FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Serial and parallel time-integration methods for the Hierarchical Hybrid Grids framework

Anton Artemov

Master Thesis

Serial and parallel time-integration methods for the Hierarchical Hybrid Grids framework

Anton Artemov

Master Thesis

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Dr. B. Gmeiner
Bearbeitungszeitraum:	09.04.2015 – 31.07.2015

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Master Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 28. Juli 2015 .....

Abstract

In this master thesis several time-integration schemes are considered in details. The schemes are derived theoretically for 1D ODE IVP, their properties like accuracy and stability are also described. Then, a parallel-in-time parareal algorithm is discussed together with its features. All the schemes and parareal algorithm are implemented within the HHG framework, and the encountered implementation issues are stated. Numerical tests are performed on a model 3D inhomogeneous heat equation in order to check the correctness of the implementation. Some tests are also done in order to find the limits of applicability of parareal algorithm. The behavior of the algorithm for extreme values of some parameters is analyzed.

List of Figures

Listings

List of Tables

Contents

1 Introduction

HHG, or Hierarchical Hybrid Grids [6], is a high performance framework for linear finite elements, which exploits spatial parallelism by efficient combination of regular refinement and grid decomposition. Such approach allows to treat the structured regions of refined grid hierarchy with stencil-based structures. In turn, this allows to implement standard multigrid component algorithms efficiently. It is written in object oriented manner and parallelized with MPI.

The HHG framework has been successfully applied to a real-world problem of Earth mantle convection simulation in Terra-Neo project [1]. In this project, the following system of equation is solved:

$$-\Delta u + \nabla p = -RTe_r, \tag{1.0.1}$$

$$\nabla \cdot u = 0, \tag{1.0.2}$$

$$\partial_t T + u \cdot \nabla T = \Delta T, \tag{1.0.3}$$

where u is a velocity field, p is a pressure field, R is the Rayleigh number, e_r is the unit vector which points to the center, T is a temperature field. Some appropriate boundary conditions are applied. Equations (1.0.1) and (1.0.2) are solved using FEM discretization on hierarchy of grids. Equation (1.0.3) is solved using streamline-upwind Petrov-Galerkin method combined with forward Euler for time-integration.

The most important feature of the system is that it is to be solved in extreme scale (up to 10^{12} points in the domain) and in a very long time interval. Moreover, equations (1.0.1) and (1.0.2) are discretized with the 2nd order od accuracy, while the only used time-integration scheme for temperature equation is of the 1st order.

All this motivates us to add more time-integration schemes to HHG. Moreover, a flexible parallel-in-time method called parareal is implemented on top of added integrators and tested on a model problem. Although this method is for integrating ODE or ODE systems in time, it is also applicable to PDE problems like the one in Terra-Neo.

1.1 Literature overview

Numerical methods for ODE integration are essentially divided into two classes. The 1st class consists of methods which use a single starting value (so-called one-step methods). The 2nd class includes methods, which use several values (so-called multistep methods).

Moreover, a second kind of classification is also used. A method is called explicit, if a new value is found explicitly, while in implicit methods it requires additional effort to find it, since the value of interest is in both sides of resulting equation.

In our case we restrict ourselves to explicit methods only. The reason for that is that in most problems a time-dependent equation is coupled with several more equations. Thus it is difficult to interpret correctly what is an implicit method for such system.

Explicit methods are more universal and can be applied to almost any kind of time-dependent problems. Of course, stiff problems are to be treated with great attention, since implicit methods are more suitable for them.

Historically, the first numerical method for solution of initial value problem (IVP)

$$y' = f(x, y),\ y(x_0) = y_0, \tag{1.1.1}$$

was introduced by Leonard Euler [9] in 1768. His idea was to approximate the next value using the tangent of the curve at the current point, or, in other words, to use linearization.

Later, Runge [28] and Heun [16] constructed new methods by adding one or two steps to Euler method. In 1901 Kutta [19] formulated the general scheme which is currently known as Runge-Kutta method.

First multistep methods were developed by Adams in order to solve a problem introduced by Bashforth [4] in 1883. In contrast to one-step methods, where the numerical solution is calculated using only initial value and the equation itself, Adams-Bashforth algorithm consists of two stages: in the 1st stage, or the starting procedure, values $y_1, \ldots, y_{k-1}$, are computed. In the 2nd one, a multistep formula is applied in order to compute approximation of $y(x_0 + kh)$. Then this formula is applied recursively to compute next values y_{k+1}, y_{k+2}, and so on.

Starting values can be obtained using different approaches. One idea is to use Taylor expansion of the exact solution. Another idea is to use a one-step method from Runge-Kutta family of methods.

First parallel-in-time method was proposed by Nievergelt [27] in 1964. His idea was to divide the time-domain into several subdomains and then to solve a number of problems on these subdomains with following parallel interpolation. Now this approach is not considered as usable, but Khalaf and Hutchinson [17], as well as Kiehl [18], extended Nievergelt's idea through the use of parallel shooting techniques.

In 1995 Burrage published a book [7], where a classification of state-of-the-art parallel methods for solving ODEs was offered. Here we present this classification as in [22]:

- **Parallelism across the system**. This class includes methods which utilize the fact that the right hand side of the equation can be partitioned over its various components, and computations of these components are distributed among different processors. This class is the most efficient for high-dimensional systems, such as molecular or gravitational simulations.

- **Parallelism across the method**. This idea is related to the numerical scheme chosen to integrate the equation and thus the efficiency of the system very much depends on the scheme.

- **Parallelism across the time**. Here the idea is to split the time interval into a set of smaller subintervals and solve the equation simultaneously on the subintervals. Most of the techniques in this class are multishooting methods, varying from parallelism across the steps by Bellen and Zennaro [5] to waveform relaxation methods by E. Lelarasmee, A. E. Ruehli and A. L. Sangiovanni-Vincentelli [20] and multigrid approaches by Hackbush [14].

Many of the methods can be applied to certain classes of problems, some of methods can't handle non-linear problems, some of them are limited to the hardware architecture, some introduce only small-scale parallelism.

The parareal algorithm considered in this thesis belongs to the 3rd category of the methods. It can be shown that the algorithm can be viewed as a 2-level multigrid with aggressive time-coarsening or as a multishooting technique [11]. The algorithm itself is free of majority of mentioned defects and has several advantages:

- relatively easily implemented in it's most basic form;
- has been successfully applied to non-linear problems;
- has been tested on wide range of problems;
- can be wrapped around existing spatial parallelism;
- fault tolerant.

Moreover, parareal can be combined with any classical time-integration scheme. Thus it is essential to add the integrating schemes into HHG gradually, starting from simple forward Euler scheme and ending with parareal on the top.

1.2 Outline

The aim of this master thesis is to study and implement different time-integration techniques within the HHG framework. The work consists of 4 main parts:

- Classical time-integration schemes;
- Parareal algorithm;
- Implementation details;
- Experimental numerical results.

The next chapter contains the detailed description of the classical integration schemes implemented and added to HHG main repository.

2 Classical time-integration schemes

In this chapter we present some classical time-integration schemes implemented in HHG. The derivation is done for a simple 1D IVP:

$$\frac{du}{dt} = f(t,u), u(t_0) = u_0, t \in [t_0, t_{end}], \tag{2.0.1}$$

where f is in general a nonlinear function of 2 variables. The time interval is assumed to be discretized with some stepsize h such that $(t_{end} - t_0)/h = n, n \in \mathbb{N}$.

2.1 Forward Euler

To derive the forward Euler scheme, one does Taylor expansion of the solution of (2.0.1) around the initial point as follows:

$$u(t) = u(t_0) + u'(t_0)(t - t_0) + \frac{1}{2}u''(\xi)(t - t_0)^2. \tag{2.1.1}$$

When $t = t_1$, we obtain

$$u(t_1) = u_0 + hf(t_0, u_0) + \frac{1}{2}u''(\xi_1)h^2. \tag{2.1.2}$$

Replacing $u(t_1)$ with approximation u_1 and neglecting the remainder term $\frac{1}{2}u''(\xi_1)h^2$ one gets forward Euler formula

$$u_1 = u_0 + hf(t_0, u_0). \tag{2.1.3}$$

Applying this formula several times sequentially, we derive the forward Euler scheme:

$$u_{i+1} = u_i + hf(t_i, u_i). \tag{2.1.4}$$

The remaining term $\frac{1}{2}u''(\xi_1)h^2, \xi_1 \in [t_0, t_1]$ in (2.1.2) characterizes the local error, i.e. the error made in a single step. Obviously, applying the scheme (2.1.4) n times might lead to accumulation of local errors.

So, the global error of the method is the error accumulated after n steps done. By substitution of the exact solution into the scheme it can be shown that if a local order of accuracy is $p + 1$, then the global one is p [31]. Thus, the forward Euler scheme is of the 1st order of accuracy.

One can do Taylor expansion of the exact solution at t_{i+1} up to p-th order

$$u(t_{i+1}) = u(t_i) + hu'(t_i) + \frac{1}{2!}h^2u''(t_i) + \ldots + \frac{1}{p!}h^pu^{(p)}(t_i) + O(h^{p+1}). \tag{2.1.5}$$

If $p = 1$ then it is the derived forward Euler scheme.

If $p = 2$, then

$$u(t_{i+1}) = u(t_i) + hu'(t_i) + \frac{1}{2!}h^2u''(t_i) + O(h^3). \tag{2.1.6}$$

The first derivative can be approximated using the original ODE (2.0.1):

$$u'(t_i) = f(t_i, u(t_i)) \approx f(t_i, u_i). \tag{2.1.7}$$

To find approximation of the 2nd derivative, one applies the chain rule:

$$u''(x) = f'_t(t_i, u(t_i)) + f'_u(t_i, u(t_i))u'(t_i) \approx f'_t(t_i, u_i) + f'_u(t_i, u_i)f(t_i, u_i). \tag{2.1.8}$$

Substituting expressions for $u(t_i),\ u'(t_i),\ u''(t_i)$ in (2.1.6) we get the following scheme for computing approximations $u_i \approx u(t_i)$:

$$u_{i+1} = u_i + h\left(f(t_i, u_i) + \frac{h}{2}(f'_t(t_i, u_i) + f'_u(t_i, u_i)f(t_i, u_i))\right). \tag{2.1.9}$$

This method is called an improved Euler scheme. When $i = 0$ formulas for u' and u'' are exact and $u(0) = u_0$, so the only error in the very first step is the error introduced by the truncation of Taylor expansion. Thus, the local error of improved Euler is proportional to h^3, and the global one is $O(h^2)$.

The major drawback of the improved Euler scheme is the usage of partial derivatives of the right-hand side, which can be expensive to compute. Moreover, there exist methods of higher accuracy which do not use these derivatives.

2.2 Midpoint and Heun's schemes

To derive the Heun's and the midpoint methods a different technique is used. Both these methods belong to explicit Runge-Kutta set of methods. The main idea of derivation is to look for approximations of $u(t_{i+1})$ in the following form:

$$u_{i+1} = u_i + h\varphi(t_i, u_i, h), \tag{2.2.1}$$

where $\varphi(t, u, h)$ is a function which approximates Taylor expansion up to p-th order and does not contain partial derivatives of $f(t, u)$.

Assuming $\varphi(t, u, h) \equiv f(t, u)$, we obtain exactly the forward Euler scheme. So, forward Euler is an explicit Runge-Kutta scheme of the 1st order.

To derive a method of the 2nd order, we assume that function φ from (2.2.1) is of the following form:

$$\varphi(t, u, h) = c_1 f(t, u) + c_2 f(t + ah, u + bhf(t, u)). \tag{2.2.2}$$

Parameters a, b, c_1, c_2 are to be tuned so that the scheme written according to (2.2.2)

$$u_{i+1} = u_i + h\left(c_1 f(t, u) + c_2 f(t + ah, u + bhf(t, u))\right) \tag{2.2.3}$$

has the local error equal to $O(h^3)$. This can be easily done using Taylor expansion of f up to linear terms:

$$f(t + ah, u + bhf(t, u)) = f(t, u) + f'_x(t, u)ah + f'_y(t, u)bhf(t, u) + O(h^2). \tag{2.2.4}$$

Inserting this expression to (2.2.3) gives the following:

$$u_{i+1} = u_i + h\left((c_1 + c_2)f(t_i, u_i) + h(c_2 a f'_x(t_i, u_i) + c_2 b f'_y(t_i, u_i)f(t_i, u_i))\right) + O(h^3). \tag{2.2.5}$$

The comparison of (2.2.5) with Taylor expansion of the exact solution up to the 2nd order terms (2.1.6) is equivalent to the comparison with the modified Euler method (2.1.9). To make (2.2.5) coincide with (2.1.9) with the local error $O(h^3)$, the following relations between the parameters must be satisfied:

$$\begin{cases} c_1 + c_2 = 1, \\ c_2 a = \frac{1}{2}, \\ c_2 b = \frac{1}{2}. \end{cases} \tag{2.2.6}$$

As one can see, this system contains 3 equations for 4 parameters, hence there is a free parameter. Let us put $c_2 = \alpha \neq 0$. Then from (2.2.6) we have

$$c_1 = 1 - \alpha,\ a = \frac{1}{2\alpha},\ b = \frac{1}{2\alpha}. \tag{2.2.7}$$

After substituting these values into (2.2.3) we come to a 1-parametric family of Runge-Kutta schemes:

$$u_{i+1} = u_i + h\left((1-\alpha)f(t_i, u_i) + \alpha f\left(t_i + \frac{h}{2\alpha}, u_i + \frac{h}{2\alpha}f(t_i, u_i)\right)\right). \tag{2.2.8}$$

If $\alpha = 1$, then (2.2.8) becomes the midpoint scheme:

$$u_{i+1} = u_i + hf\left(t_i + \frac{h}{2}, u_i + \frac{h}{2}f(t_i, u_i).\right) \tag{2.2.9}$$

If $\alpha = 0.5$, then (2.2.8) becomes the Heun's scheme:

$$u_{i+1} = u_i + \frac{h}{2}\left(f(t_i, u_i) + f(t_i + h, u_i + hf(t_i, u_i))\right). \tag{2.2.10}$$

Both schemes have the local error proportional to h^3 and the global one proportional to h^2.

