcal{F}$. Both solvers are usually single-step methods and are determined by integration schemes. $\mathcal{G}$ computes the solution on the time intervals with Δt, i.e. $\mathcal{G}$ is now referred to as $\mathcal{G}_{\Delta t}$. $\mathcal{F}$ computes a more accurate solution on Δt. To do this the interval is also divided into n_f sub intervals so that $\delta t = \frac{1}{n_f}\Delta t$. This means $\mathcal{F}$ is now referred to as $\mathcal{F}_{\delta t}$.
The initial value q^0 is determined for every interval with

$$q^0_{i+1} = \mathcal{G}_{\Delta t}(q^0_i, t_{i+1}, t_i),$$

i is the number of the timestep. For every iteration step

$$q^{k+1}_{i+1} = \mathcal{G}_{\Delta t}(q^{k+1}_i, t_{i+1}, t_i) + \mathcal{F}_{\delta t}(q^k_i, t_{i+1}, t_i) - \mathcal{G}_{\Delta t}(q^k_i, t_{i+1}, t_i)$$

is calculated, where k is the current iteration number.

In algorithm 1 the process of the calculation is shown. In lines $1 - 4$ the starting value q^0_{i+1}, $i \in [0, n_c - 1]$ is calculated for each timeinterval (t_i, t_{i+1}) using the starting value of the previous interval q^0_i and the inaccurate solver $\mathcal{G}_{\Delta t}$. The main calculation is done in lines $6 - 14$. First for every timeinterval (t_i, t_{i+1}) the calculation with the more accurate solver $\mathcal{F}_{\delta t}$ is done. This can

Algorithm 1 Parareal Algorithm

1: $q_0^0 = q_0$
2: **for** $i = 0$ to $n_c - 1$ **do**
3: $\quad q_{i+1}^0 = \mathcal{G}_{\Delta t}(q_i^0, t_{i+1}, t_i)$
4: **end for**
5: $k = 0$
6: **repeat**
7: $\quad$ **for** $i = 0$ to $n_c - 1$ **do** ▷ Parallel Step
8: $\quad\quad \tilde{q}_{i+1}^{k+1} = \mathcal{F}_{\delta t}(q_i^k, t_{i+1}, t_i)$
9: $\quad$ **end for**
10: $\quad$ **for** $i = 0$ to $n_c - 1$ **do**
11: $\quad\quad q_{i+1}^{k+1} = \mathcal{G}_{\Delta t}(q_i^{k+1}, t_{i+1}, t_i) + \tilde{q}_{i+1}^{k+1} - \mathcal{G}_{\Delta t}(q_i^k, t_{i+1}, t_i)$
12: $\quad$ **end for**
13: $\quad k = k + 1$
14: **until** $k = N_{maxit}$ or $r^k < \varepsilon$

be done in parallel, because for the calculation only values of the calculated timestep are needed. This differs from the other steps. Therefor a fine solver can be used in line 8. After that in line 11 q_{i+1}^{k+1} is calculated. As a condition to stop the calculations in line 14 either the number of iterations or an error err^k, where

$$err^k = \max_{i=1,..,n_c} ||q_{i+1}^{k+1} - q_{i+1}^k||$$

is used. This means a maximum number of iterations is set, but if $err^k < \varepsilon$ the iterations can be stopped earlier. ε is an user-defined value, which can be regarded as the upper bound for the error.

Parareal can historically be seen as a multigrid method or a predictor-corrector method. So far it has been seen from the perspective of the predictor-corrector method. The inaccurate solver $\mathcal{G}$ predicts the value and the more accurate one $\mathcal{F}$ corrects the solution. Gander and Vandewalle [GV07] showed that Parareal can be classified as a time-multigrid method. Multigrid methods are a group of algorithms where a numerical problem, most of the time a partial differential equation (PDE), is solved using a hierarchy of discretizations. Multigrid methods perform a series of approximation u^k for $k = 1, 2, ...$ from a given initial value on a grid Ω_h with step size h. A_h is the discretization of the PDE on this grid. Multigrid methods are a class of iterative methods. For every iteration, first a smoothing step is performed. The smoothing is determined by the smoothing operator S. The smoothed approximation is called $\tilde{u}^k$. After this, the problem is transferred onto a coarser grid Ω_H with the restriction parameter I_h^H. On the coarser grid, the solution is approximated to solve the problem. The approximated solution on the coarse grid is called U^{k+1}. The approximation is transferred back to the original fine grid with the

prolongation operator I_H^h and corrected with the smoothed approximation $\widetilde{u}^k$. An iteration of the two grid variant of the multigrid method can be written as

$$\begin{aligned} \widetilde{u}^k &= S(u^k, b) && (2.2)\\ A_H(U^{k+1}) &= I_h^H(b - A_h(\widetilde{u}^k)) + A_H(I_h^H \widetilde{u}^k) && (2.3)\\ u^{k+1} &= \widetilde{u}^k + I_H^h(U^{k+1} - I_h^H \widetilde{u}^k) && (2.4) \end{aligned}$$

Shown in equation (2.2) is the smoothing step. Equation (2.3) defines the coarse grid problem. Equation (2.4) can be seen as the correction step. Gander and Vandewalle [GV07] showed that every operator of the iteration can be selected in such a way that the iteration of a Parareal step can be expressed by the two grid iteration.

The convergence behavior of Parareal is dependent on the convergence properties of the solvers and the problem. To analyse the convergence properties the problem

$$u_t(t) = -f(u), u(0) = u_0, a \in \mathbb{C}, t \in [0, T]. \quad (2.5)$$

is used. For the coarse solver $\mathcal{G}$ the Euler method is used. For the fine solver $\mathcal{F}$ the exact solution of the timestep is used. Lions, Maday and Turinci [LMT01] showed in their original paper that

$$\max_{1 \le i \le n_c} |u_n^k - u(t_n)| \le C_p \Delta t^{k+1},$$

when C_p is a variable that grows with order p. This means that for a fixed iteration k the algorithm behaves in Δt like an $\mathcal{O}(\Delta t^{p+1})$ method. Thereby the case that k goes to infinity is not covered. Gander and Hairer [GH08] showed for this case that if $\mathcal{F}_{\delta t}(q_i^k, t_{i+1}, t_i)$ is denoted as the exact solution at t_{i+1} as before and $\mathcal{G}_{\Delta t}(q_i^k, t_{i+1}, t_i)$ is a one step method with local truncation error bounded by $C_1 \Delta t^{p+1}$ and if

$$|\mathcal{G}(x, t, t + \Delta t) - \mathcal{G}(y, t, t + \Delta t)| \le (1 + C_2 \Delta t)|x - y|$$

the following estimate holds:

$$\max_{1 \le n \le n_c} |u(t_n) - U_n^k| \le \frac{C_1 \Delta t^{k(p+1)}}{k!} (1 + C_2 \Delta t)^{n_c - 1 - k} \prod_{j=1}^{k} (n_c - j) \max_{1 \le n \le n_c} |u(t_n) - U_n^0|. \quad (2.6)$$

The product term $\prod_{j=1}^{k}(N - j)$ in (2.6) becomes 0, when the number of iterations reaches L. This shows that the Parareal algorithm terminates with a converged solution for any Δt on any bounded time interval in $N - 1$ steps. Typically you would like to stop the iteration with a converged or sufficiently accurate solution well before $N - 1$ iteration steps, since otherwise there is no speedup in the parallel process.

The stability is also needed for an analysis of the algorithm. Staff and Ronquist [SR05] showed that the method is stable for a system of ordinary differential equations (ODE) if the eigenvalues λ_i are real. If the eigenvalues are imaginary it was numerically proven that Parareal is unstable.

2.3. Parallelization

This section is about the theoretical efficiency of the Parareal algorithm. Parallel efficiency is needed to prove that it is a good idea to parallelize the algorithm in the first place.

To show the parallel efficiency it is assumed that the processors are identical and solving a given timestep of the same length always takes up the same amount of time. The run time for solving a timestep on the coarse level is t_C, for solving on the fine level is t_F. The number of parallel cores used is referred to as N_p. The coarse length of a timestep Δt can also be defined as $\Delta t = \frac{T}{cN_p}$, where $c \in \mathbb{N}^{>0}$ and $n_g = cN_p$.

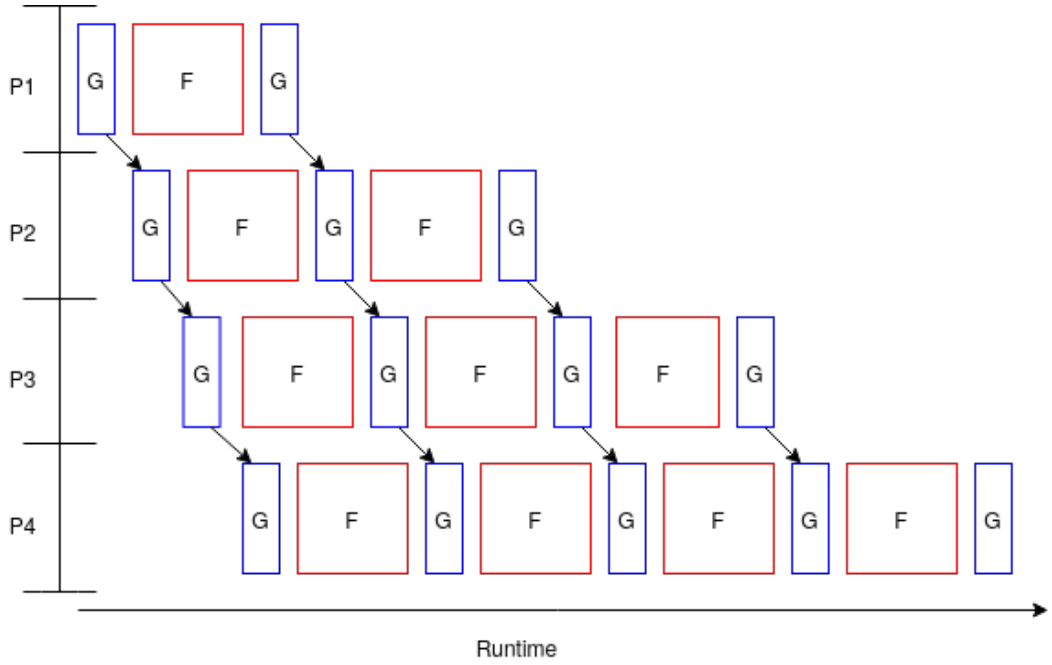


Figure 2.1.: Visualization of the Parareal algorithm with c=1

To show parallel efficiency first an example for $c = 1$ is used, which means that the number of coarse timesteps is the same as the number of parallel cores. An example of a parallel run with $N_p = 4$ and $c = 1$ is shown in figure 2.1. We can see the calculation on the coarse level in blue and marked with a "G". The calculation on the fine level is in red and labeled with an "F". $P1$ to $P4$ are representing the four cores used. An arrow marks communication

between those cores. Each timestep and therefore each core in this example performs the maximum number of iterations for its timestep as explained in chapter 2.2. It was shown in chapter 2.2, that Parareal is reasonable to use if the number of iterations is small. Later in this chapter, it is shown from the parallelization perspective, why a small number of iterations should be the target. To show efficiency first the run time in the worst case is needed. The worst-case with the longest run time is in this example on $P4$. Parareal splits, as explained in chapter 2.2, into a serial part and a parallel part. The serial part on $P4$ is the first coarse calculation plus the waiting time of the communication and the calculation before on the other cores. The time of the communication between the cores is neglected here. This means that the serial time would be four times the run time of solving a coarse timestep in this example. For the parallel part in each iteration, the run time of solving coarse and fine is added. Here four iterations are made. The worst run time, therefore, is eight times the coarse run time plus four times the fine run time. The following is a more general approach to illustrate the calculation of the run time and efficiency.

To calculate the efficiency the serial time to solve the problem referred to as T_1 is needed. To get a similar numerical result calculations on the fine level have to be made for every timestep so that $T_1 = N_p t_F$.

The parallel time to solve the problem is referred to as T_p and consists of a serial part and a parallel part as explained before. The worst case is considered for the calculation. The serial part is the initialization of each timestep. The serial time is $N_p t_G$ because the last processor has to wait for the initialization of every timestep. The parallel time depends on the number of iterations k. In every iteration, the solver does a calculation on the fine level and on the coarse level, which means the parallel time is $k(t_F + t_G)$. The total time is therefore $T_p = N_p t_G + k(t_F + t_G)$.