2.3 Runge-Kutta of 4th order

Any method of 1-parametric family of methods (2.2.8) can be rewritten as follows: on every step functions η_1 and η_2 are computed:

$$\eta_1^i = f(t_i, u_i),\ \eta_1^i = f\left(t_i + \frac{h}{2\alpha}, u_i + \frac{h}{2\alpha}\eta_1^i\right). \tag{2.3.1}$$

Then, a correction Δu_i is computed:

$$\Delta u_i = h\left((1-\alpha)\eta_1^i + \alpha\eta_2^i\right). \tag{2.3.2}$$

By adding this correction to u_i one obtains an approximation of $u(t_{i+1})$ at $t_{i+1} = t_i + h$:

$$u_{i+1} = u_i + \Delta u_i. \tag{2.3.3}$$

Such methods are called two-stage methods according to the number of computations of the right-hand side of the ODE to be done.

Analogously, a p-stage method can be written as follows:

$$\begin{cases} \eta_1^i = f(t_i, u_i), \\ \eta_k^i = f\left(t_i + a_k h, u_i + h \sum\limits_{j=1}^{k-1} b_{kj} \eta_j^i\right), \\ u_{i+1} = u_i + h \sum\limits_{k=1}^{p} c_k \eta_k^i, \end{cases} \tag{2.3.4}$$

where $k = 2, 3, \ldots, p$. Parameters a_k, b_{kj} and c_k are selected so that the error between u_{i+1} computed by (2.3.4) and Taylor expanded exact solution $u(t_{i+1})$ is equal to $O(h^{p+1})$ without taking into account errors made on previous steps.

The most widely used method of this class is Runge-Kutta 4th order of accuracy (further we will denote it as RK4), which is a 4-stage method:

$$\begin{cases} \eta_1^i = f(t_i, u_i), \\ \eta_2^i = f\left(t_i + \frac{h}{2}, u_i + \frac{h}{2}\eta_1^i\right), \\ \eta_3^i = f\left(t_i + \frac{h}{2}, u_i + \frac{h}{2}\eta_2^i\right), \\ \eta_4^i = f\left(t_i + h, u_i + h\eta_3^i\right), \\ u_{i+1} = u_i + \frac{h}{6}\left(\eta_1^i + 2\eta_2^i + 2\eta_3^i + \eta_4^i\right). \end{cases} \tag{2.3.5}$$

Full derivation of this method can be found, for instance, in [15].

2.4 Adaptive Kutta-Merson scheme

Adaptive stepsize control uses semi-empirical rule based on the Runge principle [31]. Assume that a method of the p-th order of accuracy is used and that the absolute local error should not exceed $\varepsilon > 0$. Then, according to the Runge principle, computations are performed on two grids, the first one $t_i(h) = t_0 + ih$, and the second one $t_j\left(\frac{h}{2}\right) = t_0 + j\frac{h}{2}$. When j is even, points of both girds coincide.

The step on the first grid is done using the stepsize h, and the result is $u_{i+1}(h) \approx u(t_{i+1}(h))$. The step on the second grid is done twice with stepsize $\frac{h}{2}$, i.e. from t_i to $t_i + \frac{h}{2} = t_{2i+1}\left(\frac{h}{2}\right)$ and then from $t_i + \frac{h}{2}$ to $t_i + h = t_{2i+2}\left(\frac{h}{2}\right)$, the result is

$$u_{2i+2}\left(\frac{h}{2}\right) \approx u\left(t_{2i+2}\left(\frac{h}{2}\right)\right) = u(t_{i+1}(h)). \tag{2.4.1}$$

In this case the Richardson correction [15] is computed as follows:

$$R_i\left(\frac{h}{2}\right) = \frac{u_{2i+2}\left(\frac{h}{2}\right) - u_{i+1}(h)}{2^p - 1}. \tag{2.4.2}$$

If $|R_i\left(\frac{h}{2}\right)| < \varepsilon$, then the error between $u(t_{i+1})$ and $u_{2i+2}\left(\frac{h}{2}\right)$ does not exceed ε. If $|R_i\left(\frac{h}{2}\right)| > \varepsilon$, one has to decrease the stepsize h by dividing it by 2. In case $|R_i\left(\frac{h}{2}\right)| << \varepsilon$, one can increase the stepsize h by factor of 2.

As one can see, such approach is quite expensive to use in practice due to large number of computations of the right-hand side, especially for methods with several stages, like RK4.

Another approach uses the fact that there are free parameters in (2.3.4) and it allows to write down several versions of RK methods. The main idea is to derive two methods of the same family and of the same (or close) order(s) of accuracy, such that the same values of right-hand side of ODE are used. Then, by looking at the difference between two approximations computed using theses methods with the same stepsize h, one can estimate the accuracy of one of the methods.

One of such methods is called the Kutta-Merson method. It is a 5-stage method of the 4th order, or, a nested-forms method [31].

On the i-th step the following quantities are computed:

$$\begin{cases} \eta_1^i = f(t_i, u_i), \\ \eta_2^i = f\left(t_i + \frac{h}{3}, u_i + \frac{h}{3}\eta_1^i\right), \\ \eta_3^i = f\left(t_i + \frac{h}{3}, u_i + \frac{h}{6}\eta_1^i + \frac{h}{6}\eta_2^i\right), \\ \eta_4^i = f\left(t_i + \frac{h}{2}, u_i + \frac{h}{8}\eta_1^i + \frac{3h}{8}\eta_3^i\right), \\ \eta_5^i = f\left(t_i + h, u_i + \frac{h}{2}\eta_1^i - \frac{3h}{2}\eta_3^i + 2h\eta_4^i\right), \\ \tilde{u}_{i+1} = u_i + \frac{h}{2}\left(\eta_1^i - 3\eta_3^i + 4\eta_4^i\right), \\ u_{i+1} = u_i + \frac{h}{6}\left(\eta_1^i + 4\eta_4^i + \eta_5^i\right). \end{cases} \tag{2.4.3}$$

Then, $R = 0.2|u_{i+1} - \tilde{u}_{i+1}|$ is computed. If $R > \varepsilon$, then h is to be decreased by factor of 2 and everything starting from η_2^i is to be recomputed. If $R \leq \varepsilon$, then $u(t_{i+1}) \approx u_i$ with accuracy ε. But, if $R \leq \frac{\varepsilon}{64}$, then the next step is to be done with a stepsize increased by factor 2.

2.5 Adams-Bashforth methods

Assume that several approximations $u_j \approx u(t_j),\ j = 0, 1, \ldots, i$ of the exact solution $u(t)$ of IVP (2.0.1) have been computed on a uniform grid $t_j = t_0 + jh$, and one needs to derive a rule how to compute the next approximation $u_{i+1} \approx u(t_{i+1})$. Let us use an integro-interpolation approach: we integrate the IVP (2.0.1) on $[t_i, t_{i+1}]$:

$$u(t_{i+1}) = u_{t_i} + \int_{t_i}^{t_{i+1}} f(t, u(t))dt, \tag{2.5.1}$$

and then approximate $f(y, u(t))$ by an interpolating polynomial $P_k(t)$. Although the values of $f(t, u(t))$ as a function of t are not known, we know a set of discrete values $f(t_j, u_j) = f_j \approx f(t_j, u(t_j))$. Then, one can build a set of finite differences $\Delta^k f_i$ and use the second Newton interpolation formula [31] for backward interpolation:

$$P_k(t) = P_k(t_i + qh) = f_i + q\Delta f_{i-1} + \frac{q(q+1)}{2!}\Delta^2 f_{i-2} + \ldots + \frac{q(q+1)\ldots(q+k-1)}{k!}\Delta^k f_{i-k}, \tag{2.5.2}$$

where $q = \frac{t-t_i}{h}$. The substitution of this polynomial into (2.5.1) gives us a formula for computing the next value $u_{i+1} \approx u(t_{i+1})$:

$$u_{i+1} = u_i + \int_{t_i}^{t_{i+1}} P_k(t)dt. \tag{2.5.3}$$

In order to substitute the interpolation polynomial (2.5.2), which depends on q, in (2.5.1), we do the change of variable $t = t_i + qh$, according to which

$$\int_{t_i}^{t_{i+1}} P_k(t)dt = h \int_{t_i}^{t_{i+1}} P_k(t_i + qh)dq. \tag{2.5.4}$$

Then, formula (2.5.3) can be rewritten as follows:

$$u_{i+1} = u_i + hI_k, \tag{2.5.5}$$

where

$$I_k = \int_0^1 P_k(t_i + qh)dq = \left(f_i q + \frac{q^2}{2}\Delta f_{i-1} + \left(\frac{q^3}{6} + \frac{q^2}{4}\right)\Delta^2 f_{i-2} + \right.$$

$$\left. + \frac{1}{6}\left(\frac{q^4}{4} + q^3 + q^2\right)\Delta^3 f_{i-3} + + \frac{1}{24}\left(\frac{1}{5}q^5 + \frac{3q^4}{2} + \frac{11q^3}{3} + 3q^2\right)\Delta^4 f_{i-4} + \dots \right)\Bigg|_0^1 =$$

$$= f_i + \frac{1}{2}\Delta f_{i-1} + \frac{5}{12}\Delta^2 f_{i-2} + \frac{3}{8}\Delta^3 f_{i-3} + \frac{251}{720}\Delta^4 f_{i-4} + \dots . \tag{2.5.6}$$

The derived relation is a finite-difference formula which defines the Adams-Bashforth extrapolation method:

$$u_{i+1} = u_i + h\left(f_i + \frac{1}{2}\Delta f_{i-1} + \frac{5}{12}\Delta^2 f_{i-2} + \frac{3}{8}\Delta^3 f_{i-3} + \frac{251}{720}\Delta^4 f_{i-4} + \dots \right). \tag{2.5.7}$$

Now let us look at what this formula becomes for first several values k in (2.5.5). Note that by setting $k = 0, 1, 2, \dots$ in (2.5.5) we set the degree of interpolation polynomial as well, and, accordingly, the number of summands in (2.5.6) (or, the same as the number of summands in brackets in (2.5.7)). The finite differences are to be expanded in these formulas.

So, if $k = 0$, then

$$I_0 = f_i, \ \Rightarrow u_{i+1} = u_i + hf(t_i, u_i). \tag{2.5.8}$$

If $k = 1$, then

$$I_1 = f_i + \frac{1}{2}\Delta f_{i-1} = \frac{3}{2}f_i - \frac{1}{2}f_{i-1}, \ \Rightarrow$$

$$u_{i+1} = u_i + \frac{h}{2}\left(3f(x_i, u_i) - f(t_{i-1}, u_{i-1})\right). \tag{2.5.9}$$

In case $k = 2$ we obtain

$$I_2 = I_1 + \frac{5}{12}\Delta^2 f_{i-2} = \frac{23}{12}f_i - \frac{16}{12}f_{i-1} + \frac{5}{12}f_{i-2}, \Rightarrow$$

$$u_{i+1} = u_i + \frac{h}{12}\left(23f(t_i, u_i) - 16f(t_{i-1}, u_{i-1}) + 5f(t_{i-2}, u_{i-2})\right). \tag{2.5.10}$$

Formulas (2.5.8), (2.5.9) and (2.5.10) define the Adams-Bashforth methods of the 1st, the 2nd and the 3rd orders of accuracy respectively. As one can see, (2.5.8) is exactly the forward Euler scheme (2.1.4), thus there is no doubt in its order of accuracy.

In general, one can prove [31] that the Adams-Bashforth method generated by interpolation of the right-hand side of (2.0.1) using a polynomial of degree k has $k + 1$ order of accuracy with respect to stepsize h.

2.6 Predictor-corrector methods

Predictor-corrector methods are based on a combination of explicit and implicit schemes of the same or close orders of accuracy. In the very beginning we restricted ourselves to use only explicit methods because the implicit ones can't be interpreted correctly for coupled systems. In case of predictor-corrector methods, a new value at t_{i+1} is predicted using an explicit method, then this value is corrected using an implicit method. Implicit methods are not covered in this paper, but their derivation can be found, for example, in [15].

We added a single predictor-corrector method to HHG library, namely the one of the 2nd order:

$$\begin{cases} u^P_{i+1} = u_i + \frac{h}{2}\left(3f(t_i, u_i) - f(t_{i-1}, u_{i-1})\right), \\ u_{i+1} = u_i + \frac{h}{2}\left(f(t_{i+1}, u^P_{i+1}) + f(t_i, u_i)\right). \end{cases} \tag{2.6.1}$$

2.7 Starting procedure for multistep methods

As one can see, (2.5.9), (2.5.10) and (2.6.1) are not complete difference formulas, because the way how to compute the very first several values is not given. One might try to set $u_i = u_0,\ i = 1, \ldots, k$, where k is the order of interpolation polynomial. But it turns out [15] that in this case the approximation will be only of order 1. If the scheme is itself of a high order, then such starting procedure will cancel all the advantages of the scheme.

There are 2 ways how to avoid it. The first is to do several small steps with a low-order method like forward Euler, so that the approximation is of the appropriate order of accuracy with respect to original stepsize h. The second idea is to do k steps with a one-step method of the same, or even higher, order of accuracy. In this case the total accuracy of the scheme won't be influenced.

In our implementation, Adams-Bashforth scheme (2.5.10) uses RK4 to obtain very first values, while the predictor-corrector scheme (2.6.1) and 2nd order Adams-Bashforth (2.5.9) use the midpoint rule (2.2.9).

All the numerical results regarding the accuracy of all presented schemes can be found in later sections.

2.8 Stability of linear methods

In this section we briefly describe stability conditions for the linear methods presented above. The detailed derivation can be found in [30] as well as in [31] and [15].

Stability for one-step methods. In general, a numerical method should be consistent with the differential equation it solves, i.e. if the exact solution tends to zero, then the numerical solution should behave the same way.

The stability issues are studied on a model equation of the following form:

$$\frac{du}{dt} = \lambda u,\ \lambda \in \mathbb{C},\ u(0) = u_0. \tag{2.8.1}$$

We'll consider the case when $Re(\lambda) < 0$, i.e. the exact solution tends to zero, because it describes most real-world problems with damping or frictional forces.

In case of a linear system with constant coefficients, it can be disassembled into the set of similar equations, if the system matrix can be diagonalized. Otherwise the stability issue is more tricky, and one has to consider generalized eigenvectors. For non-linear autonomous systems only the local analysis based on linearization around critical points is possible.

By applying any of one-step methods to the equation (2.8.1), we obtain the relation between two stages of numerical solution:

$$U_{k+1} = g(\lambda h)U_k, \tag{2.8.2}$$

where U_k is the numerical solution computed on step k, $g(z)$ is a complex-valued (in general) so-called stability function of a complex argument $z = \lambda h$. If $Re(\lambda) < 0$, then the exact solution goes to zero with time, while the numerical solution demonstrates the same behavior if $|g(z)| < 1$.

So, the region of stability for a one-step method applied to the model problem (2.8.11) is defined as

$$G = \{z \in \mathbb{C} : |g(z)| < 1\}. \tag{2.8.3}$$

Once λh is inside the stability region, the numerical solution will demonstrate the same behavior as the exact one.

Obviously, on a complex plane the boundary of stability region can be found as all the solutions of the following equation:

$$g(z) = e^{i\phi},\ \phi \in [0, 2\pi). \tag{2.8.4}$$

Note that stability concept is not applicable to adaptive methods like Kutta-Merson, since such methods tune the stepsize automatically in order to be consistent with the exact solution.

Here are the stability functions for the methods presented above:

- Forward Euler: $g(z) = 1 + z$;
- Heun and Midpoint $g(z) = 1 + z + \frac{1}{2}z^2$;

- RK4: $g(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4$.

The following figures 1, 2 and 3 show the stability regions for the presented methods on λh plane:

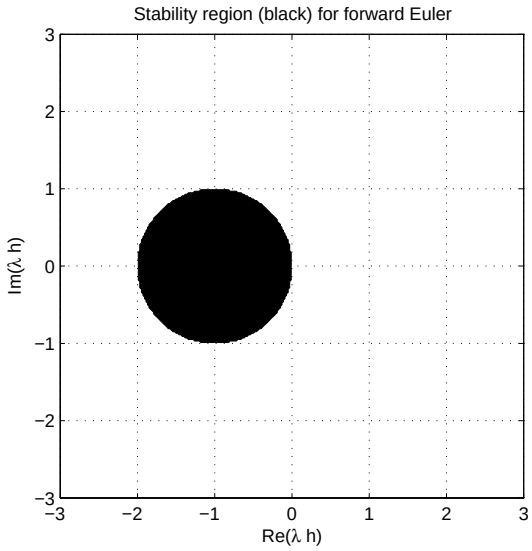

Figure 1: Stability region of the forward Euler method

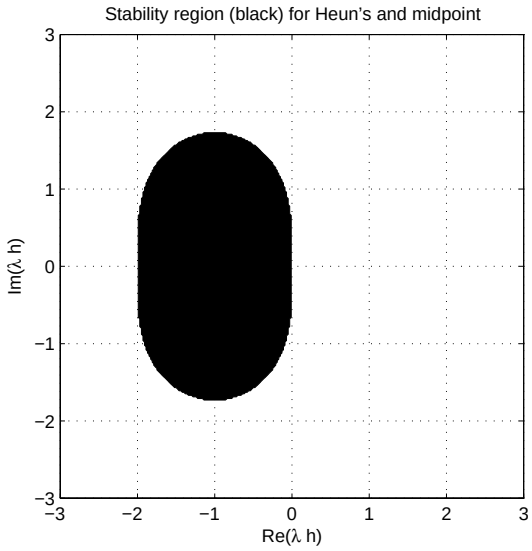

Figure 2: Stability region of the Heun's and the midpoint methods

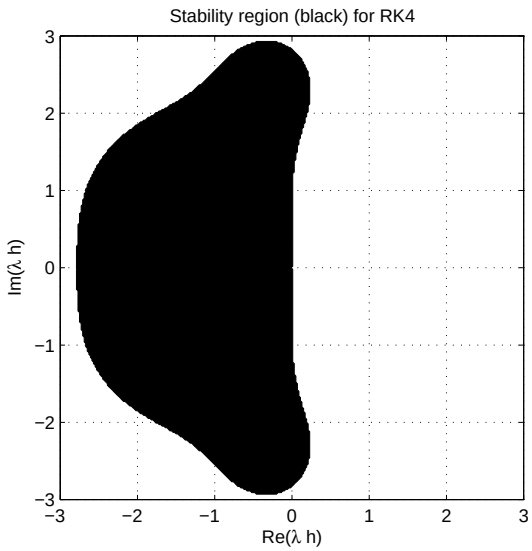

Figure 3: Stability region of the RK4 method

Multistep methods. In order to construct a stability region for a linear m-step method of the general form

$$\sum_{i=0}^{m} a_i u_{k+i} = h \sum_{i=0}^{m} b_i f(t_{k+i}, u_{k+i}) \tag{2.8.5}$$

one builds up two characteristic polynomials [30]:

$$\rho(\mu) = \sum_{i=0}^{m} a_i \mu^i, \ \sigma(\mu) = \sum_{i=0}^{m} b_i \mu^i. \tag{2.8.6}$$

Those polynomials have been computed the for presented multistep methods:

- Adams-Bashforth 2nd order: $\rho(\mu) = -\mu + \mu^2, \ \sigma(\mu) = -\frac{1}{2} + \frac{3}{2}\mu$;
- Adams-Bashforth 3rd order: $\rho(\mu) = -\mu^2 + \mu^3, \ \sigma(\mu) = \frac{5}{12} - \frac{16}{12}\mu + \frac{23}{12}\mu^2$;

Then, if a multistep method of form (2.8.5) is applied to the model problem (2.8.1), one obtains:

$$\sum_{i=0}^{m} a_i u_{k+i} = h \sum_{i=0}^{m} b_i f(t_{k+i}, u_{k+i}) \Leftrightarrow \sum_{i=0}^{m} (a_i - z b_i) u_{k+i} = 0, \tag{2.8.7}$$

where $z = \lambda h$. The last expression is a linear difference equation with respect to u_k. Its general solution is based on powers $\hat{\mu}^k$ where $\hat{\mu}$ are the roots of the polynomial

$$p(\mu) = \sum_{i=0}^{m} (a_i - z b_i) \mu^i. \tag{2.8.8}$$

Thus, the general solution tends to zero if all roots $\hat{\mu}$ lie within a unit circle, and the stability region is

$$G = \{ z \in \mathbb{C} : p(\mu) = \sum_{i=0}^{m} (a_i - z b_i) \mu^i \text{ has only roots } \hat{\mu} \text{ with } |\hat{\mu}| < 1. \} \tag{2.8.9}$$

Here, obviously, the boundary of the stability region is reached if one root crosses unit circle $|\hat{\mu}| = 1 \Leftrightarrow \hat{\mu} = e^{i\phi},\ \phi \in [0, 2\pi)$. Inserting this $\hat{\mu}$ inside $p(\mu)$ we arrive at the expression for all boundary points.

$$\sum_{i=0}^{m}(a_i - zb_i)\mu^i = 0 \Leftrightarrow z(\mu) = \frac{\rho(\mu)}{\sigma(\mu)},\ \mu = e^{i\phi},\ \phi \in [0, 2\pi), \tag{2.8.10}$$

where $\rho(\mu)$ and $\sigma(\mu)$ have been introduced above. So, the boundary of the stability region consists of all points $z(\phi),\ \phi \in [0, 2\pi)$.

The next two figures 4 and 5 demonstrate stability regions of the Adams-Bashforth schemes of the 2nd and the 3rd orders of accuracy respectively.

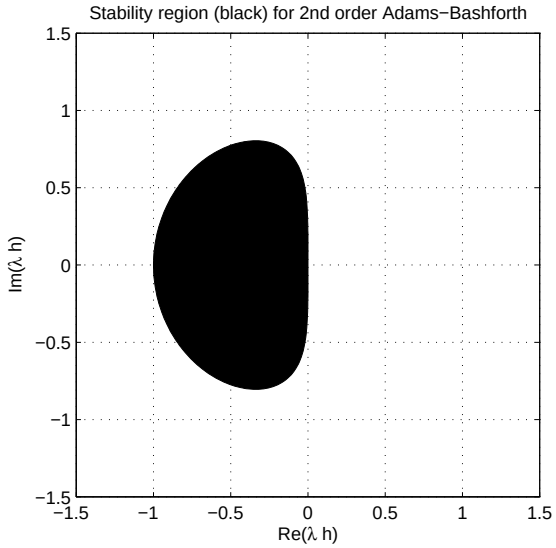

Figure 4: Stability region of the 2nd order Adams-Bashforth method

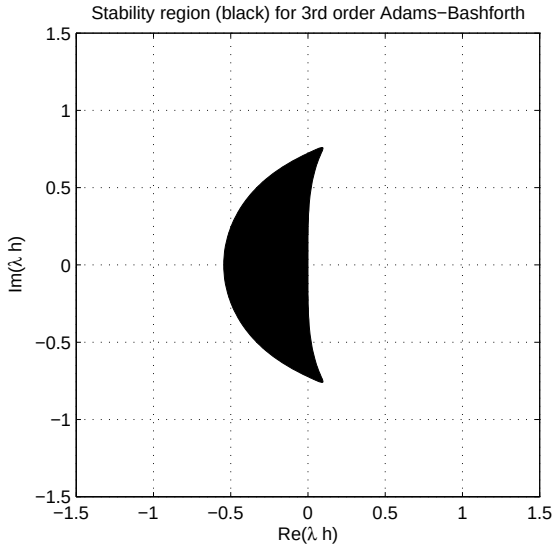

Figure 5: Stability region of the 3rd order Adams-Bashforth method

As it can be seen, the higher the order of the scheme is, the shorter the timestep must be chosen. This is supported by the fact that the forward Euler scheme is also the Adams-Bashforth scheme of the 1st order, and it has the size of stability region of 2 (on the real axis). Also in [30] the computations are done for the scheme of the 4th order, which stability region is approximately 2 times narrower (on the real axis) then the one we got for the 3rd order.

Now let us briefly discuss the stability region of the predictor-corrector scheme (2.6.1). In [13] it has been shown that in case of the predictor-corrector method based on the 2nd order Adams-Bashforth prediction and the 2nd order Adams-Multon correction the relation between z and μ is more complicated than it was in (2.8.10):

$$\mu^2 = \mu + \frac{1}{2}z\left(\mu + \frac{z}{2}(3\mu - 1)\right) + \frac{1}{2}z\mu. \tag{2.8.11}$$

This is a quadratic equation with respect to z, and once roots have been found, they are parametrized as before: $z(\phi) = e^{i\phi}$.

The next figure 6 shows the stability region computed with (2.8.11):

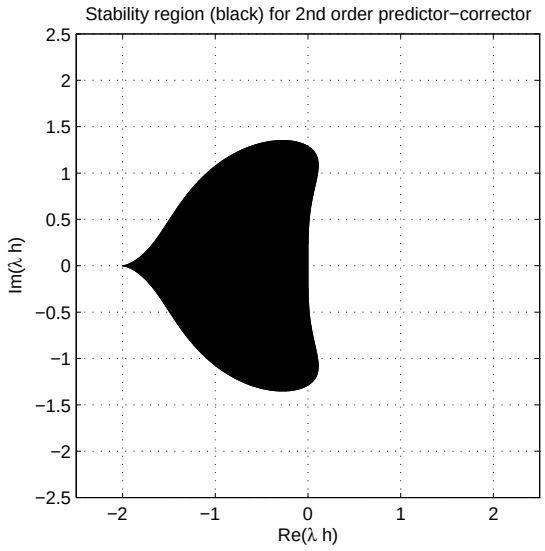

Figure 6: Stability region of the 2nd order predictor-corrector scheme

As one can see, the length of the intersection of the stability region with the horizontal axis is approximately twice large than the one for the Adams-Bashforth scheme of the same order of accuracy (see figure 4), which is an advantage over the standard multistep method.