A unit needed to measure parallel efficiency is the speedup S. Speedup measures the relative performance compared to the serial performance. Ideally, the speedup should be linear with the number of cores used, which means $S = N_p$. In this case the speedup S_1 is

$$S_1 = \frac{T_1}{T_p} = \frac{N_p t_F}{N_p t_G + k(t_F + t_G)} = \frac{1}{\frac{t_G}{t_F} + \frac{k}{N_p}(1 + \frac{t_G}{t_F})}. \tag{2.7}$$

The relationship between a coarse timestep and a fine timestep $\frac{t_G}{t_F}$ is called α. The speedup in this case is linear. If the equation (2.7) is shifted it can be shown that

$$S_1 = \frac{1}{N_p\alpha + k + k\alpha} N_p < N_p,$$

because $N_p\alpha + k + k\alpha > 1$.

So the parallel efficiency is

$$E_1 = \frac{S_1}{N_p} = \frac{\frac{1}{\alpha + \frac{k}{N_p}(1+\alpha)}}{N_p} = \frac{1}{N_p\alpha + k(1+\alpha)}. \tag{2.8}$$

The parallel efficiency depends on the number of iterations k and the relationship α between a coarse timestep and a fine timestep. These two factors have an influence on each other. The smaller α gets the fewer iteration k are needed. Ideally $E = 1$. Because of (2.8) it can be estimated as $E_1 \leq \frac{1}{k}$. This shows that the algorithm is dependent of the number of iterations. The smaller the number of iterations, the larger is E. To make E closer to 1, α should be small. If α is small, then $t_F >> f_G$ and this could mean, that it takes a long time to compute. Therefore, both must be balanced.

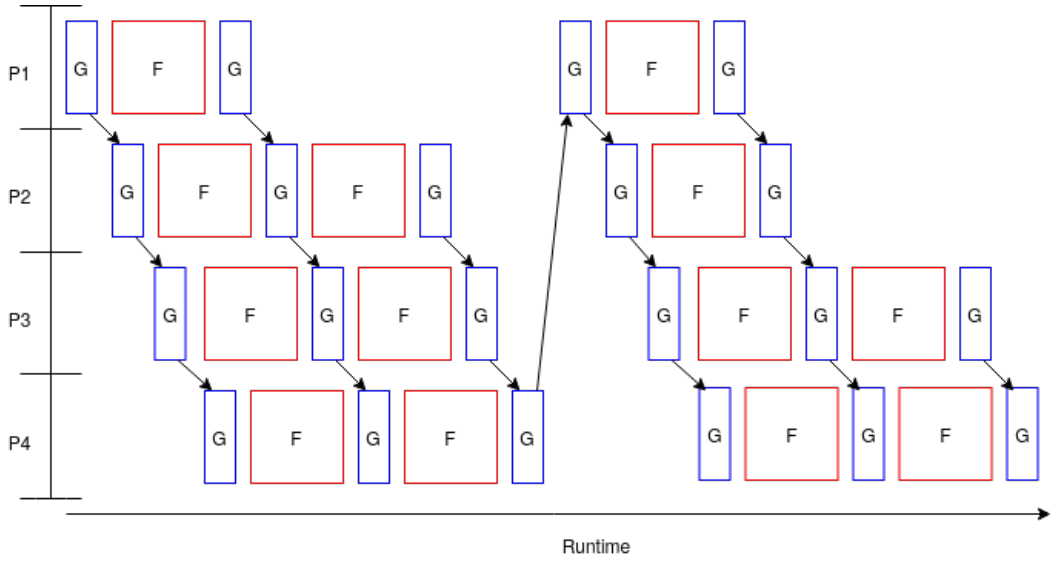


Figure 2.2.: Visualization of the Parareal algorithm with c=2

Most of the time when running Parareal there are more coarse timesteps than parallel cores. In figure 2.1 there are as much timesteps as parallel cores so that $c = 1$. The case which is now being considered has as a prerequisite that $c > 1$. An example for $c = 2$ can be seen in figure 2.2. The structure of the figure is the same as in figure 2.1, but each core has to handle two timesteps. You can see that the timesteps are worked on in blocks. The first block calculates the first four timesteps. After each core has calculated the result for his timestep, the first block is finished and $P4$ sends the results of the first block to $P1$ so $P1$ has a starting value for the second block. In the second block, the last four timesteps are solved as described in figure 2.1. If $c > 1$ you have to consider a change of parameters. The coarse length of a timestep is $\Delta t = \frac{T}{cN_p}$. The serial time to solve the problem is $T_1 = cN_p t_F$. For

the parallel time T_p the two parts, the serial part, and the parallel part have to be considered. The serial part is cN_pt_C because each core has to calculate the starting value for a timestep c times. The parallel part is $\sum_{i=1}^{c} k_i(t_C + t_F)$, where k_i is the number of iterations in each timestep i. The total time is $T_p = cN_pt_C + \sum_{i=1}^{c} k_i(t_C + t_F)$. This means the Speedup S_c is

$$S_c = \frac{1}{\frac{t_C}{t_F} + \frac{\sum_{i=1}^{c} k_i}{cN_p}(\frac{t_C}{t_F} + 1)} = \frac{1}{\alpha + \frac{\sum_{i=1}^{c} k_i}{cN_p}(\alpha + 1)}.$$

The speedup S_c is not linear, because $S_c < N_p$. The parallel efficiency E_c is

$$E_c = \frac{1}{N_p\alpha + \frac{\sum_{i=1}^{c} k_i}{c}(\alpha + 1)} \leq \frac{c}{\sum_{i=1}^{c} k_i}.$$

The efficiency is therefore as estimated before dependent on the number of iterations. The same conclusions as with E_1 can be drawn. Parareal is ideal if $k = 1$ iterations are needed. For good parallel performance, k must be small.

3. A testbed

In this section, the Parareal algorithm is tested. Three different problems are used to test the convergence properties and the efficiency of the algorithm. Parareal is implemented in Julia.

3.1. Preliminary remarks

The system on which the Parareal algorithm is to be parallelized is the supercomputer Jureca [Jül16]. It is located at the Jülich Supercomputing Centre, an institute of the Jülich Research Centre, and consists of a cluster module and a booster module. The cluster is based on a hybrid architecture. A total of 1872 computation nodes and 12 visualization nodes are available. The computation will run on its Cluster module especially on nodes, which have two Intel Xeon E5-2680 v3 Haswell CPUs. These CPU's have Intel Hyperthreading Technology, a technology which makes it possible for each processor core that is physically present, to address two virtual cores and shares the workload between them when possible. The nodes are connected with Mellanox EDR InfiniBand with 100 GiB/s. The topology of the nodes is a non-blocking fat tree.

The chosen tool for parallelization is the Message-Passing Interface (MPI). MPI describes a standard that regulates the exchange of messages during parallel calculations on distributed computer systems [Pad11]. It defines a collection of operations and their semantics, i.e. a programming interface, but no concrete protocol and no implementation. The advantage of MPI is that it is standardized, portable, and widely used. The implementation of the standard is a library of subfunctions that can be used in Fortran, C, and C++. Furthermore, there are freely available MPI implementations.
MPI is based on the Message-Passing Programming Model and builds on the so-called SPMD (Single-Program Multiple-Data) programming model. This means that all cores execute the same program with different data. Each process has its own private memory, which can be distributed or contiguous. One process cannot directly access the data of another process. Each process runs the same program. The data exchange between the processes is done by explicitly sending and receiving messages. The usage of MPI in Julia is done by the package "MPI.jl", which is a wrapper class. This class is based on the C implementation of the MPI standard. This wrapper class is inspired by

mpi4py [DPS05], a package for using MPI in Python. The MPI package in Julia can be added via the package manager. A difference to the use in C and Fortran is the sending of the data packages. In Julia, only arrays can be sent and no data types need to be specified.

For this testbed the first idea was to use the "DifferentialEquations.jl" package to provide the solvers for Parareal [RN17]. The package is a suite for numerically solving differential equations written in Julia. It implements various solvers for the different differential equations for example solvers for discrete equations, ordinary differential equations, split and partitioned ODEs, and a lot more. More information can be found in [RN17]. By using the Differential Equation Package a lot of fast and optimized solvers are available to use without the need to implement them. The package is introduced as easy to use but can be specified a lot for the given problem. The package integrates BLAS on a basic level to speed up the Code. BLAS is short for Basic Linear Algebra Subprograms [Bla+02]. Julia integrates the C version of this library. BLAS provides linear algebra routines such as matrix multiplication, cross-product, and vector addition naming a few. BLAS implementations are often optimized for speed, so using them can bring performance benefits. BLAS is integrated in a lot of other languages as well. If "DifferentialEquations.jl" is used without parallelizing and without adjusting the number of cores used, the runtime of most algorithms is quite fast. If the code is parallelized with MPI for example and the number of BLAS cores is smaller the given fast execution of the code is slower, which can lead to results, which are not predicted. Another problem could be, that if more advanced algorithms should be used it is not clear which algorithms are used in the background to solve this. An example of this is IMEX-Euler method. IMEX-Euler method is used to solve semi-implicit problems. IMEX-Euler method needs to solve an explicit part and an implicit part. The solving of the implicit part is problematic here because Newton's method is always used to solve this problem. Newton's can be used in any case. To calculate Newton's method, a Jacobi Matrix is needed. If the Jacobi matrix is not given, Julia uses auto differentiation to calculate the Jacobi matrix. This first needs a lot of time but also the step size has to be sufficiently small for the algorithm to work. To avoid this, instead of a function a system of linear equations is given as the implicit part, but "DifferentialEquations.jl" does not check for that. By solving a system of linear equations instead of using Newton's algorithm a lot of time could be saved. The tests run confirmed not to use this package, but to self-implement the solvers, because the run time is better, bigger stepsizes can be used and it can be exactly seen, what code is behind the used solvers.

3.2. Lorenz system

The Lorenz system is a set of differential equations first expressed around 1963 by Edward N. Lorenz. It is a simplified mathematical model for atmospheric convection. It is used here as a test problem. The model is a system of three ordinary differential equations:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z, \end{aligned}$$

where $t \in [0, 10]$ and $\sigma = 10$, $\rho = 28$ and $\beta = \frac{8}{3}$ is chosen [GH08]. The initial value for to test this problem is $u_0 = (1, 0, 0)$. To solve this problem explicit Euler method as coarse and fine solver can be used. Explicit Euler method calculates the time step $t_{n+1} = t_n + h$ as $y_{n+1} = y_n + hf(t_n, y_n)$. The solution for the problem if only Euler method without Parareal is used can be seen in figure 3.1. In this plot a chaotic behavior is seen. Depending of the choice of

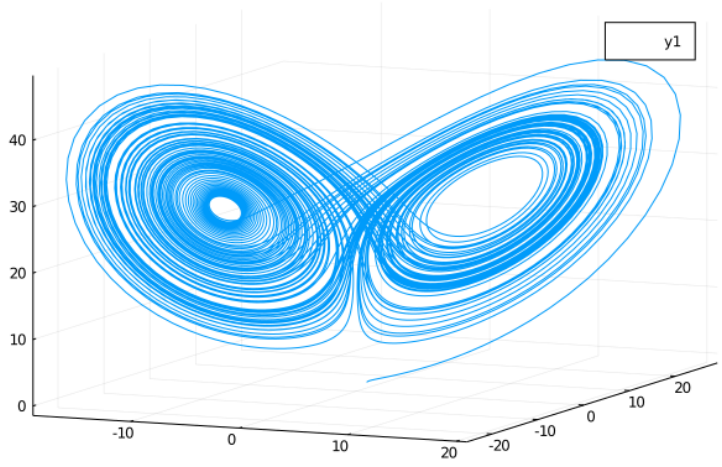
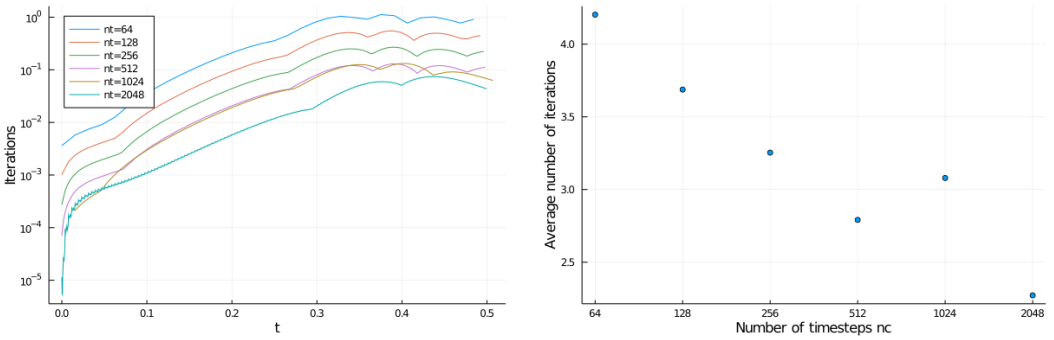


Figure 3.1.: Solution of the Lorenzsystem over time $t = (0, 100)$ with stepsize $\Delta t = 0.01$ and explicit Euler as solver

variables σ, ρ, β different simplified physical models can be emulated.