***A*-stability.** A numerical scheme is called A-stable, if its stability region contains the left half-plane of a complex space. Obviously, none of the presented methods is A-stable, since the stability region is bounded for all the cases which have been considered.

It can be shown that A-stability can be reached only for implicit methods, which are not covered in this paper. However, it turns out [31] that the class of A-stable methods is very narrow: it has been proven that there are no A-stable methods among explicit multistep methods, while the orders of accuracy of A-stable implicit methods can't exceed two.

The concept of A-stability will be recalled in the next chapter in relation with convergence analysis of the parareal algorithm.

3 The parareal algorithm

The parareal algorithm was introduced in 2001 by J.-L. Linos, Y. Maday, G Tutinici in [21]. Later, a modified predictor-corrector method was introduced in [3]. Both versions are equivalent, but the later one is shown to work better on nonlinear equations.

In this chapter we'll consider the latter version of the algorithm, its important properties together with some performance analysis.

3.1 Basic ideas

Although time is considered to be completely sequential and it seems to be impossible to simulate distant future without knowledge of what will happen in the next moment, there exist some techniques which overcome this difficulty. And the parareal algorithm is one of them.

The main idea is similar to the spatial domain decomposition. The global time domain is split into several smaller subdomains and thus the global time-dependent problem is split into the same number of subproblems, and each of them has a smaller evolution interval and it's own initial state. These subproblems can be integrated in parallel by different processors once the initial states are found. The keypoint is that these initial states are computed using a fast, but not so accurate sequential time integrator. Then, each subproblem is integrated using a more accurate and more computationally expensive integrator on its own time-subdomain. Then, a predictor-corrector approach is applied on the numerical solutions of the independent subproblems generated by the accurate integrator together with results of the non-accurate integrator in order to achieve fast propagation of the global solution. It will be described in a more detailed way later in terms of pseudocode.

Let us define the decomposition of time-domain as follows:

$$t_0 = T_0,\ t_{end} = T_N,\ T_0 < T_1 < \ldots < T_n = n\Delta T < T_{n+1} < \ldots < T_N. \tag{3.1.1}$$

A general problem of the following form is considered on the time domain which has been decomposed:

$$\frac{\partial u}{\partial t} + \mathcal{A}u = 0,\ u(T_0) = u_0, t \in [T_0, T_N], \tag{3.1.2}$$

where $\mathcal{A} : \mathbb{R} \times V \to V'$, and V is a Hilbert space.

We also define a numerical operator $\mathcal{F}_{\Delta T}$, which has $u(T_{n-1}) = U_{n-1}$ as an input, and does several small steps with a stepsize $\delta t << \Delta T$ in order to approximate the solution of (3.1.2) at $T_{n-1} + \Delta T = T_n$. This operator is also called a fine integrator further.

So, the numerical solution of the general IVP (3.1.2) can be computed by applying the fine integrator $\mathcal{F}_{\Delta T}$ sequentially for $n = 1, 2, \ldots, N$ starting from the initial state u_0 :

$$\hat{U}_n = \mathcal{F}_{\Delta T}\left(T_{n-1}, \hat{U}_{n-1}\right), \hat{U}_0 = u_0. \tag{3.1.3}$$

Additionally, a coarse integrator $\mathcal{G}_{\Delta T}$ is to be defined. Exactly as $\mathcal{F}_{\Delta T}$, the coarse one takes $u(T_{n-1}) = U_{n-1}$ as an input, and integrates the equation over time ΔT, but using a

bigger stepsize δT. Usually, $\delta t < \delta T < \Delta T$. It has been demonstrated [3] that in order to make the parareal algorithm efficient, coarse integrator must be significantly faster than the fine one. There are several ways how to do that, including making longer steps with the same numerical scheme, using a different numerical scheme or even using a simplified problem. In our case, we'll use a combination of a different numerical scheme with different stepsizes, while the problem is kept the same. So, the coarse integrator works as follows:

$$\tilde{U}_n = \mathcal{G}_{\Delta T}\left(T_{n-1}, \tilde{U}_{n-1}\right), \tilde{U}_0 = u_0, \tag{3.1.4}$$

where $\tilde{U}_n,\ m = 0, 1, \ldots, N$ is a less accurate solution of the same IVP (3.1.2).

Let us now describe how the parareal algorithm works. The first step after initialization is a purely sequential propagation of the solution using the coarse integrator $\mathcal{G}_{\Delta T}$ for $n = 1, 2, \ldots, N$. This provides initial states $\tilde{U}_n^0$ for all the subproblems. Now these problems can be solved simultaneously using the fine integrator: $\hat{U}_n^0 = \mathcal{F}_{\Delta T}\left(\tilde{U}_{n-1}^0\right)$.

After the parallel part is done, the sequential coarse integrator is applied again to compute predictions which are corrected using the information generated by parallel integration with the fine integrator. By this, the new initial states for the next iteration are computed:

$$U_n^k = \mathcal{G}_{\Delta T}\left(U_{n-1}^k\right) + \mathcal{F}_{\Delta T}\left(U_{n-1}^{k-1}\right) - \mathcal{G}_{\Delta T}\left(U_{n-1}^{k-1}\right), \tag{3.1.5}$$

where $k = 1, 2, \ldots, K_{max}$. The iterations over k are repeated until the solution converges to the solution which would be obtained by sequential use of $\mathcal{F}_{\Delta T}$ starting from the same initial state.

The whole algorithm can be written in pseudocode (see algorithm 1).

Algorithm 1 pseudocode for parareal

$U_0^0 \leftarrow \tilde{U}_0^0 \leftarrow y_0$
\\iteration 0
for $n = 1$ **to** N **do**
  $\tilde{U}_n^0 \leftarrow G_{\Delta T}\left(\tilde{U}_{n-1}^0\right)$ \\initial prediction
  $U_n^0 \leftarrow \tilde{U}_n^0$
end for
\\first parareal iteration
for $k = 1$ **to** K_{max} **do**
  $U_0^1 \leftarrow y_0$
  for $n = 1$ **to** N **do**
    $\hat{U}_n^{k-1} \leftarrow F_{\Delta T}\left(\tilde{U}_{n-1}^{k-1}\right)$ \\parallel step
  end for
  for $n = 1$ **to** N **do**
    $\tilde{U}_n^k \leftarrow G_{\Delta T}\left(U_{n-1}^k\right)$ \\predict
    $U_n^k \leftarrow \tilde{U}_n^k + \hat{U}_n^{k-1} - \tilde{U}_n^{k-1}$ \\correct
  end for
  if $|U_n^k - U_n^{k-1}| < \varepsilon\ \forall n$ **then**
    BREAK \\terminate if converged
  end if
end for

As one can see, the main parameter which determines the time consumption is the number k of parareal iterations. Obviously, if there are too many of them, then the algorithm is unlikely to provide any speedup. The algorithm terminates if results of two consequent iterations are close enough. Since this algorithm is proposed for integrating an ODE(s), it might be difficult to interpret what is the $|U_n^k - U_n^{k-1}|\ \forall n$ in case of more or less complicated PDE, like the heat equation. We'll discuss it later.

3.2 Computational complexity

In this section a complexity analysis of parareal algorithm is presented. Obviously, an iterative serial algorithm can't be expected to work faster than the purely sequential fine propagation. The main advantage of parareal is that it has a part which can be done in parallel, thus reducing the computational time.

Let us now denote the complexity of the coarse integrator as $C(\mathcal{G}_{\Delta T})$. By complexity of the integrator we mean the number of arithmetical operations which are to be done in order to get the next value.

Then, it requires $C(\mathcal{G}_{\Delta T})\frac{\Delta T}{\delta T}$ operations by the coarse integrator to propagate solution from T_n to T_{n+1}, while the fine integrator requires $C(\mathcal{F}_{\Delta T})\frac{\Delta T}{\delta t}$ operations respectively.

Now one can write down the number of operations required to perform the algorithm 1 in case of serialization:

$$C_{serialized} = (K+1)NC(\mathcal{G}_{\Delta T})\frac{\Delta T}{\delta T} + KNC(\mathcal{F}_{\Delta T})\frac{\Delta T}{\delta t}, \tag{3.2.1}$$

where K is the number of parareal iterations needed to achieve convergence. At most, $K = K_{max}$.

Assume that one has P computational units available. Then, the second summand in (3.2.1) can be distributed among these P units. In this case, the complexity decreases due to the parallel fine propagation:

$$C_{parallel} = (K+1)NC(\mathcal{G}_{\Delta T})\frac{\Delta T}{\delta T} + K\frac{N}{P}C(\mathcal{F}_{\Delta T})\frac{\Delta T}{\delta t}. \tag{3.2.2}$$

Ideally, one can even have $P = N$, hence the total number of operations becomes

$$C_{parallel,P=N} = (K+1)NC(\mathcal{G}_{\Delta T})\frac{\Delta T}{\delta T} + KC(\mathcal{F}_{\Delta T})\frac{\Delta T}{\delta t}. \tag{3.2.3}$$

Now we need to estimate the complexity of the purely sequential fine propagation. In this case one has to do N steps of complexity $C(\mathcal{F}_{\Delta T})\frac{\Delta T}{\delta t}$:

$$C_{serial} = NC(\mathcal{F}_{\Delta T})\frac{\Delta T}{\delta t}. \tag{3.2.4}$$

Here we assume that the code works within a shared memory model, hence there is no communication involved, that we would have to consider otherwise. Also we assume that all arithmetical operations take the same amount of time t_a.

In order to find an expression for the speedup as a function of P, one divides T_{serial} by $T_{parallel}$:

$$S_P = \frac{T_{serial}}{T_{parallel}} = \frac{t_a \cdot C_{serial}}{t_a \cdot C_{parallel}} = \frac{NC\left(\mathcal{F}_{\Delta T}\right)\frac{\Delta T}{\delta t}}{(K+1)NC\left(\mathcal{G}_{\Delta T}\right)\frac{\Delta T}{\delta T} + K\frac{N}{P}C\left(\mathcal{F}_{\Delta T}\right)\frac{\Delta T}{\delta t}}. \tag{3.2.5}$$

The last expression can be simplified:

$$\begin{aligned} S_P &= \frac{NC\left(\mathcal{F}_{\Delta T}\right)\frac{\Delta T}{\delta t}}{(K+1)NC\left(\mathcal{G}_{\Delta T}\right)\frac{\Delta T}{\delta T} + K\frac{N}{P}C\left(\mathcal{F}_{\Delta T}\right)\frac{\Delta T}{\delta t}} = \\ &= \frac{\left(NC\left(\mathcal{F}_{\Delta T}\right)\frac{\Delta T}{\delta t}\right) / \left(\frac{N}{P}C\left(\mathcal{F}_{\Delta T}\right)\frac{\Delta T}{\delta t}\right)}{\left((K+1)NC\left(\mathcal{G}_{\Delta T}\right)\frac{\Delta T}{\delta T} + K\frac{N}{P}C\left(\mathcal{F}_{\Delta T}\right)\frac{\Delta T}{\delta t}\right) / \left(\frac{N}{P}C\left(\mathcal{F}_{\Delta T}\right)\frac{\Delta T}{\delta t}\right)} = \\ &= \frac{P}{(K+1)P\frac{C(\mathcal{G}_{\Delta T})}{C(\mathcal{F}_{\Delta T})}\frac{\delta t}{\delta T} + K} = \frac{1}{\frac{K}{P} + (K+1)\frac{C(\mathcal{G}_{\Delta T})}{C(\mathcal{F}_{\Delta T})}\frac{\delta t}{\delta T}}. \end{aligned} \tag{3.2.6}$$

As one can see, it is possible to obtain a linear speedup if $\frac{C(\mathcal{G}_{\Delta T})}{C(\mathcal{F}_{\Delta T})}\frac{\delta t}{\delta T} \to 0$. In other words, complexity of the coarse integrator should be almost neglectable with respect to the fine one. In this case if $K = 1$, i.e. parareal converges in a single iteration, and the speedup is perfect. But it can't be reached in practice, because of a number of assumptions made in the beginning. If considering parallel efficiency $E_P = \frac{S_P}{P}$, it is obviously limited by $\frac{1}{K}$.

One should note that there is a strong relation between the number of parareal iterations K needed to converge and the ratio $\frac{C(\mathcal{G}_{\Delta T})}{C(\mathcal{F}_{\Delta T})}\frac{\delta t}{\delta T}$. It can be explained very simply: if the initial propagation is done using a relatively accurate coarse solver, then parareal is likely to take less iterations. On the other hand, if the initial stepping is computed with a very coarse integrator, then it will take more iterations for parareal algorithm to converge. Thus, there is an inverse relation between K and $\frac{C(\mathcal{G}_{\Delta T})}{C(\mathcal{F}_{\Delta T})}\frac{\delta t}{\delta T}$. Taking into account this empirical relation, one can deduce that none of these two quantities can be set arbitrarily small (or equal to 1 for K). In other words, the problem of finding the optimal speedup is an optimization problem which is not covered in this paper. The reader can find more information in [2].