To check how the Parareal converges the error and the iteration number over time has to be considered. In figure 3.2 in (a) the error and in (b) the number of iterations over time and with different sizes of time steps and so different n_c are shown. In 3.2a it can be seen that the smaller the time step is chosen, the smaller the error is. For every time stepsize first, the error becomes bigger



(a) Error against time for different n_c (b) Iteration against time for different n_c

Figure 3.2.: Error and Iteration of the Lorenz system over time solved with Euler as coarse and fine solver with $n_f = 10$ for different n_c run on 8 cores

but after approximately $t = 0.3$ remains at about the same level. In 3.2b it can be seen that the smaller the time step the fewer iterations are used as a maximum.

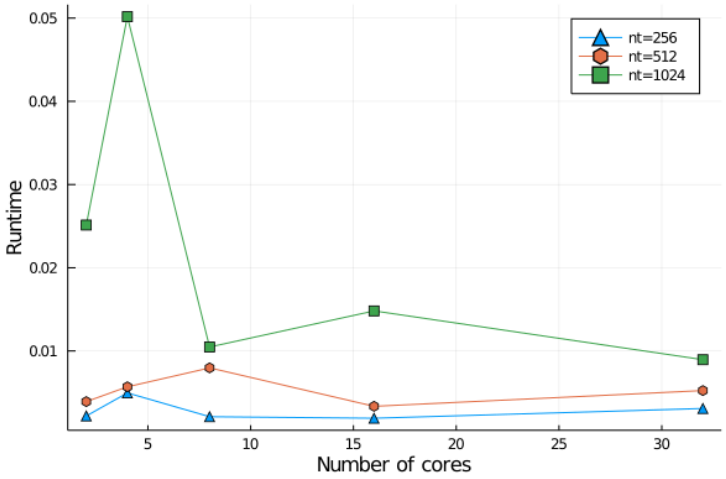


Figure 3.3.: Runtime of the Lorenz equation with different number of cores and number of time steps n_t

In Figure 3.3 the run-time tests made for the Lorenz equation are shown. The tests are made on Jureca with Julia 1.3.1 and Intel MPI. On the x-axis, the number of cores and on the y-axis, the runtime of the code can be seen. The tests are made for different size of coarse time steps n_c and the fine time steps is set as $n_f = 10$. The typical behavior that with more cores the runtime becomes smaller and with more coarse time steps n_c the runtime becomes bigger is not seen here. The behavior is really chaotic. A problem could be that as seen in Figure 3.2b the number of iterations for 8 cores is not small.

Also, the timings are fairly small. Therefore the calculations are run multiple times and the average runtime of them is taken in this plot.

3.3. Heat equation in 1D with periodic boundary conditions

Another test problem is the heat equation in 1D with periodic boundary conditions on a spatial finite differences grid. The heat equation is a partial differential equation. The equation to solve is

$$\frac{\partial u(x,t)}{\partial t} = - \triangle u(x,t) + f(x,t) \quad \text{on} \quad \Omega \tag{3.1}$$

$$\tag{3.2}$$

with periodic boundary conditions, where $\Omega = (0,1)$, $\triangle\rho = \mathrm{div}(\mathrm{grad}(\rho))$ is the Laplace operator.
The Laplace operator in $1D$ can be discretized as a matrix A as

$$\triangle = A = \frac{1}{h^2} \begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & 1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{pmatrix}$$

where h is the spatial step size. Due to the periodic boundary conditions, the matrix changes here to

$$\triangle = A = \frac{1}{h^2} \begin{pmatrix} -2 & 1 & 0 & \cdots & 1 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & 1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 1 & \cdots & 0 & 1 & -2 \end{pmatrix}. \tag{3.3}$$

The right-handside of the equation 3.2 f is chosen as

$$f(x,t) = ((k\pi)^2 \cos(t) - \sin(t)) \sin(k\pi x),$$

so that the exact solution is

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

The exact solution is needed to calculate the error.

IMEX-Euler method is used as a fine and coarse solver. It is here used because the heat equation with periodic boundary conditions is semi-implicit and therefore needs a specific solver to solve the implicit, stiff part and the explicit also called non-stiff part of the equation. IMEX-Euler is an integrator that can solve such an equation. IMEX-Euler is defined in equation (3.4), where $q(u_{i+1}, t_{i+1})$ is the implicit part and $p(u_i, t_i)$ is the explicit part. In this specific case $q(u_{i+1}, t_{i+1}) = Au_{i+1}$, because of the Laplace operator and $p(u_i, t_i) = f(x, t)$. The equation can be rewritten to equation (3.5). This means that instead of solving an implicit problem, the problem can now be seen as solving an equation system.

$$\begin{aligned} u_{i+1} &= u_i + \Delta t q(u_{i+1}, t_{i+1}) + \Delta t p(u_i, t_i) && (3.4) \\ \Leftrightarrow u_{i+1} &= u_i + \Delta t A u_{i+1} + \Delta t f(x, t_i) \\ \Leftrightarrow (I - \Delta t A) u_{i+1} &= u_i + \Delta t f(x, t_i) && (3.5) \end{aligned}$$

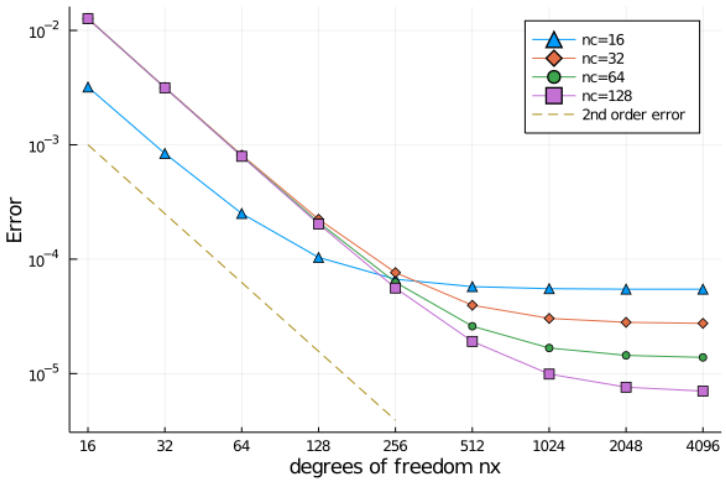


Figure 3.4.: Error of the heat equation over n_x and IMEX-Euler method as coarse and fine solver for different n_c and $n_f = 10$

In figure 3.4 on the x-axis the number of the spatial grid points n_x also known as degrees of freedom and on the y-axis the error at $t = 0.1$ is plotted for different numbers of time intervals n_c. This kind of plot is used to show convergence and find the sweet spot. The sweet spot is the point at which the error starts staying the same. It is important to know for simulations at what specifications a small error in comparison to the size of the problem is reached. It applies that the bigger n_x is the smaller the error is for $n_x > 256$. The error till the sweet spot is reached goes down like $\mathcal{O}(n^2)$. By looking at the different number of time steps n_c it can be seen, that the error is halved if the number of time steps is doubled.

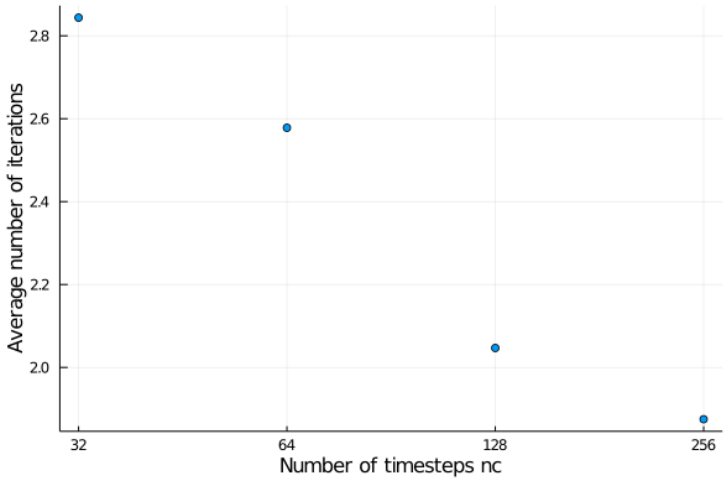


Figure 3.5.: Number of iterations of the heat equation with periodic boundry conditions over time and IMEX-Euler method as coarse and fine solver with $n_f = 10$ for different n_c and $n_x = 256$

In figure 3.5 for $n_x = 256$ the average number of iterations for different numbers of time steps n_c is applied. The more time steps are used, the smaller is the average number of iterations. In general, the number of iterations is fairly small. This is good, because like explained in chapter 2.3 the smaller the number of iterations is, the better the parallel efficiency can be.

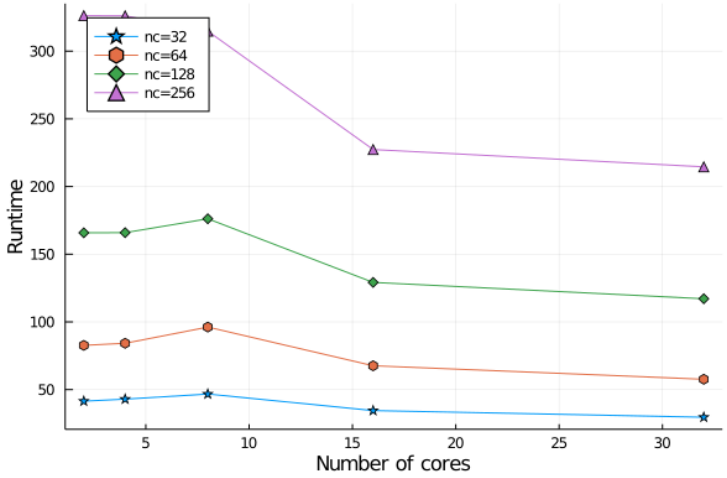


Figure 3.6.: Runtime of the heat equation for $n_x = 256$ with different number of cores and number of time steps n_c and $n_f = 10$

Figure 3.6 shows the runtime of the example for different n_c and number of cores with $n_x = 256$ and $n_f = 10$. The bigger n_c is the longer is the runtime. Between the runtimes is a factor 2. Till 8 cores the runtime gets longer, but

after that, it gets smaller.

3.4. Allen-Cahn Equation

Another problem interesting to look at is the two-dimensional Allen-Cahn equation [AC79]. The equation can be used to model various problems but has become the basic model equation for the diffuse interface approach developed to study phase transitions and interfacial dynamics in materials science. The equation can be displayed as

$$\frac{\partial u}{\partial t} = \triangle u + \frac{1}{\varepsilon^2}(-2u(1-u)(1-2u)) = 0,$$

with periodic boundary conditions on $[0,1]^2 \times [0,T], T > 0$, $u(x,0) = u_0(x), x \in [0,1]^2$, $\triangle$ is the Laplace operator introduced in equation (3.3) and ϵ is a scaling parameter. This problem is also known as a phase field problem. Phase-field models solve interfacial problems, which are characterized by a field, which takes two distinct values, here, for instance, 1 and 0, in each of the phases, with a smooth change between both values. This zone between the values is determined by a parameter, here ϵ also called the diffuse interface width parameter. If $\epsilon \to 0$, the solution of the problem should behave like a piecewise function.
As a initial condition

$$u_0 = \tanh\left(\frac{R_0 - x^2 + y^2}{\sqrt{2}\varepsilon}\right).$$

is chosen. The initial condition describes a circle with initial radius R_0. Over time this circle should shrink. Its radius can be calculated as $r(t) = \sqrt{R_0^2 - 2t}$. Calculation of the error err_t for every time step t is here done with the help of the radius. To get the calculated radius the volume is used. For a exact solution $V_{exact,t} = \pi * max(R_0 - 2t, 0)$ and $r_{exact,t} = \sqrt{\frac{V_{calculated,t}}{\pi}}$. To get the calculated error every time step checks, how many elements e_t of the spatial grid are $u > 0.5$. The volume can therefore be calculated by $V_{calculated,t} = e_t h^2$ and the radius by $r_{calculated,t} = \sqrt{\frac{V_{calculated,t}}{\pi}}$. The error $err_t = |r_{calculated,t} - r_{exact,t}|$ helps as an indicator whether the calculation with Parareal produces reasonable results. For calculation $h < \epsilon$ to catch the transition between the phases in space and $\Delta t < \epsilon^2$ get the transition of phases in time. Here $\epsilon = 0.04$ is chosen. As a fine and coarse solver, IMEX-Euler is used since the Allen-Cahn equation is a semi-implicit equation. IMEX-Euler was introduced in equation (3.4) and here be calculated as