3.3 Stability and convergence

Since parareal combines two integrators, it is essential to find out how the stability of those integrators is related to the stability of the parareal algorithm. Moreover, since the algorithm is iterative, we would like to know how fast it is expected to converge.

In this section the most important results are presented. Most part is presented as in [11], [12] (convergence) and [29], [23] (stability). A very detailed survey on parareal properties can also be found in [26].

Stability. Recall the model ODE IVP (2.8.1) which has been used for studying stability of classical time-integration schemes:

$$\frac{du}{dt} = \lambda u,\ \lambda \in \mathbb{C},\ Re(\lambda) < 0,\ u(0) = u_0. \tag{3.3.1}$$

The idea is the same as before: to build a stability function and to find conditions when the amplitude of this function is less than 1. If before (see (2.8.2)) the stability function was dependent only on the stepsize, now the situation is more complicated, because we have to take into account not only the number of step, but also the number of parareal iteration as well as stability function of both coarse and fine integrators. So, in this case the stability function H looks as follows:

$$U_n^k = H\left(n, k, \Delta T, r(\lambda\delta t), R(\lambda\delta T)\right) U_0^0, \tag{3.3.2}$$

where U_n^k is the numerical solution computed on step n on parareal iteration k, $r(\lambda\delta t)^{\frac{\Delta T}{\delta t}}$ and $R(\lambda\delta T)^{\frac{\Delta T}{\delta T}}$ are the stability functions of fine and coarse integrators respectively. Powers $\frac{\Delta T}{\delta t}$ and $\frac{\Delta T}{\delta T}$ are put due to the fact that the integrators are applied appropriate numbers of times while making a step of length ΔT. Let us denote $\bar{r} = r(\lambda\delta t)^{\frac{\Delta T}{\delta t}}$ and $\bar{R} = R(\lambda\delta T)^{\frac{\Delta T}{\delta T}}$ for convenience.

Now we can express H in terms of $\bar{r}$ and $\bar{R}$ by applying the predictor-corrector scheme (3.1.5) to the model problem (3.3.1):

$$U_n^k = \bar{R}U_{n-1}^k + \bar{r}U_{n-1}^{k-1} - \bar{R}U_{n-1}^{k-1}. \tag{3.3.3}$$

The last expression can be rearranged as follows:

$$U_n^k = \bar{R}U_{n-1}^k + (\bar{r} - \bar{R})U_{n-1}^{k-1}. \tag{3.3.4}$$

One can find that this is a Pascal tree, thus the expression (3.3.3) can be rewritten in terms of initial state U_0^0 and using the binomial coefficients:

$$U_n^k = \left(\sum_{i=0}^{k} \binom{n}{i} (\bar{r} - \bar{R})^i \bar{R}^{n-1}\right) U_0^0 = \left(\sum_{i=0}^{k} \frac{n!}{i!(n-i)!} (\bar{r} - \bar{R})^i \bar{R}^{n-1}\right) U_0^0. \tag{3.3.5}$$

This means that the stability function H of parareal is

$$H\left(n, k, \Delta T, r(\lambda\delta t), R(\lambda\delta T)\right) = \sum_{i=0}^{k} \binom{n}{i} (\bar{r} - \bar{R})^i \bar{R}^{n-1}. \tag{3.3.6}$$

So, one obtains a non-increasing parareal solution for a fixed number k of parareal iterations if H is bounded by some constant

$$\sup_{1\le n\le N} \sup_{1\le k\le N} |H\left(n, k, \Delta T, r(\lambda\delta t), R(\lambda\delta T)\right)| < C. \tag{3.3.7}$$

Note that in case of a linear system with constant coefficient, this has to be fulfilled for all the eigenvalues.

The absolute value of H can be expanded using complex arithmetics and combinatorics:

$$\begin{aligned} |H| = &\left|\sum_{i=0}^{k} \binom{n}{i} (\bar{r} - \bar{R})^i \bar{R}^{n-1}\right| \le \sum_{i=0}^{k} \binom{n}{i} |\bar{r} - \bar{R}|^i |\bar{R}|^{n-1} \le \\ &\le \sum_{i=0}^{n} \binom{n}{i} |\bar{r} - \bar{R}|^i |\bar{R}|^{n-1} = \left(|\bar{r} - \bar{R}| + |\bar{R}|\right)^n \le C. \end{aligned} \tag{3.3.8}$$

In [23] there is a theorem, which establishes the stability condition in case of a fixed number of iterations. It turns out that the stability is guaranteed in this case if

$$|\bar{r}| \leq 1, \ |\bar{R}| \leq 1. \tag{3.3.9}$$

This condition has been shown not to be a sufficient with an increasing number k of parareal iterations.

If one needs to prove that the algorithm does converge for all possible number N of time slots as well as for all possible number $k \in [1, N]$ of parareal iterations, or, in other words, that there is a strong stability, a different approach is to be used. Strong stability is guaranteed if in (3.3.7) and (3.3.8) $C = 1$. For this case real and complex values of λ are to be considered separately. We will state the case of real λ (without proof), while the case of the complex one is much more complicated and thus is omitted here, but can be found in [26].

So, let $\lambda \in \mathbb{R}$, then the algorithm is strongly stable if

$$\frac{\bar{r} - 1}{2} \leq \bar{R} \leq \frac{\bar{r} + 1}{2}, \tag{3.3.10}$$

where $\bar{r} = r(\lambda \delta t)^{\frac{\Delta T}{\delta t}}$ and $\bar{R} = R(\lambda \delta T)^{\frac{\Delta T}{\delta T}}$, assuming both stability functions to be real.

Unfortunately, there is no universal combination of numerical schemes which would guarantee the parareal algorithm to be stable for all possible numbers of temporal subdomains, all possible numbers of iterations and all possible eigenvalues (applicable for systems of equations). In case of pure imaginary eigenvalues it is difficult to guarantee strong stability for some classes of problems. Considering time-dependent PDEs or their systems, numerical solutions of hyperbolic problems or convection-diffusion problems with convection domination computed with parareal may be unstable (see, for instance [8], [10], [25]). Most techniques to overcome this difficulty include the modification of the original algorithm, thus they are beyond the scope of this work.

Convergence. We would also like to know how fast the iterative parareal solution converges to the the solution computed using only the fine integrator $\mathcal{F}_{\Delta T}$ in completely sequential manner. In general, convergence rates are studied on the same model ODE IVP problem (3.3.1).

In [11] and [12] it has been shown that provided $t_{end} < +\infty$, the coarse integrator $\mathcal{G}_{\Delta T}$ has the p-th order of accuracy and the number of parareal iterations k is fixed, then the algorithm converges superlinearly, namely the error is bounded as follows:

$$\max_{1 \leq n \leq N} |u(t_n) - U_n^k| \leq C_k \Delta T^{p(k+1)}. \tag{3.3.11}$$

It can also be shown that the constant C_k grows with k. It the same papers a different case is also considered, when k is large, but ΔT is fixed. A generalization for this case is an unbounded time domain, and for this case the convergence rate is found not to be worse than linear.

Parareal convergence rates are also studied on model PDEs, but authors claim that in order prove the convergence rates for these cases, coarse integrators must be an A-stable one-step methods. As it was noted before (see p. 15), there are no A-stable explicit integrators.

Since we restricted ourselves to use only explicit integrator (due to the equation coupling), the results from the references are not likely to be applicable to our case, but this is definitely a point for further research.

3.4 Algebraic representation

Let us also briefly discuss another way of presenting the parareal algorithm. The initial problem (3.1.2) can be rewritten as a set of IVPs which are to be solved in sequential manner: given a time-domain decomposition

$$T_0 < T_1 < \ldots < T_n = n\Delta T < T_{n+1} < \ldots < T_N,$$

we obtain N IVPs of the following form:

$$\begin{cases} \frac{d\bar{u}_n}{dt} + \mathcal{A}\bar{u}_n = 0, \\ \bar{u}_n(T_n) = U_n,\ T_n \le t \le T_{n+1}. \end{cases} \tag{3.4.1}$$

The solution (or, more precisely, a vector of solutions $[\bar{u}_0, \bar{u}_1, \ldots, \bar{u}_{N-1}]$) of these problem is equivalent to the solution of (3.1.2) if the solutions are computed one-by-one:

$$\bar{u}_0(T_0) = u_0 \Rightarrow \bar{u}_1(T_1) = \bar{u}_0(T_1) = U_1 \Rightarrow \ldots \Rightarrow \bar{u}_{N-1}(T_{N-1}) = U_{N-1}. \tag{3.4.2}$$

Using a fine integrator $\mathcal{F}_{\Delta T}$ we can write this sequence in a matrix-vector form [24]:

$$\begin{pmatrix} I & 0 & \ldots & 0 \\ -\mathcal{F}_{\Delta T} & I & \ldots & 0 \\ 0 & -\mathcal{F}_{\Delta T} & \ldots & 0 \\ 0 & \ldots & -\mathcal{F}_{\Delta T} & I \end{pmatrix} \begin{pmatrix} U_0 \\ U_1 \\ \vdots \\ U_{N-1} \end{pmatrix} = \begin{pmatrix} u_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \tag{3.4.3}$$

Or, using the obvious notation

$$M\Lambda = F. \tag{3.4.4}$$

Actually, to integrate the original IVP (3.1.2) with the fine integrator $\mathcal{F}_{\Delta T}$ means to solve the linear system (3.4.4) by inverting the matrix M. Since this matrix is a triangular one, the complexity of such inversion is $O(N^2)$.

In matrix setting parareal turns out to be an iterative procedure which accelerates computations. Basically with parareal one constructs a sequence of vectors Λ^k which converges to the exact solution of problem (3.4.4).

It has been shown [24] that the prediction-correction procedure (3.1.5) is equivalent to the following iterative process:

$$\Lambda^{k+1} = \Lambda^k + \tilde{M}^{-1}(F - M\Lambda^k), \tag{3.4.5}$$

where

$$\tilde{M} = \begin{pmatrix} I & 0 & \ldots & 0 \\ -\mathcal{G}_{\Delta T} & I & \ldots & 0 \\ 0 & -\mathcal{G}_{\Delta T} & \ldots & 0 \\ 0 & \ldots & -\mathcal{G}_{\Delta T} & I \end{pmatrix} \tag{3.4.6}$$

Obviously, assuming that Λ^k is known, matrix-vector multiplication $M\Lambda^k$ can be done in parallel, while the rest operations are to be done in serial. Note that since $\tilde{M}$ and M are the matrices for integrators, where the coarse one is less accurate that the fine one, $\tilde{M}$ and M are quite close to each other in the sense that $\tilde{M}^{-1}M$ is close to the identity matrix.

4 Implementation details

In this chapter we will give a detailed description of how the concept of time-integrators and parareal have been implemented within the HHG framework.

4.1 Time-dependent problem

Recall the ODE IVP which is to be integrated:

$$\frac{du}{dt} = f(t,u), u(t_0) = u_0, t \in [t_0, t_{end}]. \tag{4.1.1}$$

In principle, each integrator is just an implementation of a particular numerical scheme of integration. It has to be able to compute the right-hand side of (4.1.1). But, on the other hand, the integrator must be independent of the equation.

One approach for this case is to use the features of object-oriented programming paradigm of C++ (which is mainly used in HHG with some addition of Fortran for numerics).

We start from an abstract class of `hhgTimeDependentProblem`, which has a pure virtual member function `computeRHS`. When a user wants to use an integrator, he has to derive a new class from the base class of `hhgTimeDependentProblem`, and inside a derived one to implement the member function `computeRHS`. It is necessary to keep all the auxiliary quantities inside the instance of the derived class and not to pass them as parameters of the function `computeRHS`. The following listing 1 contains the definition of `hhgTimeDependentProblem`.

```
1  class hhgTimeDependentProblem{
2  public:
3
4    hhgTimeDependentProblem(hhgMesh &mesh):mesh_(mesh){}
5
6    virtual void computeRHS(double t, hhgScalarVariable &U_current,
       hhgScalarVariable &result,
7          lvl_t level, hhgPrimitiveGroup groups) = 0;
8
9    virtual ~hhgTimeDependentProblem(){};
10
11 protected:
12   hhgMesh &mesh_;
13 };
```

Listing 1: Class hhgTimeDependentProblem

Here, `mesh_` is a reference to `hhgMesh` object, and `hhgMesh` class does not support a copy constructor up to now, and only one object of this class was created. This deficiency will show up later when doing parallelization of the parareal algorithm. The datatype `hhgScalarVariable` represents the scalar 3D field, and it is convenient to think of it as of a vector in algebraic sense, `lvl_t` shows on which level of multigrid the operations are done, and `hhgPrimitiveGroup` is a bitmap which shows whether computations touch the boundary or not.

4.2 Time integrator

Since we decided to work in an abstract way and to compute the right-hand side in a different object, the instance of time-integrator class should only compute the next state given the current state, current time and the temporal stepsize. A reference to `hhgTimeDependentProblem` is passed in a constructor, by this the same numerical scheme can be easily used for any right-hand side just by defining a new object. But at the same time, since we have studied several time-integration schemes in chapter 1, all of them are to be implemented with a similar interface, but independently of each other. We do it using C++ virtual classes, exactly as it is done for time-dependent problem. Each particular time-integrator class is inherited from an abstract one and redefines the member function `nextStep` according to the chosen numerical scheme. The following listing 2 shows the source code of the abstract time integrator class.

```
1  class hhgTimeIntegrator{
2  public:
3
4    // constructor, need to know properties of mesh
5    hhgTimeIntegrator(hhgMesh &mesh, lvl_t level, hhgTimeDependentProblem &
       timeDepProb):
6    mesh_(mesh),level_(level),timeDepProb_(timeDepProb){}
7
8    virtual void nextStep(double t, double &dt, hhgScalarVariable &U_current,
       hhgScalarVariable &U_next,
9            hhgPrimitiveGroup groups) = 0;
10
11   virtual ~hhgTimeIntegrator(){};
12
13   hhgTimeDependentProblem& getTimeDepProblem(){ return timeDepProb_;}
14
15   // only for multistep methods, not required for one step methods
16   virtual void setStepCounterToOne() = 0;
17
18
19   friend void propagation(hhgTimeIntegrator &integrator, double t_current,
       double dt, hhgScalarVariable &U_current, hhgScalarVariable &U_next, int
       N_steps, lvl_t level, hhgPrimitiveGroup groups);
20
21 protected:
22   hhgMesh &mesh_;
23   lvl_t level_;
24   hhgTimeDependentProblem &timeDepProb_;
25 };
```

Listing 2: Class hhgTimeIntegrator

Here we also pass a reference to the global object of `hhgMesh` class, exactly as in previous case. The reference to the mesh object is important for memory allocation which happens in the constructor. We also restrict the level of the multigrid, on which the integrator works. A member function `nextStep` does all the job for computing the next state. A temporal stepsize `dt` is passed as a reference in order to provide a support of adaptive methods, which tune it

according to prescribed tolerance. Currently, we have implemented a single adaptive method, namely Kutta-Merson one, but, of course, more of them can be added.

Meaning of `setStepCounterToOne` and `propagation` will be shown later in the section describing parareal implementation.

4.3 Parareal in Matlab

For prototyping purpose we decided to do a Matlab version of parareal algorithm in order to get the insight of the algorithm and to find a good way of organizing data. Matlab is typically used for making a prototype of the programm, so called a "proof-of-concept". Then, if the code works and produces sensible results, one can translate it to a more problem-specific language, which is C++ in our case.

Although Matlab is not an environment for parallel programming, it has some useful tools for parallelization of numerical routines, which are available in Matlab Parallel Computing Toolbox. This toolbox has an extensive set of tools for parallel computing such as special data-types, parallelized numerical algorithms and parallel for-loops. It supports computations on multicore systems, GPUs and clusters. A big advantage of this toolbox is that it supports shared memory model out of the box, and this model is essential for the algorithm.

As it was shown in algorithm 1, the only part done in parallel is the fine propagation, and since it is done in a for-loop with known initial states for all iterations and without any data-races, Matlab parallel for-loop can be applied here almost without modifications of the serial code.

The source code of our "proof-of-concept" implementation can be found in appendix (listing 8), here we present the plots which illustrate the work of parareal on the following simple ODE IVP:

$$\frac{du}{dt} = \sin(t)\cos(u),\ u(0) = 1,\ t \in [0, 20]. \tag{4.3.1}$$

We set $\Delta T = 0.5,\ \delta T = 0.5,\ \delta t = 0.05$, thus the coarse integrator always makes a single step to proceed to the next grid point, while the fine one makes 10 steps. The coarse propagator is a simple forward Euler of the 1st order of accuracy, while the fine one implements the midpoint scheme of the 2nd order. So, the following figures 7 and 8 show how the solution produced by parareal converges to the solution produced by sequential application of the fine integrator. The fine solution is plotted at every fine grid point, while the parareal one is only at points of coarse grid.

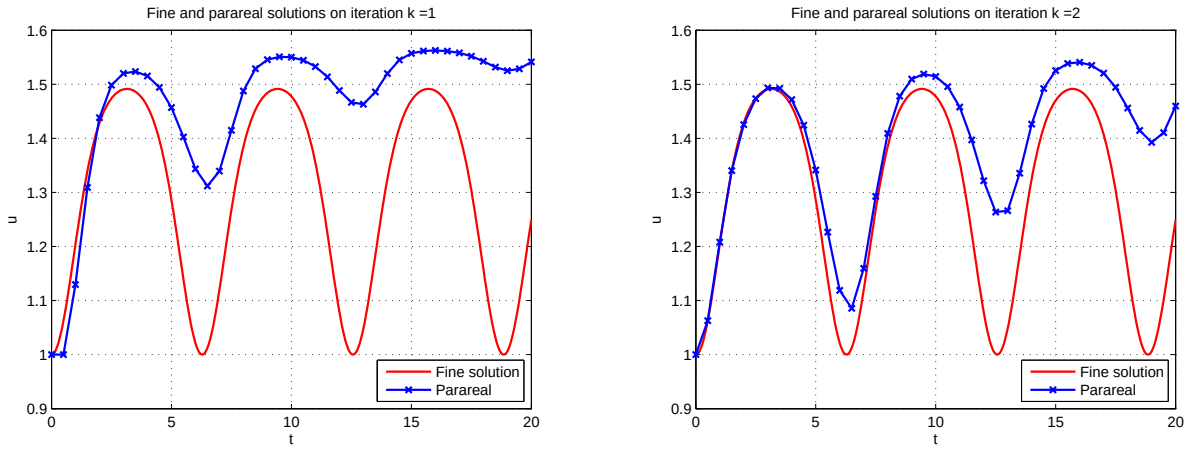

Figure 7: Fine and parareal solutions, iterations 1 and 2

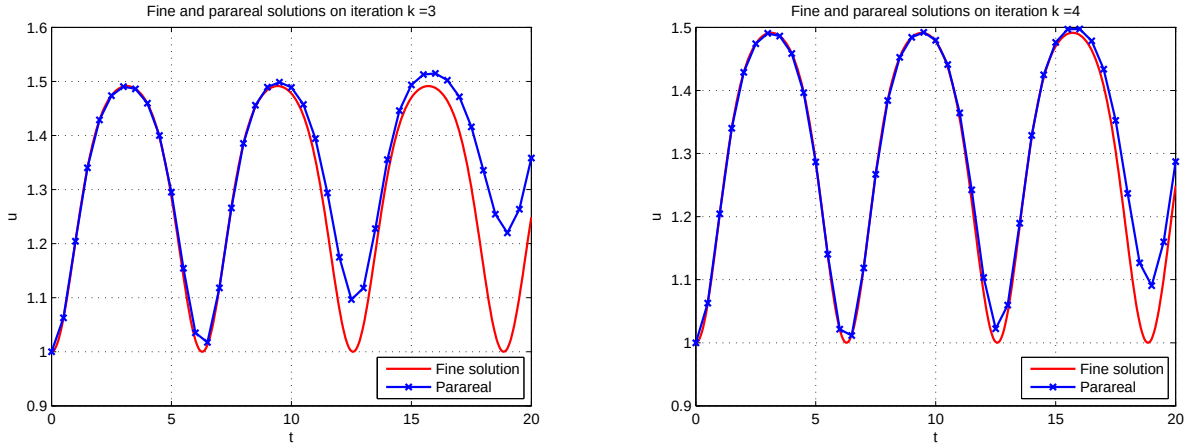

Figure 8: Fine and parareal solutions, iterations 3 and 4

Further iterations are not so much indicative, because the difference between two solutions becomes smaller and smaller. As one can see, the bigger the iteration number is, the closer parareal solution becomes to the fine one, as well as it takes more iterations for later time-points to get the solutions close than it takes for time points close to the beginning of integration interval.

Please note that if one runs the code from the appendix, it might be a good idea to create a Matlab pool of workers manually, otherwise the default configuration will be used, and this configuration might be actually serial.

Moreover, the presented code is not so flexible, since it has simple integrators and the constraint that coarse integrator always makes a single step. Nevertheless, it works and even shows some speedup on a laptop with some combination of parameters. Hence, we can translate the code in C++ and parallelize it within a shared memory model.

4.4 Parareal in HHG

The next steps is to implement the parareal algorithm within the HHG framework. This framework itself is parallelized with MPI, and all the calls for MPI are hidden inside other function calls. It means that from the end user's point of view there is no difference whether to write a serial or a parallel version of the code, they look absolutely identically. The only difference is the compilation process. More information regarding the compilation can be found in readme files of HHG.

There are two main parallelization techniques for parareal implementation. First of all, parareal can be implemented using MPI. In this case one has to rewrite some parts of HHG in order to allocate the processes for parareal. Currently all MPI processes, which are run in parallel, are responsible only for spatial parallelism. Moreover, in case of MPI one would have to send and receive huge chunks of memory representing 3D blocks. This would definitely be a bottleneck for the algorithm. Second of all, one can use shared memory model and openMP. For simplicity, HHG in this case is assumed to be serial, which puts constraints on the problem size and on the computational systems where the code can be run. In principle, both techniques can be combined in a hybrid version, where MPI handles the spatial parallelism, and openMP is responsible the temporal one. But it turns out that the cost of such development is too high. That is why we focus on openMP parallelization.

Let us say a couple of words about openMP. It is an open standard for parallelizing programs in C, C++ and Fortran, which describes a set of compiler directives, library routines and environment variables for making many-thread programs for systems with shared memory.

For us, the most important feature of openMP is the parallelization of for-loops, which can be done by a single line of code. In theory, it is fairly easy, but in practice there are some hidden dangers related to the simultaneous memory access from different threads. It turns out that some HHG objects do not support this feature, especially those which do not completely support OOP paradigm ("rule of three"). At the same time, we would not like to change the code of such objects, mainly because they are organized in a relatively complex hierarchy. And, what is more important, it is logical to assume that, for instance, there is an only object of `hggMesh` class. Thus the only way out is to study what exactly makes the openMP-parallel code crash and try to prevent such situations.

Let us list the requirements for the routines/objects which are to be implemented for parareal:

1. We would like to have a function which does propagation of the solution from T_n to T_{n+1}. Moreover, this function should be able to work with any kind of fixed-stepsize integrators, and should not keep the history of intermediate steps inside. Additionally, this function is to be called from several threads at once, this means that some memory races might occur.

2. Object of class derived from `hhgTimeDependentProblem` class should be available at any thread, but data race is to be prevented.

3. Each thread has to have a local copy of the fine integrator, simply because they start

from different states.

4. Outer loops of parareal should be terminated by break command at the same time if convergence reached.

Although all the time-integrators are written in object-oriented method, we encountered technical problems when trying to organize parareal as a class. Most of them are related to sharing of class members. Because of that, parareal currently is implemented in procedural manner. The next listing 3 shows the code of `propagation` function.

```
void propagation(hhgTimeIntegrator &integrator, double t_current, double dt,
    hhgScalarVariable &U_current, hhgScalarVariable &U_next, int N_steps, lvl_t
     level, hhgPrimitiveGroup groups){

  hhgScalarVariable *tmp;

  #pragma omp critical
  tmp = new hhgScalarVariable("", integrator.mesh_, level, level);

  tmp->initialize(level,0.0);

  hhgLinAlg::Basic::copyVariable(U_current, *tmp, level, groups);

  for(int j = 0; j < N_steps; ++j){
    integrator.nextStep(t_current, dt, *tmp, U_next, groups);
      hhgLinAlg::Basic::copyVariable(U_next, *tmp, level, groups);
    t_current += dt;
  }

  hhgLinAlg::Basic::copyVariable(*tmp, U_next, level, groups);

  // here step counter for multistep methods should be set to 1, for one step
      methods this function does nothing
  integrator.setStepCounterToOne();

  #pragma omp critical
  delete tmp;
}
```

Listing 3: Propagation function

As one can see, a temporary variable is used in the code, and the memory allocation for this variable is thread-safe in some sense. We came to this conclusion experimentally, otherwise, if one deletes these openMP instructions, then a segmentation fault will happen while running. Memory deallocation is also to be done in thread-safe manner. So, this function satisfies the first requirement.