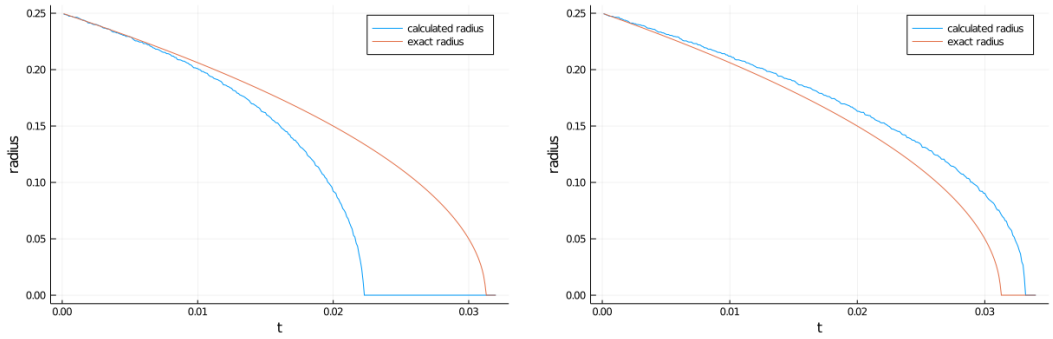
$$(I - \Delta t A)u_{i+1} = u_i + \Delta t f(u_i, t_i).$$

Zhang and Du [ZD09] showed that normally the error for IMEX-Euler should increase with $\Delta t \sim \epsilon^3$, but is increasing with $\Delta t \sim \epsilon^2$ for this specific case. Zhang and Du [ZD09] propose a stabilized Euler, which can reach $\Delta t \sim \epsilon^3$ by introducing a factor k for Allen-Cahn so that the equation is modified to

$$\begin{aligned}(I - \Delta t A)u_{i+1} + \Delta t k u_{i+1} &= u_i + \Delta t f(u_i, t_i) + \Delta t k u_i \\ \Leftrightarrow (kI + \frac{1}{\Delta t} I - A)u_{i+1} &= (k + \frac{1}{\Delta t})u_i + f(u_i, t_i).\end{aligned}$$

Here $k = \frac{2}{\epsilon^2}$ is chosen.

In Figure 3.7 the two approaches are tested on solving the Allen-Cahn equation. This test is done without Parareal and the stepsize $\Delta t = 0.001$ was chosen.



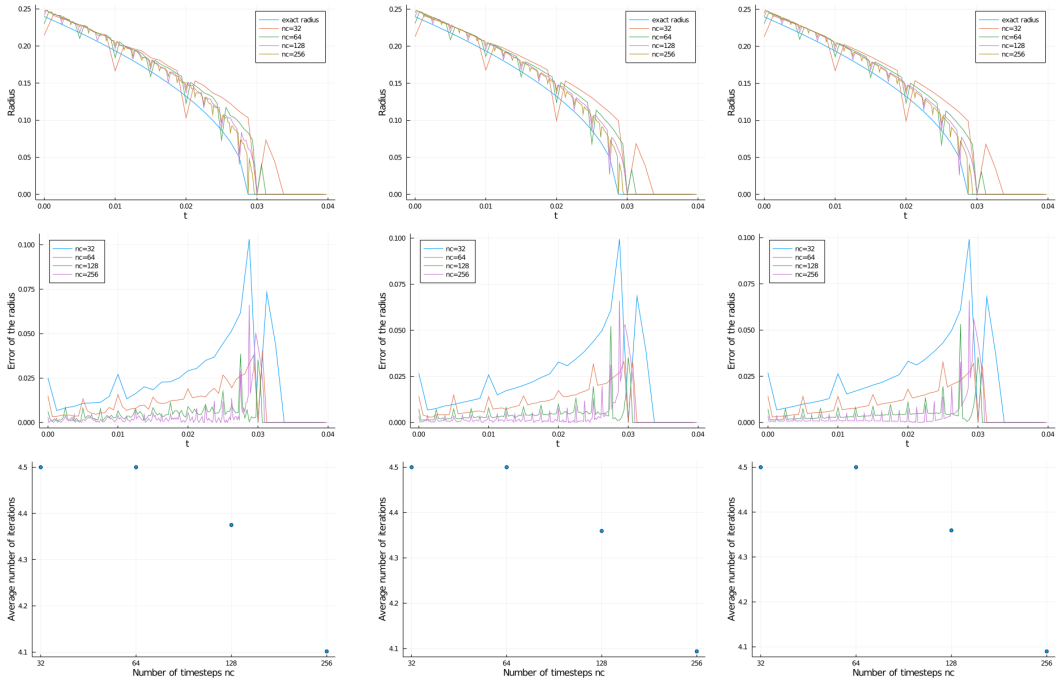
(a) Original IMEX-Euler (b) Stabilized IMEX-Euler

Figure 3.7.: Calculated and exact Radius of the Allen-Cahn equation with IMEX-Euler and stabilized IMEX-Euler

In Figure 3.7 the calculated radius of the original IMEX-Euler is near to the exact radius but gets distant fast. In Figure 3.7b the calculated radius of the stabilized IMEX-Euler is overall closer to the solution. In the following stabilized IMEX-Euler is used.

Figure 3.8 shows the calculations with Parareal. In the first row, the radius is plotted against the time, in the second row the error of the radius is plotted against the time and in the third row the number of iterations is plotted against the time, with different n_c. In the first column 3.8a is $n_x = 64$, the second 3.8b is $n_x = 128$ and the third 3.8c is $n_x = 256$. Generally, the smaller n_c is, the closer the calculated radius is to the exact radius, the smaller the error is and the fewer iterations are used. That applies to every n_x. The bigger n_x gets the smoother is the radius as seen in the first row.

Figure 3.9 shows the runtime of the example for different n_c and number of cores with $n_x = 128$ and $n_f = 10$. The bigger n_c is the longer is the runtime. Between the runtimes is a factor of 2. Till 8 cores the runtime gets longer, but after that, it gets smaller.



(a) $n_x = 64$ (b) $n_x = 128$ (c) $n_x = 256$

Figure 3.8.: Calculations with Parareal of the radius, error of the radius and iteration number against time with stabilized IMEX-Euler for different n_c, different n_x and $n_f = 10$

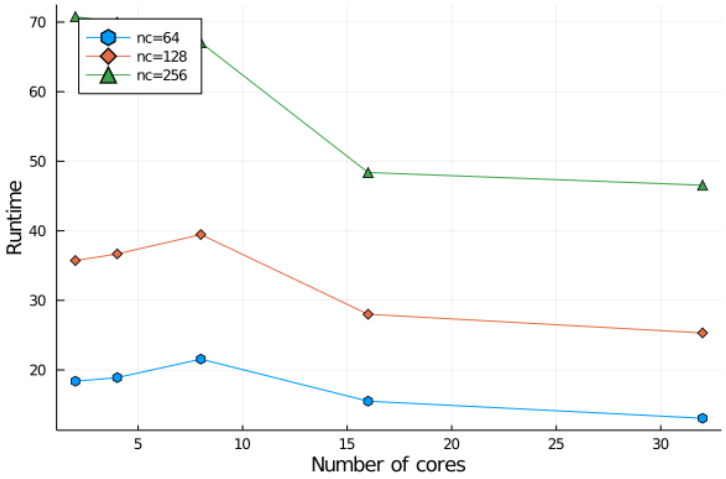


Figure 3.9.: Runtime of the Allen-Cahn equation for $n_x = 128$ with different number of cores and number of time steps n_t and

4. Containerization

Getting an application to run on a different host system than the own computer can be challenging and time-consuming. Julia is a new programming language and not installed on most host systems by default. Creating the environment to run the applications presented in the last chapters can be challenging. In this chapter, containerization is introduced as a tool to make this easier. At first, virtualization in general is introduced and different virtualization strategies are presented and compared. Then containerization in general and Singularity as the tool chosen here is explained in more depth. An example of a container for the application presented in chapter 3, its specifications are given and the advantages of using a Singularity container in this specific case are explained. This chapter is based on [KSB17] and [Inc].

4.1. Introduction to virtualization

Virtualization refers to the replication of a hardware or software object by a similar object of the same type with the help of an abstraction layer. This allows the creation of virtual, non-physical, devices or services such as emulated hardware, operating systems, data storage or network resources. Virtualization can basically be divided into three different main types, desktop virtualization, hardware virtualization and operating-system-level virtualization, also known as containerization. These types can be implemented with different tools.

Desktop virtualization is the concept of isolating the logical instance of an operating system from the client that accesses it. The virtualization of the desktop can be made locally or remotely. A use case is remote desktop virtualization which is integrated into a client/server computing environment. Application execution takes place on a remote operating system that communicates with the local client device over a network using a remote display protocol through which the user can interact with an application. This type of virtualization does not meet the requirement of our application, because a desktop is not needed.

Hardware virtualization can be seen as the creation of a virtual machine that acts like a real computer with a kernel and an operating system. Software executed on a virtual machine like this is separated from the underlying hardware resources. Virtual machines always distinguish between the host machine, which is used by the virtualization and the guest machine, which is the vir-

tual machine. There are different types of virtual machines depending on the use case. The most common use case of virtual machines is to run additional isolated operating systems to the one that already runs on the host machine. Virtual machines can be used to run many different applications on the same machine, independent of the application's requirements. The virtual machine is completely isolated and can have an entirely different environment, including operating system, hardware dependencies, software dependencies and libraries. Virtual machines are extremely portable but come with large computational overhead, due to the need of emulating an operating system and a kernel. Another disadvatage of virtual machiens is it that due to the isolation there can be additional complexities when trying to access host specific resources. A virtual machine can be saved in a single image file. A widely known option for working with virtual machines is VirtualBox [Ora]. VirtualBox is a cross-platform virtualization application, which means that the virtual machine is practically only limited by the hosts disk space and memory but not limited in terms of the hosts operation system or hardware configuration as well as the virtual machines operation system or configurations. It is also possible to have a number of virtual machines running at the same time. It can run anywhere from desktop machines to data center deployments and even Cloud environments.

Operating system virtualization is an operating system paradigm in which the kernel allows the existence of multiple isolated user-space instances. A modern computer operating system usually segregates virtual memory into kernel space and user space to provide memory and hardware protection. Kernel space is strictly reserved for running an operating system kernel. User space is the memory area where application software runs and in this case the virtualization as well.

Such instances can for example be virtual environments or containers. Virtual environments have the goal to create an isolated environment for a specific project. The most used one is called Pip [PyP], a package manager purely for Python packages. Another often used environment management system is called Conda [20]. Using Conda it is possible to install packages into environments written in any language which even makes it suitable for non-Python projects, although it is mostly used to create Python environments. All environments will always share all global Python packages and other software, but will not fully isolate the host machine from the environment. Since the application runs in Julia and on a HPC system a virtual environment is not the choice of virtualization in this thesis.

A container on the other hand is an isolated process that shares the same kernel as the host operating system, as well as the libraries and other files needed for the execution of the program running inside of the container. Often the container runs with a network interface so that the container can be exposed to the world in the same way as a virtual machine. Typically, con-

tainers are designed to run a single application and are specified only for this application. Containers can be generated by specific definition files and are saved in just one image file.

There are different container solutions such as Docker [Mer14], CharlieCloud [Sec], Shifter [NER] and Singularity [Inc]. The most popular container solution is Docker. Docker is the industry standard for micro-service virtualization, but it initially did not meet the requirements for a scientific working environment. Docker uses Linux namespaces in combination with cgroups. Cgroups limit the resources the container can use, which means one container can only see their designated resources. To execute and build these containers the user had to be a root user. Therefore, when working in an HPC environment Docker does not meet the security requirements. On a shared system, such as a HPC system, if any user is able to run any code it would be strictly against all security guidelines. With the new Docker update 20.10 Docker container are able to run in an so called "rootless" mode due to a new cgroups version. CharlieCloud [Sec] is an open-source user defined softwarestack. The goal is to make Docker container runable on HPC systems, which means to make them runable without root privileges. The main issue with this approach is the compatibility. The software makes use of kernel namespaces that are not deemed stable by multiple prominent distributions of Linux such as RedHat and therefore are not able to run on these operating systems. Shifter is also a service that works with Docker containers and modifies the container to make them runable without root privileges with the goal to have a Docker container run on a HPC system. Shifter uses a different strategie than CharlieCloud. The disadvantage of Shifter is that it requires a complex administrative setup. Singularity is container solution that specializes to the needs of scientific workflows and HPC. The architecture of Singularity allows users to safely run containers on a HPC cluster system without the possibility of root privilege escalation. Singularity runs on the target system Jureca.

Generally, containers are the more portable virtualization approach compared to a virtual machine which is due to the fact that images of virtual machines oftentimes a lot bigger in size, because the kernel has to be emulated as well. In the HPC domain, containers are typically preferred over virtual machines because of the overhead virtual machines come with both regarding disk space and performance. Therefore in this thesis, a container is chosen as the tool to virtualize the application with the container solution Singularity.