Next, let us show the code (see listing 4) which does the main part of parareal, all the auxiliary stuff is skipped (but can be found in full version of the source code). We also assume that the very first iteration (coarse propagation) is already done just by using the `propagation` function described above.

```
#pragma omp parallel firstprivate(equation, fine_integrator)
{
  // here user can change the type integrator, section is critical because it
     touches the mesh
  #pragma omp critical
  fine_integrator = new hhgAdamsBashforth3TimeIntegrator(mesh, maxLevel,
     equation);

  //iteration 1...K_max-1
  for(int k = 0; k < K_max - 1; ++k){

    // if we are done, just idle
    if(parareal_converged) continue;

    //loop to parallelize
    #pragma omp for
    for(int n = 0; n < N - 1; ++n){
      propagation(*fine_integrator, T[n], dt_fine, *U[k][n], *U_hat[k][n+1],
         static_cast<int>(Dt/dt_fine), maxLevel, groups);
    }

    #pragma omp single
    {
      for(int n = 0; n < N - 1; ++n){
        //prediction
        propagation(G, T[n], dt_coarse, *U[k+1][n], *U_tilde[k+1][n+1],
           static_cast<int>(Dt/dt_coarse), maxLevel, groups);

        //correction
        hhgLinAlg::Basic::addVariables(1.0, *U_tilde[k+1][n+1], 1.0, *U_hat[k][
           n+1], *U[k+1][n+1], maxLevel, groups, Replace);
        hhgLinAlg::Basic::addVariables(1.0, *U[k+1][n+1], -1.0, *U_tilde[k][n
           +1], *U[k+1][n+1], maxLevel, groups, Replace);
      }

      //check if parareal converged
      for(int n = 0; n < N; ++n){
        hhgLinAlg::Basic::addVariables(1.0, *U[k+1][n], -1.0, *U[k][n],
           difference, maxLevel, groups, Replace);
        diff_norm += hhgLinAlg::Basic::maxNorm(difference, maxLevel, groups);
      }
      std::cout << "Parareal iteration " << k + 1 << " finished, diff_norm = "
         << diff_norm << std::endl;

      if(diff_norm < parareal_eps){
        parareal_converged = true;
        K_break = k + 1;
      }
    }
    diff_norm = 0.0;
```

```
43
44    #pragma omp barrier
45    }
46    delete fine_integrator;
47  }
```

Listing 4: Parareal main section

First of all, by defining `eqaution` of class derived from `hhgTimeDependentProblem` as first private, we do copying from the global address space to the thread-local space. For this, an appropriate copy constructor should be defined. But, if the `eqaution` object has any temporary variables, their allocation must be thread safe, as well as the deallocation. In general, we found out that the copy constructor and the destructor of the class must be thread safe in order to avoid segmentation faults. By this, the second requirement is satisfied.

Then, we also declare `fine_integrator` (which is a pointer to `hhgTimeIntegrator`) as first private. Note, that this pointer is defined outside parallel region, but memory is not allocated. So, after copying, all the local copies will point to the same location (up to now unknown) in memory. Inside parallel region we do a thread-safe allocation of local copy of the integrator. Thread safety is necessary because a "dangerous" mesh object is used. There a user can put any class instead of `hhgAdamsBashforth3TimeIntegrator` which is in the listing. This is possible due to polymorphism in C++: a pointer of a base class can point to an object of a derived class, and the particular behavior is resolved while running the code. At this point, the third requirement is also satisfied.

Then next step is to check if the results of two consecutive parareal iterations are close enough to each other. If one integrates an ODE, then it is simple: just compute the maximum norm of difference of two vectors. But, this is a zero-dimensional case (no space dimensions, only time involved). In the example listing 4 above, a 3D heat equation is integrated, thus the problem is more complicated. That is why we compute difference norms at every time point T_n (although the very first one can be excluded), and then sum them up. If one assumes that time is continuous, but not discrete as in our case, as well as space, then such sum, with small modification, can represent an approximation of 4D integral of difference of two functions, which, in turn, is a norm in 4D space.

One more important thing to keep in mind is that break command can't be used in parallel openMP loops. That is why a different approach is implemented. As soon as the results of two consecutive parareal solutions are close enough, a special boolean flag is set to true, and if this flag is true, all threads just continue without performing any actions. Then, before the end of the outer loop, threads are synchronized by calling openMP barrier. By this, the forth requirement is satisfied.

So, the code presented above in listings 3 and 4 satisfies all the requirements we demanded. The only drawback is that it does not take care of MPI calls, which are hidden inside other functions. By this, a normal hybrid parallelization method is broken: usually openMP threads are run inside a MPI-processes, while our code manages the other way around: openMP is wrapped around MPI calls. We tried to combine the parallel version of HHG together with this openMP wrapping, and what we found out is that, first of all, MPI-library must have

full multithreading support. This meas that it must be possible to do MPI call from any thread. Only in this case, the desired hybrid parallelism can be reached. Although the author was able to configure openMPI library to have the full multithreading support, some technical difficulties occur. Moreover, we also did a vectorized implementation of `propagation` function and tried to insert openMP instructions only in regions free of MPI calls, but with high a computational load. In this case, we saw very exotic error messages from MPI. Moreover, the source of this messages has been found. Unfortunately, we were not be able to fix it. Nevertheless, the provided implementation can be run on a shared memory machine, and even on a cluster. The results of numerical tests will be presented in the next chapter.

5 Numerical tests

In this chapter the results of numerical test are presented. First of all, a model problem is considered and an analytical solution for the problem is computed. Then, the orders of accuracy for all the fixed-size implemented integrators are computed and compared with theoretical expectations.

For the parareal algorithm we do the speedup test as well as tests on a model problem with varying parameters in attempt to find the limits of applicability of the method.

5.1 Model problem

As the model problem a 3D heat equation on a unit cube was chosen:

$$\begin{cases} \frac{\partial u}{\partial t} = \Delta u + f, \ x \in \Omega = [0,1]^3, \ t \in [0,1], \\ u(x,y,z,0) = u_0(x,y,z), \\ u\Big|_{\partial\Omega} = 0. \end{cases} \tag{5.1.1}$$

5.2 Analytical solution

In order to find an analytical solution of the problem (5.1.1), we do the following assumptions:

- $u_0(x,y,z) = \sin(\pi x)\sin(\pi y)\sin(\pi z)$,
- $u(x,y,z,t) = \sum\limits_{k=1}^{+\infty}\sum\limits_{l=1}^{+\infty}\sum\limits_{m=1}^{+\infty}\beta_{klm}(t)\sin(k\pi x)\sin(l\pi y)\sin(m\pi z)$,
- $f(x,y,z,t) = F(t)\sin(\pi x)\sin(\pi y)\sin(\pi z)$.

Note that solution $u(x,y,z,t)$ in general has infinite number of time-dependent coefficients $\beta_{klm}(t)$, while the source term can be as well rewritten using infinite sums:

$$f(x,y,z,t) = \sum_{k=1}^{+\infty}\sum_{l=1}^{+\infty}\sum_{m=1}^{+\infty}F_{klm}(t)\sin(k\pi x)\sin(l\pi y)\sin(m\pi z),$$

where $F_{111}(t) = F(t)$, $F_{klm}(t) \equiv 0$, $\forall k,l,m > 1$. At the same time, such representation satisfies boundary conditions automatically.

Now let us compute the terms involved in the equation (5.1.1):

$$\frac{\partial u}{\partial t} = \sum_{k=1}^{+\infty}\sum_{l=1}^{+\infty}\sum_{m=1}^{+\infty}\beta'_{klm}(t)\sin(k\pi x)\sin(l\pi y)\sin(m\pi z), \tag{5.2.1}$$

$$\Delta u = \sum_{k=1}^{+\infty}\sum_{l=1}^{+\infty}\sum_{m=1}^{+\infty}\beta_{klm}(t)(-(\pi k)^2 - (\pi l)^2 - (\pi m)^2)\sin(k\pi x)\sin(l\pi y)\sin(m\pi z). \tag{5.2.2}$$

The equation (5.1.1) will be satisfied if the following relation holds:

$$\beta'_{klm}(t) = -\pi^2(k^2+l^2+m^2)\beta_{klm}(t) + F_{klm}(t),\ \forall k,l,m = 1,\dots,+\infty. \tag{5.2.3}$$

In turn, this equation can be rewritten slightly differently:

$$\beta'_{klm}(t) + \pi^2(k^2+l^2+m^2)\beta_{klm}(t) = F_{klm}(t),\ \forall k,l,m = 1,\dots,+\infty. \tag{5.2.4}$$

One can recognize a first order non-homogeneous linear ODE. The general solution of such equation can be found easily using a technique of integrating factor.

An ODE of the form $\frac{du}{dt} + p(t)u = q(t)$ has a general solution

$$u(t) = \frac{1}{y(t)}\left(\int y(t)q(t)dt + C\right), \tag{5.2.5}$$

where $y(t) = e^{\int p(t)dt}$ is an integrating factor. Actually this means that if one multiplies both sides of equation by $y(t)$, then it will become a separable differential equation, which is easy to integrate.

In our case, the general solution of (5.2.4) is

$$\beta_{klm}(t) = e^{-\int \pi^2(k^2+l^2+m^2)dt}\left(b_{klm} + \int e^{\int \pi^2(k^2+l^2+m^2)dt}F_{klm}(t)dt\right),\ \forall k,l,m = 1,\dots,+\infty, \tag{5.2.6}$$

where b_{klm} are the constants which are to be determined.

In order to find the constants, one uses the initial function $u_0(x,y,z)$:

$$\sum_{k=1}^{+\infty}\sum_{l=1}^{+\infty}\sum_{m=1}^{+\infty}\beta_{klm}(0)\sin(k\pi x)\sin(l\pi y)\sin(m\pi z) = \sin(\pi x)\sin(\pi y)\sin(\pi z). \tag{5.2.7}$$

This is possible if and only if $\beta_{klm}(t) \equiv 0,\ \forall k,l,m > 1$.So, the only time-dependent coefficient which is left is $\beta_{111}(t)$, and it can be expressed as follows:

$$\beta_{111}(t) = e^{-3\pi^2 t}\left(b_{111} + \int e^{3\pi^2 t}F(t)dt\right). \tag{5.2.8}$$

Now let us pick a particular function $F(t) = A\sin(2\omega\pi t)$. Here A is obviously the amplitude and ω is the frequency of oscillations. Such source term allows us to check whether parareal is suitable for integrating problems which involve oscillations, and, what is more important, probably to find some limit cases of applicability.

So, to write down the exact expression for $\beta_{111}(t)$ we have to compute the indefinite integral $\int e^{3\pi^2 t}A\sin(2\omega\pi t)dt$:

$$\int e^{3\pi^2 t}A\sin(2\omega\pi t)dt = A\int e^{3\pi^2 t}\sin(2\omega\pi t)dt = \frac{A}{2\omega\pi}\int e^{3\pi^2 t}\sin(2\omega\pi t)d(2\omega\pi t) =$$

$$= -\frac{A}{2\omega\pi}\int e^{3\pi^2 t}d\cos(2\omega\pi t) = -\frac{A}{2\omega\pi}\left(e^{3\pi^2 t}\cos(2\omega\pi t) - \int \cos(2\omega\pi t)de^{3\pi^2 t}\right) = \tag{5.2.9}$$

$$= -\frac{A}{2\omega\pi}\left(e^{3\pi^2 t}\cos(2\omega\pi t) - 3\pi^2\int\cos(2\omega\pi t)e^{3\pi^2 t}dt\right) =$$

$$= -\frac{A}{2\omega\pi}\left(e^{3\pi^2 t}\cos(2\omega\pi t) - \frac{3\pi^2}{2\omega\pi}\int\cos(2\omega\pi t)e^{3\pi^2 t}d(2\omega\pi t)\right) =$$

$$= -\frac{A}{2\omega\pi}\cos(2\omega\pi t)e^{3\pi^2 t} + \frac{3\pi^2 A}{4\omega^2\pi^2}\int e^{3\pi^2 t}d\sin(2\omega\pi t) =$$

$$= -\frac{A}{2\omega\pi}\cos(2\omega\pi t)e^{3\pi^2 t} + \frac{3\pi^2 A}{4\omega^2\pi^2}\left(\sin(2\omega\pi t)e^{3\pi^2 t} - 3\pi^2\int\sin(2\omega\pi t)e^{3\pi^2 t}dt\right) =$$

$$= -\frac{A}{2\omega\pi}\cos(2\omega\pi t)e^{3\pi^2 t} + \frac{3\pi^2 A}{4\omega^2\pi^2}\sin(2\omega\pi t)e^{3\pi^2 t} - \frac{9\pi^4}{4\omega^2\pi^2}\int A\sin(2\omega\pi t)e^{3\pi^2 t}dt. \quad (5.2.10)$$

At this point one can notice that we have arrived to the starting expression, because the last summand contains the indefinite integral we want to compute. Let us denote $I = \int e^{3\pi^2 t}A\sin(2\omega\pi t)dt$, the the following simple linear equation with respect to I is to be solved:

$$I = -\frac{A}{2\omega\pi}\cos(2\omega\pi t)e^{3\pi^2 t} + \frac{3\pi^2 A}{4\omega^2\pi^2}\sin(2\omega\pi t)e^{3\pi^2 t} - \frac{9\pi^4}{4\omega^2\pi^2}I. \quad (5.2.11)$$

After all the simplifications, we obtain

$$I = \frac{e^{3\pi^2 t}}{4\omega^2 + 9\pi^2}A\left(-\frac{2\omega}{\pi}\cos(2\omega\pi t) + 3\sin(2\omega\pi t)\right). \quad (5.2.12)$$

Now the time-dependent coefficient $\beta_{111}(t)$ can be written down:

$$\beta_{111}(t) = e^{-3\pi^2 t}\left(b_{111} + \frac{e^{3\pi^2 t}}{4\omega^2 + 9\pi^2}A\left(-\frac{2\omega}{\pi}\cos(2\omega\pi t) + 3\sin(2\omega\pi t)\right)\right). \quad (5.2.13)$$

In order to find the value of b_{111}, as we mentioned above, the initial function is used. $\beta_{111}(0)$ has to be equal to 1 to satisfy (5.2.7). This means that

$$b_{111} = 1 + \frac{2A\omega}{\pi(4\omega^2 + 9\pi^2)}. \quad (5.2.14)$$

So, the particular solution of ODE (5.2.6) when $k = l = m = 1$ is

$$\beta_{111}(t) = e^{-3\pi^2 t}\left(1 + \frac{2A\omega}{\pi(4\omega^2 + 9\pi^2)} + \frac{e^{3\pi^2 t}}{4\omega^2 + 9\pi^2}A\left(-\frac{2\omega}{\pi}\cos(2\omega\pi t) + 3\sin(2\omega\pi t)\right)\right), \quad (5.2.15)$$

and the analytical solution of the model problem under the assumptions made above is

$$u = e^{-3\pi^2 t}\left(1 + \frac{2A\omega}{\pi(4\omega^2 + 9\pi^2)} + \frac{e^{3\pi^2 t}}{4\omega^2 + 9\pi^2}A\left(-\frac{2\omega}{\pi}\cos(2\omega\pi t) + 3\sin(2\omega\pi t)\right)\right)\cdot$$

$$\cdot\sin(\pi x)\sin(\pi y)\sin(\pi z). \quad (5.2.16)$$

5.3 Implementation of the model problem in HHG

Since HHG is a finite element framework, we have to rewrite the problem in the corresponding form. To do that, a standard technique is applied to equation (5.1.1). After the discretization, we obtain the following:

$$\frac{dU}{dt} = -M^{-1}SU + F, \tag{5.3.1}$$

where M is a mass matrix, S is a stiffness matrix generated by a set of basis functions, U is a vector of time-dependent coefficients of decomposition of the solution onto a finite element basis. Actually, $U(0)$ has to contain coefficients of decomposition of $u_0(x, y, z)$. By using this technique, a PDE has been transformed into a system of ODEs, which is integrated using any integrator we listed in previous chapter or the parareal algorithm.