4.2. Singularity

Singularity [Inc] as described befor is a container solution to create the necessities for scientific application driven workloads. The goals of Singularity are:

- **Mobility of computing** - With a singularity container a workflow can be executed on different hosts, Linux operating systems, and cloud providers while containing the entire software stack.

- **Reproducibility** - Once a workflow is working the container can be saved so that the user can later come back to it and reproduce the results. This is important in the scientific community as well so that when publishing a paper for example an author can use a Singularity container for others to verify the results.

- **User freedom** - The user can install applications, versions, and dependencies for their projects without being dependent on a host system, that could have a different environment. The container can be copied to most host systems and the user can run their project inside the container without having to look if packages are installed or the right version is available.

- **Support on existing traditional HPC resources** - The goal is to support HPC environments. Singularity supports technologies such as Infiniband and at the same time integrating with resource managers like SLURM.

Singularity containers are made up of a single file also called the Singularity image file (`.sif`). A `.sif` file can be generated in multiple ways. The easiest way is to just pull a container from an image library such as the official Singularity library or the Docker library [Mer14]. The more common way is to build an image file from a build file. A build file consists of different parts. Not all of these parts have to be included in a build file. Following the parts will shortly be introduced:

- **Header** - The header configures the base image used for the container which will be an operating system in most situations, but could as well be a simple executable or library. Two keywords are essential for the header: "Bootstrap" determines the bootstrap agent that will be used to create the base operating system, "From" is the named container that is used as a base image. More specific information can be given as well. A base image can also be a Docker image [Mer14]. A header has always to be included.

- **Help** - The text in this section can be displayed when running the `singularity run-help <container image>` command.
- **Setup** - The setup section is executed first. It is executed on the host system outside of the container after the base OS has been installed. Commands in this section can alter the host. Therefore it is not recommended to use this section.
- **Files** - In this section, files from the host system can be copied into the Singularity file system. The path of the file to copy has to be valid on the host as well as the path to where to copy it inside the container. This provides more safety than the setup section because files can only be copied but not altered.
- **Labels** - The labels section can be used to add metadata. The metadata will be added to the file "/.singularity.d/labels.json" within the container. The metadata has to be given in a name-value pair. An example could be name "Author" with a value "name". The metadata can be displayed by running `singularity inspect <container image>`. Some labels are also automatically inserted from the build process.
- **Environment** - In the environment section environment variables can be defined. These variables will be set at run time, not at build time. For the environment variables to be available during build the variables need to be defined in the post section as well. The script is also setting an environment variable at build time.
- **Post** - In this section, files can be downloaded, software and library installed, directories created, configuration files written and environment variables can be set during the build time. The commands that can be used here are dependent on the base image.
- **Runscript** - The commands written in the run script section are written into a file within the container that is executed when the container image is run by `singularity run <container image>`. By running this command arguments can be passed to the runscript as well.
- **Startscript** - Like the runscript section, the contents of the startscript section are written to a file within the container at build time. This file is executed when the command `singularity instance start <container image> <instance to start>` is issued.

An image file can be built in two different ways. The native way is by executing "`sudo singularity build <image file> <definition file>`". You need root privileges to build an image file. The second way is to build a so-called "sandbox directory". A sandbox directory is a writable directory that can

be transformed into an image file later. The advantage is that the resulting directory operates just like a container in a `.sif` file, you only need root privileges to transform the directory into an image file, but not to build the sandbox container and you can make changes inside the sandbox directory. A sandbox directory can be build by executing "`singularity build -sandbox <sandbox directory> <definition file>`".

Singularity can be run in the foreground and the background. Running in the foreground there are different ways to do so. The shell command allows you to spawn a shell inside your container. This can be executed by "`singularity shell <image file>`". By executing this the prompt in the shell should change. In this shell commands can be executed. The image can be interacted with like a virtual machine. Another way is to execute a specific command inside the container by "`singularity exec <image file> <command>`". Files on the host are reachable from within the container for both commands. The third option is executing the run scripts inside the container by "`singularity run <image file>`". Running a file in the background is also called detached mode. Detached containers are persistent and can be started multiple times simultaneously, while each of these containers is called a container instance. Instances are fully isolated from each other and can be addressed individually. Running containers like this in the background is especially suited for services like web servers that multiple clients are supposed to use and will not be used in this thesis.

4.3. Example: Singularity container for Julia and OpenMPI

In this case, a Singularity container is needed in which a Julia code can be executed and the Julia MPI package can be used. In the following, some subsections of the definition file are introduced. The whole definition file can be found in the appendix. This specific definition file is for a container which runs on Julia 1.3.1 and Open MPI 4.0.2 [Gab+04]. The introduced approach here for working with MPI in a Singularity Container is the hybrid model. The Singularity container in this approach works with MPI from inside and outside the container. The two MPI containers have to be compatible. By calling `mpirun` to execute the container the MPI process outside of the container will work together with the MPI inside the container to run the code. The advantage of the hybrid approach is that it is easy to use because it is similar to the way of natively running the code. The disadvantage of this approach is that the MPI versions of the host and the container must be compatible and that the MPI implementation in the container must be configured for optimal use of the hardware if performance is critical. The other approach is called the bind approach. Here the container does not include an MPI implementation, which

means that Singularity needs to bind the MPI version available on the host into the container. The advantage here is that the container image is smaller and there is no need to configure the MPI implementation. For this approach to work, the user has to know where MPI is installed on the host system and that binding is possible. The reason the approach is not used here has to do with the Julia MPI package. During the building of the container, the MPI package in Julia is built as explained before. Without a MPI implementation in the container but only a link where MPI will be, the package cannot be built. Therefore this approach can not be used here.

Listing 4.1: Header of the definition file

```
BootStrap: docker
From: bitnami/minideb:jessie
```

In listing 4.1 the header of the definition file is shown. In this case from docker hub, a minimalist Debian-based image built specifically to be used as a base image called "jessie" is chosen. It is chosen, because this base image is small, but has many packages available for easy integration. It is based on "glibc" for wide compatibility and has apt for access to a large number of packages. Glibc stands for GNU C Library project and provides the core libraries for GNU/Linux systems.

Listing 4.2: Environment of the definition file of the Julia Open MPI container

```
%environment
    # Set env variables so we can compile our
       application for Julia
    export JULIA_DEPOT_PATH=$PWD/containerhome/.julia
       :/user/.julia
    export PATH=/opt/julia/bin:$PATH
    export HOME=/user

    # Set env variables so we can compile our
       application for OpenMPI
    export OMPI_DIR=/opt/ompi
    export SINGULARITY_OMPI_DIR=$OMPI_DIR
    export SINGULARITYENV_APPEND_PATH=$OMPI_DIR/bin
    export SINGULAIRTYENV_APPEND_LD_LIBRARY_PATH=
       $OMPI_DIR/lib

    export PATH=$OMPI_DIR/bin:$PATH
    export LD_LIBRARY_PATH=$OMPI_DIR/lib:
       $LD_LIBRARY_PATH
    export MANPATH=$OMPI_DIR/share/man:$MANPATH
```

In listing 4.2 the environment variables are set for Julia and Open MPI. The environment variables set here are used during executing the container and not during building the container.

After that in the ”post”-section first the needed packages are installed, Julia is downloaded and then unzipped. Listing 4.3 shows how first the environment variables are exported than needed packages for the Julia installation are installed. After that the needed Julia version (here 1.3.1) is downloaded from the official Julia source, unzipped and moved to the target destination.

Listing 4.3: Installation of Julia into the container

```
%post
    #Configurations for Julia

    mkdir -p /user
    export HOME=/user
    export JULIA_DEPOT_PATH=/user/.julia
    export PATH=/opt/julia/bin:$PATH

    JULIA_MAJOR=1.3
    JULIA_MINOR=.1

    install_packages curl tar gzip openssh-client git
        ca-certificates

    curl -k https://julialang-s3.julialang.org/bin/
       linux/x64/$JULIA_MAJOR/julia-
       $JULIA_MAJOR$JULIA_MINOR-linux-x86_64.tar.gz >
       julia.tar.gz
    mkdir /opt/julia
    tar xzf julia.tar.gz -C /opt/julia
    rm julia.tar.gz
    mv /opt/julia/$(cd /opt/julia; echo julia-*)/*
       opt/julia/
```

Also in the post-section, the installation of Open MPI 4.0.2 [Gab+04] is done. First, the required packages are installed, then the environment variables needed for the installation are set. Next, Open MPI 4.0.1 is downloaded from the source. After that, the package is unzipped, compiled, configured, and installed.

In listing 4.4 it is shown how Open MPI is integrated with Julia. This is done with the help of the MPI-package in Julia [Lan]. First, the MPI Julia package is downloaded with the help of the Julia package manager. Here the

Listing 4.4: Installation of Open MPI into the Container

```
    apt-get update && apt-get install -y wget git
       bash gcc gfortran g++ make file

    export OMPI_DIR=/opt/ompi
4   export OMPI_VERSION=4.0.2
    export OMPI_URL="https://download.open-mpi.org/
       release/open-mpi/v4.0/openmpi-$OMPI_VERSION.tar
       .gz"
    mkdir -p /tmp/ompi
    mkdir -p /opt

9   cd /tmp/ompi
    wget -O openmpi-$OMPI_VERSION.tar.gz https://
       download.open-mpi.org/release/open-mpi/v4.0/
       openmpi-4.0.2.tar.gz
    tar xfvz openmpi-$OMPI_VERSION.tar.gz
    # Compile and install
    cd /tmp/ompi/openmpi-$OMPI_VERSION && ./configure
        --prefix=$OMPI_DIR && make install
```

specific version 0.15.1 is downloaded. In the next step, the Package is built with the given installed MPI version.

Required packages for running the Julia code can be added with the package manager like shown in listing 4.5. In this example the package "ArgParse" is added in version 1.1.0. ArgParse is a package for managing input with flags over the terminal.

The container can be executed by calling `mpirun -np <number of procs> singularity exec <container image> julia <code to run>`.

Listing 4.5: Binding MPI into Julia

```
    # Combine MPI and Julia here

3   export CC=`which mpicc`
    export FC=`which mpif90`
    julia -e 'using Pkg; Pkg.add(PackageSpec(name="
       MPI", version="0.15.1"));'
    julia --project -e 'ENV["JULIA_MPI_BINARY"]="
       system"; using Pkg; Pkg.build("MPI"; verbose=
       true)'
```

Listing 4.6: Building of Julia packages

```
    julia -e 'using Pkg; Pkg.add(PackageSpec(name="
       ArgParse", version="1.1.0"));'
```

5. Testing with Singularity

In this chapter the impact of the container on the runtime of the application is tested. In the following, timing tests for the test cases proposed in chapter 3 are presented. Then the portability to other host systems is tested. After, that working with another MPI version than Open MPI is presented.

5.1. Singularity on Jureca

Jureca is a modular supercomputer which is located at Jülich Supercomputing Centre[Jül16]. The supercomputer Jureca was introduced in 3.1. Used for computation with and without a container Julia 1.3.1, Open MPI 4.0.1, JuliaMPI v0.15.1 are used. The container is built with Singularity version 3.5.2 and executed with Singularity version 3.6.4-1.el7. For all runs, the number of BLAS threads used is 1. BLAS threads were introduced in chapter 3.1. Runtimes can differ from chapter 3 because another MPI version is used and the number of BLAS threads is set. This topic is explained in chapter 3.1. The definition file of the container used here can be seen in the appendix.

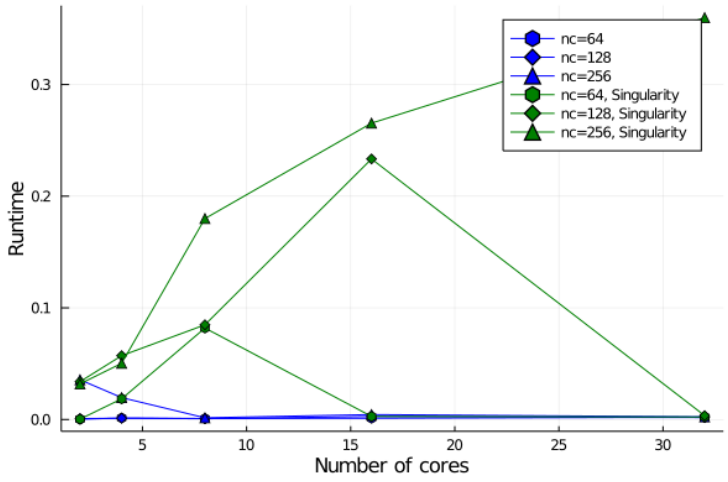


Figure 5.1.: Run time in seconds of the Lorenz equation run on Jureca with Open MPI comparing a native run and a run with a container

Figure 5.1 shows the run time of the Lorenz example, which is introduced in chapter 3.2. It shows the run time in seconds plotted against the number

of cores used on one node. The use of Singularity is compared for a different number of time steps n_c. The run time in this example is fairly small, so all runs were performed ten times and the average run time was taken here. It can be seen that the run time with the container for every number of time steps is bigger than the native run without the container. This means the overhead here is up to 15625%. This could be due to the small run time.

Figure 5.2 shows the run time of the heat equation. The heat equation is introduced in chapter 3.3. Figure 5.2 shows the run time in seconds plotted against the number of cores used on one node. The use of singularity is compared for a different number of time steps n_c for $n_x = 256$, $n_f = 10$ and IMEX Euler as integrator. The run time in this example is smaller than in figure 3.6. In this plot it can be seen that the more time steps are used, the more run time is needed to calculate. Until 16 cores the run time goes down for every number of time steps. For 32 cores the run time is worse than for 16 cores. For 32 cores hyperthreading is used. This can lead to a higher runtime. By using a container in this example the run time is slightly bigger than without. This leads to an overhead up to 31%. Especially until 16 cores the run time is nearly the same. For 32 cores there is a bigger difference between the usage of Singularity and the native run, which could be explained by hyperthreading as well.

Figure 5.3 shows the run time of the Allen-Cahn equation, which is introduced in chapter 3.4. It shows the run time in seconds plotted against the number of cores used on one node. The use of singularity is compared for a different number of timesteps n_c for $n_x = 128$, $n_f = 10$ and IMEX Euler as integrator. Compared to figure 3.9 the run time is a little bit better for the native run. In this plot it can be seen that the more time steps are used, the

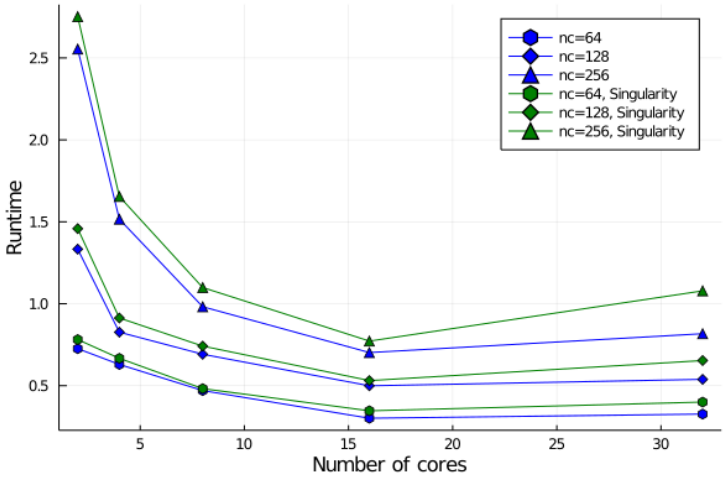


Figure 5.2.: Run time in seconds of the heat equation run on Jureca with Open MPI comparing a native run and a run with a container

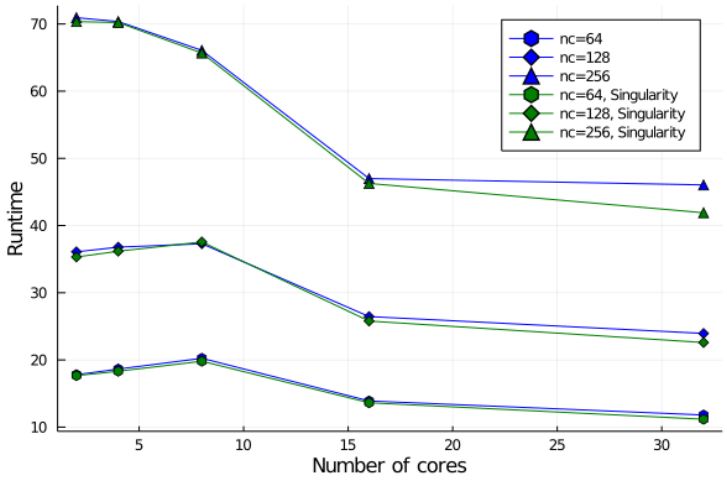


Figure 5.3.: Run time in seconds of the Allen-Cahn equation run on Jureca with Open MPI comparing a native run and a run with a container

more run time is needed to calculate. The minimal run time for every number of time steps shows when 16 cores are used on one core. For 32 cores the run time is nearly the same as 16 cores. For 32 cores hyperthreading is used, but the runtime does not get bigger. This could be because the number of iterations in total is smaller. By using a container in this example the run time is nearly the same until 16 cores. For 32 cores there is a bigger difference, which can be explained by hyperthreading as well. This leads to an overhead up to 9%.

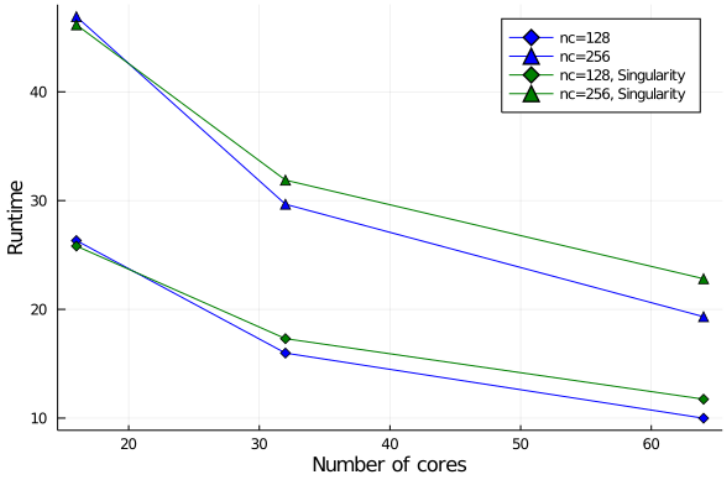


Figure 5.4.: Run time in seconds of the Allen-Cahn equation run on Jureca on multiple nodes with Open MPI comparing a native run and a run with a container

Until now only one node was used and the overhead was small. That means intra-node communication was tested. Another test case would be inter-node communication, which means the communication between the nodes. In figure 5.4 the use of Singularity is compared for different number of time steps n_c for $n_x = 128$, $n_f = 10$ and IMEX Euler as integrator. The more nodes are used, the smaller the run time is. Nevertheless it can additionally be observed that the more nodes are used the bigger the overhead gets for using a container. This leads to an overhead up to 16%. This could be explained because the container tested here works with the hybrid approach, which was introduced in chapter 4.3. The hybrid approach works inside the container with the downloaded Open MPI. The configuration of Open MPI inside the container is most likely not the same as on Jureca, which can lead to a slower run time with the singularity container.

5.2. Portability to other hosts

One of the advantages of working with singularity containers is their high portability. Tests on Jureca showed that the container can be run without a severe overhead. Therefore the application should be able to run on other hosts as well.

Jusuf [JSC] is the an other supercomputer available at Jülich Supercomputing Centre. One node on which the calculation could be done has two AMD EPYC 7742 with 64 cores each. The nodes are connected with Mellanox InfiniBand HDR 100 and the topology of the nodes is a full fat-tree network. The current software stage on Jusuf has Open MPI not installed. A native run of the application on Jusuf is not possible, because the Julia version available in the current software stage is not compatible with the code and the Julia version needed, which is available on the software stage "Devel-2019" is not able to execute the application due to issues with MPI.

The software stage 2020 has an Open MPI installation. Open MPI 4.1.0rc1 is used in this case. A container was build with this version, but due to other errors the container could not run. The same problem occurs with the Intel MPI container. A serial run with Julia is possible. This shows, that this container is dependent on the outside MPI and the container with MPI can only be run successfully if it is compatible with the MPI version outside of the container. The software stage "Devel-2019" has an OpenMPI 4.0.2 installation. This installation was used for the tests on Jureca and is here used for the tests on Jusuf. The same container is used for run time tests.

Figure 5.5 shows the run time of the Allen-Cahn equation. The Allen-Cahn equation is introduced in chapter 3.4. Figure 5.3 shows the run time in seconds plotted against the number of cores used on one node. The use of Singularity is compared for a different number of timesteps n_c for $n_x = 128$, $n_f = 10$

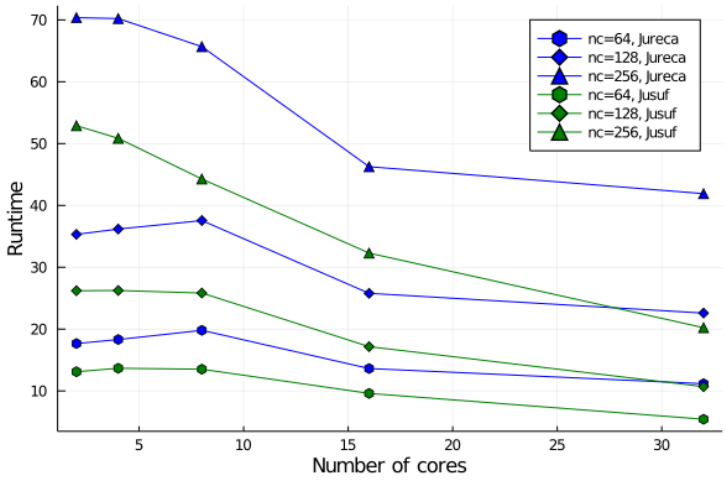


Figure 5.5.: Run time in seconds of the Allen-Cahn equation run on Jusuf with Open MPI compared to the runtime on Jureca run with a container

and IMEX-Euler as integrator. In this plot, the run time on Jusuf in green and Jureca in blue are compared. The run time of the container on Jureca was discussed in 5.1. The run time on Jusuf compared to the run time on Jureca is smaller. It can be seen that on Jusuf the more processors are used the smaller the time gets. In comparison to Jureca Jusuf has two processors with 64 cores each. No hyperthreading is needed to run with 32 cores on Jusuf like on Jureca. This concludes with a better run time and speedup on Jusuf for 32 cores.

A rather different host would be a cloud system. Clouds have become more popular these days and can run HPC applications as well. The advantage is that the user can specifically choose its own configurations of hardware, operating system, scheduler and software. Also, working on HPC systems can lead to long waiting times until the application can run, because of the Job scheduler. Cloud systems try to solve this problem. The disadvantage is that to start working with a cloud system you need time to understand how it works and what is possible. For this test case, Amazon Web Services (AWS) [Amaa] is chosen as the cloud computing platform. AWS is one of the largest cloud computing providers in the world. AWS has a lot of services and resources available to its users. The goal is to show that the container build for Jureca with Open MPI and Julia can run in the cloud as well. To properly run the HPC application on AWS the user would have to create a cluster on which the HPC application could run. AWS provides tools for that like StarCluster, CycleCloud or AWS Parallel Cluster. To test, if the Singularity container could run on AWS it is sufficient to create a virtual machine to run in. The cluster on which the virtual machine should run is the Elastic Cluster 2, also called EC2

[Amab]. The hardware instance chosen to run the virtual machine is called C5. C5 features custom 2nd generation Intel Xeon Scalable Processors with a sustained all-core turbo frequency of 3.6GHz and a maximum single-core turbo frequency of 3.9GHz. One processor consists of two cores. C5 instances are chosen because they are made to run advanced compute-intensive workloads at a low price. The C5 instance chosen costs $0.085 per Hour for example. THe costs are charged exactly to the second used. As an operating system, Ubuntu 18.04 was chosen. For this virtual machine, only 2 cores were used. This is due to the fact that for working with more cores another processor would have to be used. To accomplish running multiple processors a virtual cluster has to be created. To run the container Singularity and Open MPI had to be installed inside the virtual machine. The container was able to run the application on the virtual machine.

A comparison of the run time of the different host systems is done in table 5.1. The Allen-Cahn equation with Parareal and IMEX Euler as the integrator, $n_c = 128$ and $n_f = 10$ was used here to make the comparison. The run time was tested for $n_x = 128$ and $n_x = 256$ for the host systems discussed before.

	$n_x = 128$	$n_x = 256$
Jureca native	36.09	70.94
Jureca	35.32	70.36
Jusuf	26.2	52.88
AWS	35.63	68.68

Table 5.1.: Runtime in seconds of the Allen-Cahn equation with Parareal and IMEX Euler as integrator, $n_c = 128$ and $n_f = 10$ compared for different hosts

It can be seen that the shortest run time can be reached when running the container on Jusuf. It can be seen that with doubling the number of steps in the time domain n_c the run time doubles as well.

5.3. Singularity with a different MPI version

Open MPI is not the only MPI implementation. Others existing are Parastation MPI [MPIa], MPICH [MPIb] and Intel MPI [Int]. Parastation MPI is not compatible with Julia [Lan], so it couldn't be used here. MPICH is not installed on Jureca so it could not be used either. Intel MPI has an installation on Jureca and is compatible with Julia. It was also used in chapter 3 to test the applications. The build script of the Intel MPI Singularity container for a Julia MPI application can be seen in the appendix. The build script is more complex in this case than for Open MPI. The main difference between the build script presented in 4.3 and the one used for the Intel MPI container is

that the base image, in this case, is the latest Ubuntu image, the environment variables have to be adapted and instead of downloading the Intel MPI inside the container the compressed Intel MPI directory is downloaded outside of the container and only copied into the container. Also as shown in listing 5.1 the installation of Intel MPI has to be a silent installation. After the installation of the mpivars.sh, script has to be executed to set the environment variables. This line has to be included in the environment section as well.

Listing 5.1: Specifications of the container with Intel MPI

```
    ./install.sh --silent silent.cfg --accept_eula

3   . /opt/intel/compilers_and_libraries_2020/linux/
      mpi/intel64/bin/mpivars.sh
```

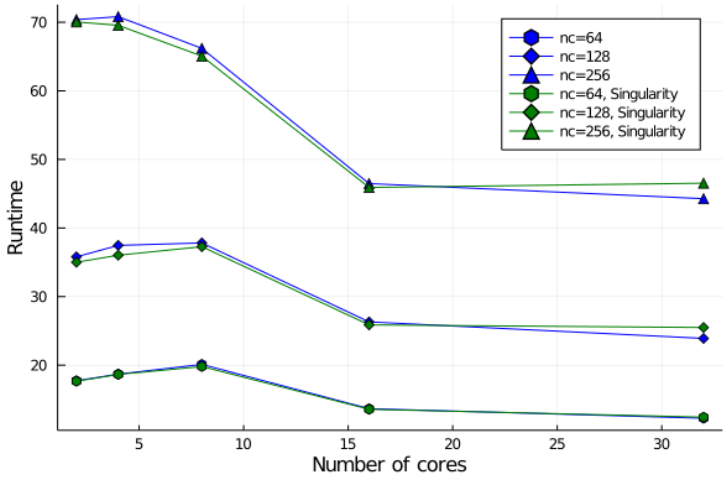


Figure 5.6.: Run time in seconds of the Allen-Cahn equation run on Jureca with Intel MPI comparing a native run and a run with a container

Figure 5.6 shows the run time of the Allen-Cahn equation with Intel MPI. The figure shows the run time in seconds plotted against the number of cores used on one node. The use of singularity is compared for different number of time steps n_c for $n_x = 128$, $n_f = 10$ and IMEX Euler as integrator. Compared to figure 5.3 the run time is better and the run time of the singularity container is closer to the native run time. The overhead here is only up to 4%.

Figure 5.7 shows the run time of the Allen-Cahn equation against the number of cores used. Here 1, 2 and 4 nodes are used with 16 cores each. This tests if there is an overhead when using inter-node communication for Intel MPI. The run time of the container run and the native run is nearly the same. Only an overhead of up to 1% can be seen here.

In figure 5.8 Open MPI and Intel MPI are compared as a native run and a containerized run. This figure combines figure 5.4 and figure 5.7. The figure

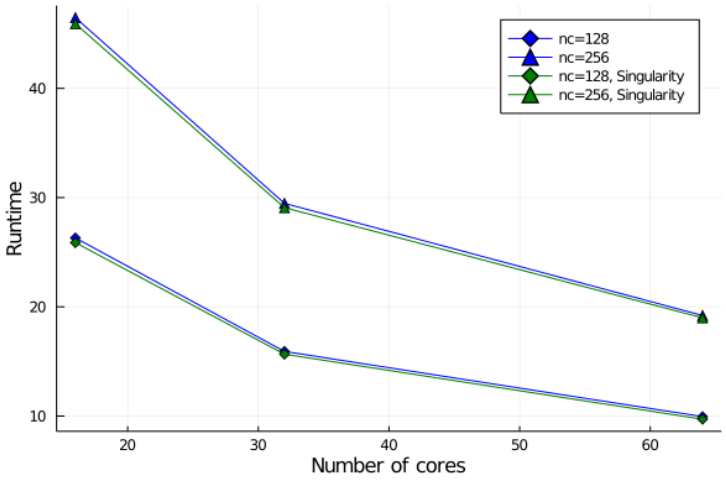


Figure 5.7.: Run time in seconds of the Allen-Cahn equation run on Jureca on multiple nodes with Intel MPI comparing a native run and a run with a container

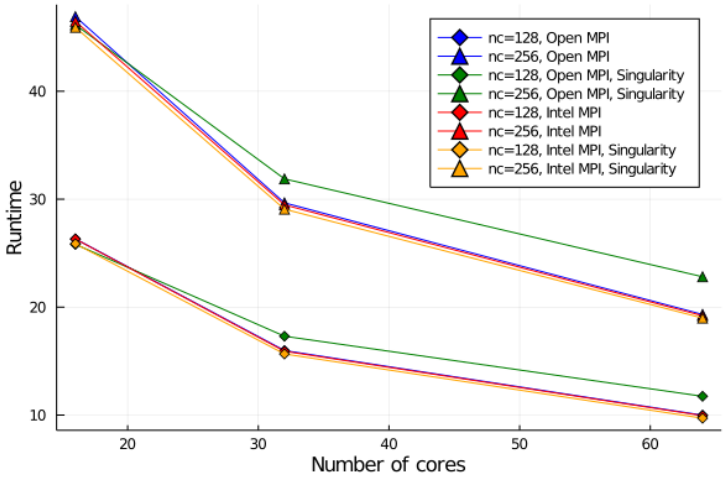


Figure 5.8.: Run time in seconds of the Allen-Cahn equation run on Jureca on multiple nodes comparing Open MPI and Intel MPI as well a native run and a run with a container

shows the run time of the Allen-Cahn equation against the number of cores used for 1, 2 and 4 nodes and 16 cores per node. It can be seen that for each coarse step size n_c the Singularity container with Open MPI has the slowest run time. On the other side, the Singularity container with Intel MPI has nearly the same run time as both native versions. This means that the Intel MPI Singularity container for a Julia MPI application on Jureca can be run without any overhead.

6. Summary and outlook

In this work, Parareal was implemented using Julia and MPI. First Parareal was introduced and the parallelization strategy was presented. In chapter 3 it was shown that the algorithm can be applied to the Lorenz equation, the heat equation, and the Allen-Cahn equation. It was shown that Parareal converges for each of the test cases with a decent number of iteration steps. The speedup of the algorithm is not ideal for the setting presented in chapter 3. After that containerization was introduced with a focus on Singularity and how to build and execute Singularity containers. A container for Julia 1.3.1 with Open MPI was further discussed in 4.3. This container is built with the hybrid approach which is introduced in chapter 4.3. This container runs on Jureca. For each of the test cases run time tests were done. It can be concluded that for the Lorenz equation the run time was too small to get decent results with the Singularity container. The run time for the heat equation and the Allen-Cahn equation with the container was, on one node, nearly the same as without a container. So there was nearly no overhead. For the Allen-Cahn test case, the performance for more nodes was tested as well. The more nodes are taken, the larger the overhead is. The best performance was reached if the number of BLAS threads is 1. The portability of the container was shown in chapter 5.2. The container could be ported onto a cloud-based host as well as the supercomputer Jusuf. The portation to Jusuf was not as easy due to the fact that the currant software stage did not have the required version of MPI. By using an other software stage running the container on Jusuf could be reached. This shows that the container for our application is dependent on the MPI version used on the host system and therefore the portability is limited. It was possible to run a container with another MPI version named Intel MPI as well. The results of running Intel MPI on Jureca were better than running Open MPI. Especially when doing using more than one node the Intel container had no overhead compared to the Open MPI container. This was discussed in more detail in chapter 5.3.

Altogether it can be said that the goal of this work to have a working Parareal algorithm written in Julia that runs on the Supercomputer Jureca inside a Singularity container could be reached. The implementation on the other hand is not adjustable for different problems and integrators. This is due to the problem that using the ”DifferentialEquation.jl” package was not possible like shown in chapter 3.1. By using a singularity container on Jureca the usual workflow by loading the provided modules could be bypassed for

Julia. MPI still had to be loaded and the MPI version loaded had to be compatible with the MPI version inside the container. Working with Julia there had to be a MPI version inside the container during the building of the container. This could lead to performance issues as shown in chapter 5.1. It would be more convenient to bind the MPI version available on the host system into the container. Further topics that can be explored are the following:
HPC on a Cloudsystem - Cloud computing was introduced in 5.2. It was shown that the singularity container for the application runs on the AWS cloud. Running an own cluster in the cloud was shortly introduced but not tested. It would be interesting to see how well the cluster performs compared to Jureca for example and if it can be used as an alternative or an addition to HPC computing.
Variation in solver and stepsize - It would be interesting to see how well Parareal performs when a different solver as the fine solver is chosen. For example a Runge-Kutta method with a different order for the Lorenz system or a higher-order solver for the heat equation like SBDF2. It would also be interesting to look at the changes in the error, iterations, and timings for a changed n_f.

A. Appendix

A.1. Singularity Definition File

These buildscrips build a image for my private computer and can differ on other hostsystems.

OpenMPI:

Listing A.1: Definition file Open MPI

```
BootStrap: docker
From: bitnami/minideb:jessie

%files
    # add your files here


%environment
    # Set env variables so we can compile our
       application for Julia
    export JULIA_DEPOT_PATH=$PWD/containerhome/.julia
       :/user/.julia
    export PATH=/opt/julia/bin:$PATH
    export HOME=/user

    # Set env variables so we can compile our
       application for OpenMPI
    export OMPI_DIR=/opt/ompi
    export SINGULARITY_OMPI_DIR=$OMPI_DIR
    export SINGULARITYENV_APPEND_PATH=$OMPI_DIR/bin
    export SINGULAIRTYENV_APPEND_LD_LIBRARY_PATH=
       $OMPI_DIR/lib

    export PATH=$OMPI_DIR/bin:$PATH
    export LD_LIBRARY_PATH=$OMPI_DIR/lib:
       $LD_LIBRARY_PATH
    export MANPATH=$OMPI_DIR/share/man:$MANPATH

%post
```

```
    #Configurations for Julia

    mkdir -p /user
    export HOME=/user
    export JULIA_DEPOT_PATH=/user/.julia
    export PATH=/opt/julia/bin:$PATH

    JULIA_MAJOR=1.3
    JULIA_MINOR=.1

    install_packages curl tar gzip openssh-client git
        ca-certificates

    curl -k https://julialang-s3.julialang.org/bin/
       linux/x64/$JULIA_MAJOR/julia-
       $JULIA_MAJOR$JULIA_MINOR-linux-x86_64.tar.gz >
       julia.tar.gz
    mkdir /opt/julia
    tar xzf julia.tar.gz -C /opt/julia
    rm julia.tar.gz
    mv /opt/julia/$(cd /opt/julia; echo julia-*)/*
       opt/julia/
    rm -rf /opt/julia/$(cd /opt/julia; echo julia-*)

    rm -rf /opt/julia/share/doc
    rm -rf /opt/julia/share/icons
    rm -rf /opt/julia/share/appdata
    rm -rf /opt/julia/share/applications
    rm -rf /opt/julia/share/man
    rm -rf /opt/julia/share/julia/test
    rm /opt/julia/LICENSE.md
    rm -rf /opt/julia/etc
    rm -rf /opt/julia/include

    mkdir -p /user/.julia/environments/$JULIA_MAJOR

# Configurations for OpenMPI

    cd

    apt-get update && apt-get install -y wget git
       bash gcc gfortran g++ make file
```

```
    export OMPI_DIR=/opt/ompi
    export OMPI_VERSION=4.0.2
    export OMPI_URL="https://download.open-mpi.org/
       release/open-mpi/v4.0/openmpi-$OMPI_VERSION.tar
       .gz"
    mkdir -p /tmp/ompi
    mkdir -p /opt

    cd /tmp/ompi
    wget -O openmpi-$OMPI_VERSION.tar.gz https://
       download.open-mpi.org/release/open-mpi/v4.0/
       openmpi-4.0.2.tar.gz
    tar xfvz openmpi-$OMPI_VERSION.tar.gz
    # Compile and install
    cd /tmp/ompi/openmpi-$OMPI_VERSION && ./configure
        --prefix=$OMPI_DIR && make install
    # Set env variables so we can compile our
       application
    export PATH=$OMPI_DIR/bin:$PATH
    export LD_LIBRARY_PATH=$OMPI_DIR/lib:
       $LD_LIBRARY_PATH
    export MANPATH=$OMPI_DIR/share/man:$MANPATH

#JULIA

    # Put in your Julia Specifications here for
       packages

    julia -e 'using Pkg; Pkg.add(PackageSpec(name="
       ArgParse", version="1.1.0"));'
    julia -e 'using Pkg; Pkg.add(PackageSpec(name="
       CSV", version="0.7.7"));'
    julia -e 'using Pkg; Pkg.add(PackageSpec(name="
       DataFrames", version="0.21.7"));'
    julia -e 'using Pkg; Pkg.add("LinearAlgebra");'
    julia -e 'using Pkg; Pkg.add("Profile");'
    julia -e 'using Pkg; Pkg.add("Statistics");'

    # Combine MPI and Julia here

    export CC=`which mpicc`
    export FC=`which mpif90`
    julia -e 'using Pkg; Pkg.add(PackageSpec(name="
```

```
        MPI", version="0.15.1"));'
    julia --project -e 'ENV["JULIA_MPI_BINARY"]="
        system"; using Pkg; Pkg.build("MPI"; verbose=
        true)'

    # Maybe make your files runable with: chmod 774 /
        opt/parareal_imex_ac.jl
```

IntelMPI:

Listing A.2: Definition file Intel MPI

```
Bootstrap:docker
From:ubuntu:latest

%files
    ./l_mpi_2019.9.304.tgz /opt
    ../parareal-julia/src/parareal_imex_ac.jl /opt
    ../parareal-julia/src/parareal_imex_heq.jl /opt
    ../parareal-julia/src/lorenz_euler.jl /opt

%environment
    # add local directory for precompile files
    export JULIA_DEPOT_PATH=$PWD/containerhome/.julia
        :/user/.julia
    export PATH=/opt/julia/bin:$PATH
    export HOME=/user

    #MPI
    export PATH=/opt/intel/
        compilers_and_libraries_2020/linux/mpi/intel64/
        bin:$PATH
    export LD_LIBRARY_PATH=/opt/intel/bin/
        compilers_and_libraries_2020/linux/mpi/intel64/
        lib:$LD_LIBRARY_PATH

    . /opt/intel/compilers_and_libraries_2020/linux/
        mpi/intel64/bin/mpivars.sh


%post
#JUlia
    mkdir -p /user
    export HOME=/user
    export JULIA_DEPOT_PATH=/user/.julia
```

```
28      export PATH=/opt/julia/bin:$PATH

        JULIA_MAJOR=1.3
        JULIA_MINOR=.1
        # could also be .0
33
        # minideb specific install script, shaves off
           about 20mb compared to apt-get
        apt-get update && apt-get install -y  curl tar
           gzip openssh-client git ca-certificates

        curl -k https://julialang-s3.julialang.org/bin/
           linux/x64/$JULIA_MAJOR/julia-
           $JULIA_MAJOR$JULIA_MINOR-linux-x86_64.tar.gz >
           julia.tar.gz
38      mkdir /opt/julia
        tar xzf julia.tar.gz -C /opt/julia
        rm julia.tar.gz
        mv /opt/julia/$(cd /opt/julia; echo julia-*)/*
           opt/julia/
        rm -rf /opt/julia/$(cd /opt/julia; echo julia-*)
43
        rm -rf /opt/julia/share/doc
        rm -rf /opt/julia/share/icons
        rm -rf /opt/julia/share/appdata
        rm -rf /opt/julia/share/applications
48      rm -rf /opt/julia/share/man
        rm -rf /opt/julia/share/julia/test
        rm /opt/julia/LICENSE.md
        rm -rf /opt/julia/etc
        rm -rf /opt/julia/include
53
        mkdir -p /user/.julia/environments/$JULIA_MAJOR

    #mpi
        cd
58
        apt-get update && apt-get install -y wget git
           bash gcc gfortran g++ make file cpio

        cd ../opt
        ls -ll
63      pwd
```

```
    cp l_mpi_2019.9.304.tgz /tmp
    cd /tmp
    tar -xzf l_mpi_2019.9.304.tgz

    cd l_mpi_2019.9.304
    ls -ll
    #cp <license key>.lic ../
    cat silent.cfg
    ./install.sh --silent silent.cfg --accept_eula

    . /opt/intel/compilers_and_libraries_2020/linux/
       mpi/intel64/bin/mpivars.sh

    # Set env variables so we can compile our
       application
    #export PATH=$OMPI_DIR/bin:$PATH
    #export LD_LIBRARY_PATH=$OMPI_DIR/lib:
       $LD_LIBRARY_PATH
    #export MANPATH=$OMPI_DIR/share/man:$MANPATH

    export CC=`which mpicc`
    export FC=`which mpif90`

#JULIA



    julia -e 'using Pkg; Pkg.add(PackageSpec(name="
       ArgParse", version="1.1.0"));'
    julia -e 'using Pkg; Pkg.add(PackageSpec(name="
       CSV", version="0.7.7"));'
    julia -e 'using Pkg; Pkg.add(PackageSpec(name="
       DataFrames", version="0.21.7"));'
    julia -e 'using Pkg; Pkg.add("LinearAlgebra");'
    julia -e 'using Pkg; Pkg.add("Profile");'
    julia -e 'using Pkg; Pkg.add("Statistics");'
    julia -e 'using Pkg; Pkg.add(PackageSpec(name="
       MPI", version="0.15.1"));'

    julia --project -e 'ENV["JULIA_MPI_BINARY"]="
```

```
        system"; using Pkg; Pkg.build("MPI"; verbose=
        true)'

    # mkdir -p /opt/.julia/environments/v1.3/

    # julia -e 'using Pkg;  Pkg.status()'

    chmod 774 /opt/parareal_imex_ac.jl
    chmod 774 /opt/parareal_imex_heq.jl
    chmod 774 /opt/lorenz_euler.jl
```

Bibliography

[20] *Anaconda Software Distribution*. Version Vers. 2-2.4.0. 2020. URL: https://docs.anaconda.com/.

[AC79] Samuel M. Allen and John W. Cahn. "A microscopic theory for antiphase boundary motion and its application to antiphase domain coarsening". In: *Acta Metallurgica* 27.6 (1979), pp. 1085–1095. ISSN: 0001-6160. DOI: https://doi.org/10.1016/0001-6160(79)90196-2. URL: http://www.sciencedirect.com/science/article/pii/0001616079901962.

[Amaa] Inc. Amazon Web Services. *AWS*. URL: https://docs.aws.amazon.com/ (visited on 12/08/2020).

[Amab] Inc. Amazon Web Services. *EC2*. URL: https://docs.aws.amazon.com/ec2/index.html (visited on 12/08/2020).

[Bau+14] A.-M. Baudron et al. "The Parareal in Time Algorithm Applied to the Kinetic Neutron Diffusion Equation". In: *Domain Decomposition Methods in Science and Engineering XXI*. Cham: Springer International Publishing, 2014, pp. 437–445. ISBN: 978-3-319-05789-7.

[Bez+14] Jeff Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *CoRR* abs/1411.1607 (2014).

[Bla+02] L Susan Blackford et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[BZ89] Alfredo Bellen and Marino Zennaro. "Parallel algorithms for initial-value problems for difference and differential equations". In: 1989.

[Cla+19] Andrew T. Clarke et al. "Parallel-in-time integration of Kinematic Dynamos". arXiv:1902.00387 [physics.comp-ph]. 2019. URL: https://arxiv.org/abs/1902.00387.

[CP93] Philippe Chartier and Bernard Philippe. "A parallel shooting technique for solving dissipative ODE's". In: *Computing* 51 (Aug. 1993), pp. 209–236. DOI: 10.1007/BF02238534.

[DPS05] Lisandro Dalcin, Rodrigo Paz, and Mario Storti. “MPI for Python”. In: *Journal of Parallel and Distributed Computing* 65.9 (2005), pp. 1108–1115. ISSN: 0743-7315. DOI: http://dx.doi.org/10.1016/j.jpdc.2005.03.010.

[Dyb09] R. Kent Dybvig. *The Scheme Programming Language, 4th Edition*. 4th. The MIT Press, 2009.

[Gab+04] Edgar Gabriel et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Proceedings, 11th European PVM/MPI Users’ Group Meeting*. Budapest, Hungary, 2004, pp. 97–104.

[GH08] Martin Gander and Ernst Hairer. “Nonlinear Convergence Analysis for the Parareal Algorithm”. In: vol. 60. Jan. 2008, pp. 45–56. DOI: 10.1007/978-3-540-75199-1_4.

[Git] GitHub. *Atom*. URL: https://atom.io/ (visited on 07/20/2018).

[GT15] S. Kim G. Ariel and R. Tsai. *Parareal methods for highly oscillatory ordinarydifferential equations*. 2015.

[GV07] Martin Gander and Stefan Vandewalle. “Analysis of the Parareal Time-Parallel Time-Integration Method”. In: *SIAM J. Scientific Computing* 29 (Jan. 2007), pp. 556–578. DOI: 10.1137/05064607X.

[Inc] Sylabs Inc. *Singularity*. URL: https://sylabs.io/guides/3.6/user-guide/introduction.html (visited on 12/08/2020).

[Int] Intel. *Intel MPI*. URL: https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-mpi-for-linux/top.html (visited on 12/08/2020).

[JSC] JSC. *Jusuf*. URL: https://fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUSUF/Configuration/Configuration_node.html (visited on 12/08/2020).

[Jül16] Jülich Supercomputing Centre. “JURECA: General-purpose supercomputer at Jülich Supercomputing Centre”. In: *Journal of large-scale research facilities* 2.A62 (2016). DOI: 10.17815/jlsrf-2-121. URL: http://dx.doi.org/10.17815/jlsrf-2-121.

[Jup] Project Jupyter. *Jupyter*. URL: http://jupyter.org/ (visited on 12/12/2017).

[KR14] Rolf Krause and Daniel Ruprecht. “Hybrid Space–Time Parallel Solution of Burgers’ Equation”. In: *Domain Decomposition Methods in Science and Engineering XXI*. Cham: Springer International Publishing, 2014, pp. 647–655. ISBN: 978-3-319-05789-7.

[KSB17] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific containers for mobility of compute". In: *PLOS ONE* 12.5 (May 2017), pp. 1–20. DOI: 10.1371/journal.pone.0177459. URL: https://doi.org/10.1371/journal.pone.0177459.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[Lan] The Julia Language. *Julia MPI*. URL: https://juliaparallel.github.io/MPI.jl/stable/configuration/ (visited on 12/08/2020).

[LMT01] Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. "Résolution d'EDP par un schéma en temps « pararéel »". In: 2001.

[Mer14] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583.

[MPIa] Parastation MPI. *Parastation MPI*. URL: https://docs.par-tec.com/html/psmpi-userguide/index.html (visited on 12/08/2020).

[MPIb] MPICH. *MPICH*. URL: https://www.mpich.org/ (visited on 12/08/2020).

[NER] NERSC. *Shifter*. URL: https://github.com/NERSC/shifter (visited on 12/08/2020).

[Nie64] Jürg Nievergelt. "Parallel methods for integrating ordinary differential equations". In: *Commun. ACM* 7 (1964), pp. 731–733.

[Ora] Oracle. *VirtualBox*. URL: https://www.virtualbox.org/manual/ch01.html (visited on 12/08/2020).

[Pad11] "Message Passing Interface (MPI)". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1116–1116. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2085. URL: https://doi.org/10.1007/978-0-387-09766-4_2085.

[PyP] PyPa. *Pip*. URL: https://pip.pypa.io/en/stable/ (visited on 12/08/2020).

[RN17] Christopher Rackauckas and Qing Nie. "Differentialequations. jl–a performant and feature-rich ecosystem for solving differential equations in julia". In: *Journal of Open Research Software* 5.1 (2017).

[RSK16] Daniel Ruprecht, Robert Speck, and Rolf Krause. “Parareal for Diffusion Problems with Space- and Time-Dependent Coefficients”. In: *Domain Decomposition Methods in Science and Engineering XXII* (2016), pp. 371–378. ISSN: 2197-7100. DOI: 10.1007/978-3-319-18827-0_37. URL: http://dx.doi.org/10.1007/978-3-319-18827-0_37.

[Sec] Triad National Security. *CharlieCloud*. URL: https://hpc.github.io/charliecloud/ (visited on 12/08/2020).

[SR05] Gunnar Andreas Staff and Einar M. Ronquist. “Stability of the Parareal Algorithm”. In: *Domain Decomposition Methods in Science and Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 449–456. ISBN: 978-3-540-26825-3.

[SST97] Prasenjit Saha, Joachim Stadel, and Scott Tremaine. “A Parallel Integration Method for Solar System Dynamics”. In: (1997).

[ZD09] Jian Zhang and Qiang Du. “Numerical studies of discrete approximations to the Allen-Cahn equation in the sharp interface limit”. In: *SIAM* (2009).