One important feature of HHG is that it is a "matrix-free" framework, and thus it is not possible to get direct access to stiffness and mass matrices. That is why one has to implement the right-hand side of the equation using only vector operations. HHG also has an extensive set of operators, which can be used instead of matrix operations.

For instance, computation of mass matrix can be replaced by application of so called lumped mass operator, which computes an approximation of the mass matrix with the only diagonal. By applying this operator to a unit vector, we get the vector containing the diagonal of the lumped mass matrix. Since we want to have the inverse of this matrix, this vector is to be inverted termwise. The next listing 5 shows how to compute the inverse of the mass matrix in HHG.

```
1 // M_diag = M * ones(n) = diag of M
2 hhgLinAlg::Basic::applyOperator(ones, lumped_mass_opr, M_diag, maxLevel,
      WorkingSet, Replace);
3
4 // we need to invert M_diag termwise
5 hhgLinAlg::Basic::copyVariable(M_diag, M_inv_diag, maxLevel, WorkingSet);
6 hhgLinAlg::Basic::invertVariable(M_inv_diag,maxLevel, WorkingSet);
```

Listing 5: Definition of mass matrix in HHG

One may also ask how do we compute the stiffness matrix. This can be done by applying the diffusion operator to a vector of coefficients. Note that HHG defines operators, which are applied to the problem of the form $\frac{\partial u}{\partial t} + Lu = f$, which, in turn, is slightly different from what we use. Hence, in HHG $Lu = -\Delta u$.

$$\int_\Omega L\left(\sum_i U_i\phi_i\right) v_j d\Omega = -\sum_i U_i \int_\Omega \Delta\phi_i v_j d\Omega =$$

$$= -\sum_i U_i \int_\Omega -\nabla\phi_i \nabla v_j d\Omega = SU. \tag{5.3.2}$$

Here ϕ_i is a basis function, v_j is a test function from the appropriate test space. We also used the rules of multidimensional integration and the fact that we have homogenous boundary conditions. Since we have Laplacian at the right-hand side, we compute $-Lu = -SU$.

The next step is to implement the routine (listing 6) which will compute the right-hand of the ODE system (5.3.1).

```
1  void computeRHS(double t, hhgScalarVariable &U_current,  hhgScalarVariable &
       result, lvl_t level, hhgPrimitiveGroup groups){
2
3    // step 1: tmp = S * U_current, where S is a stiffness matrix
4    hhgLinAlg::Basic::applyOperator(U_current, diffusion_opr_, *tmp, level,
         groups, Replace);
5
6    // step 2: tmp = -M_inv * result = - M_inv * S * U_current
7      hhgLinAlg::Basic::addVariables(-1.0, M_inv_diag_, 1.0, *tmp, *tmp, level,
           groups, Replace, Multiply);
8
9    // step 3: result = tmp + F = - M_inv * S * U_current + F
10   hhgLinAlg::Basic::addVariables(1.0, *tmp, 1.0, F_, result, level, groups,
         Replace);
11 }
```

Listing 6: ComputeRHS routine for the heat equation

Of course, all the necessary objects like operators or forcing term have to be defined in the constructor. Moreover, the presented implementation is for the constant forcing term. Anyway, it demonstrates the main idea.

In case of our particular problem (5.1.1) with analytical solution (5.2.15), a trick can be done: computation of the Laplacian is nothing else but the multiplication by $-3\pi^2$. This helps us to get rid of unnecessary errors which come from spatial discretization, because we are mostly interested in temporal errors, and we would like to measure them somehow. The forcing term has a time-dependent coefficient, which is put in front of the sine product as well. So, the right-hand side for the particular model problem can be written as follows (listing 7).

```
1  void computeRHS(double t, hhgScalarVariable &U_current,  hhgScalarVariable &
       result, lvl_t level, hhgPrimitiveGroup groups){
2
3     // a trick - change numerical approximation of laplacian to analytic
          expression, works for the only problem!
4    hhgLinAlg::Basic::addVariables(-3.0 * M_PI * M_PI, U_current, 0.0, U_current,
         *tmp, level, groups, Replace);
5
6    hhgLinAlg::Basic::addVariables(1.0, *tmp, A_ * sin(2 * w_ * M_PI * t), F_,
         result, level, groups, Replace);
7  }
```

Listing 7: ComputeRHS routine for the model problem

5.4 Time-integrators accuracy tests

First of all, let us describe the approach to measuring the accuracy of a time integrator. Let us fix the end time of integration, and denote by $\tilde{u}_h$ a numerical solution computed using the integrator with stepsize h, while u is the exact (in our case analytical) solution. Then, as we

mentioned before, the numerical method is of order p (globally) if $\|\tilde{u}_h - u\| \leq Ch^p$. Here and further $\|u\| = \sqrt{(\#gridpoints)^{-1}u^T u}$ (scaled Euclidean norm). We may also assume that the error $\tilde{u}_h - u$ depends smoothly on h, then

$$\tilde{u}_h - u = Ch^p + O(h^{p+1}). \tag{5.4.1}$$

So, if the exact solution is known, and that is our case, then it is quite obvious how to determine the accuracy of the integrator. It can be done by checking a sequence

$$\log \|\tilde{u}_h - u\| = \log |C| + p \log h + O(h), \tag{5.4.2}$$

for $h_1, h_2, \ldots$, and fit it to a linear function of $\log h$ to approximate p. This can be done in a "quick-and-dirty" way by plotting $\|\tilde{u}_h - u\|$ as a function of h on loglog plot. In this case p will be the slope of the line. The standard way how to get the exact value for p is to compute ratios between errors $\tilde{u}_h - u$ and $\tilde{u}_{h/2} - u$:

$$\frac{\tilde{u}_h - u}{\tilde{u}_{h/2} - u} = \frac{Ch^p + O(h^{p+1})}{C(h/2)^p + O((h/2))^{p+1}} = 2^p + O(h). \tag{5.4.3}$$

Thus,

$$\log_2 \frac{\|\tilde{u}_h - u\|}{\|\tilde{u}_{h/2} - u\|} = p + O(h). \tag{5.4.4}$$

We tested the integrators using the following setting: $t_{start} = 0$, $t_{end} = 0.01$, in total 6 stages, $dt_1 = 0.001$, on each stage excluding the initial the dt is halved, so the final one is $dt_6 = 3.125e-05$. All the computations are done on a unit cube, with a meshfile "12el_tet.ugm", on the 4th level of grid. As the exact solution we use the one from (5.2.15), where $A = 1$, $\omega = 1$.

The computed results are presented in the following tables 1–7.

dt	**Error norm**	**Ratio**	**p**
0.001	0.00129316319585	2.0179353369	1.0128799452
0.0005	0.00064083480387	2.0088696445	1.00638395066
0.00025	0.000319002681744	2.00441074473	1.00317817653
0.000125	0.000159150355077	2.00219940541	1.00158566392
6.25e-05	7.94877646288e-05	2.00109821745	1.00079197902
3.125e-05	3.97220705788e-05		

Table 1: Accuracy of the Forward Euler integrator

dt	**Error norm**	**Ratio**	**p**
0.001	1.28206418322e-05	4.04477059591	2.01605788077
0.0005	3.16968330545e-06	4.02229663115	2.00801947958
0.00025	7.88028232654e-07	4.01112595296	2.00400726879
0.000125	1.964606053e-07	4.00555736671	2.00200300524
6.25e-05	4.90470082723e-08	4.00277726957	2.00100134067
3.125e-05	1.22532444274e-08		

Table 2: Accuracy of the midpoint integrator

dt	Error norm	Ratio	p
0.001	1.28200893882e-05	4.04477131184	2.01605813613
0.0005	3.16954616215e-06	4.02229697082	2.00801960141
0.00025	7.87994070339e-07	4.01112611701	2.00400732779
0.000125	1.96452080376e-07	4.005557436	2.0020030302
6.25e-05	4.90448791499e-08	4.00277731283	2.00100135627
3.125e-05	1.22527123837e-08		

Table 3: Accuracy of the Heun's integrator

dt	Error norm	Ratio	p
0.001	5.633038521e-10	16.198723187	4.01780819662
0.0005	3.47745835024e-11	16.098768488	4.0088784252
0.00025	2.16007724618e-12	16.049400637	4.00444751596
0.000125	1.34589278132e-13	16.0004788972	4.00004318077
6.25e-05	8.41157811565e-15	15.7076890971	3.97339904297
3.125e-05	5.35507041403e-16		

Table 4: Accuracy of the RK4 integrator

Please note that in case of Runge-Kutta 4th order (table 4) the error computed with the smallest stepsize is very close to machine precision, and that is why the values are not quite correct for the last line of the table.

dt	Error norm	Ratio	p
0.001	3.00108927744e-05	3.91247076819	1.96807997307
0.0005	7.67057303492e-06	3.95696945132	1.98439592743
0.00025	1.93849690509e-06	3.97867935851	1.99228963702
0.000125	4.8722119337e-07	3.98938964656	1.99616803937
6.25e-05	1.22129256988e-07	3.99470747261	1.99808986028
3.125e-05	3.05727660474e-08		

Table 5: Accuracy of the 2nd order Adams-Bashforth integrator

dt	Error norm	Ratio	p
0.001	6.89616815492e-07	7.22516226317	2.85302998812
0.0005	9.54465505927e-08	7.63943337766	2.93346563649
0.00025	1.24939306195e-08	7.82587081793	2.96825129579
0.000125	1.59649078169e-09	7.91441190966	2.98448215307
6.25e-05	2.01719445476e-10	7.95755943575	2.99232602752
3.125e-05	2.53494110983e-11		

Table 6: Accuracy of the 3rd order Adams-Bashforth integrator

dt	Error norm	Ratio	p
0.001	4.87738167442e-06	3.48956266338	1.80304623904
0.0005	1.39770571413e-06	3.76839178717	1.91394896496
0.00025	3.70902441432e-07	3.88948595387	1.95957949691
0.000125	9.53602727534e-08	3.94599784097	1.9803901667
6.25e-05	2.4166326642e-08	3.97330464079	1.99033941183
3.125e-05	6.08217310948e-09		

Table 7: Accuracy of the 2nd order predictor-corrector integrator

As one can see, asymptotically all the integrators have the right order of accuracy. Thus we can conclude that they have been implemented correctly and can be used for integrating bigger and more complex problems. An interesting observation is that somehow one-step methods tend to the asymptote from above, while multistep methods do it from below.

Now let us present the loglog plots of the error versus the stepsize, where the slope of the line will be actually the order of accuracy of the corresponding integrator.

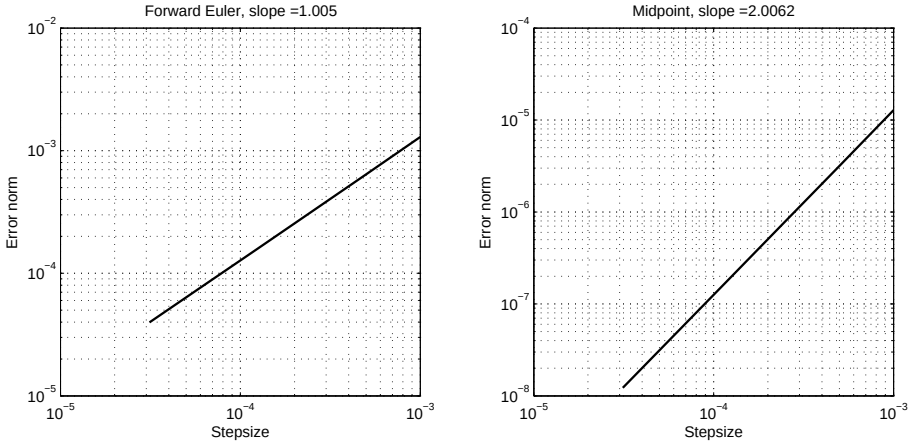

Figure 9: Error norm vs stepsize at t_{end} for Forward Euler and midpoint

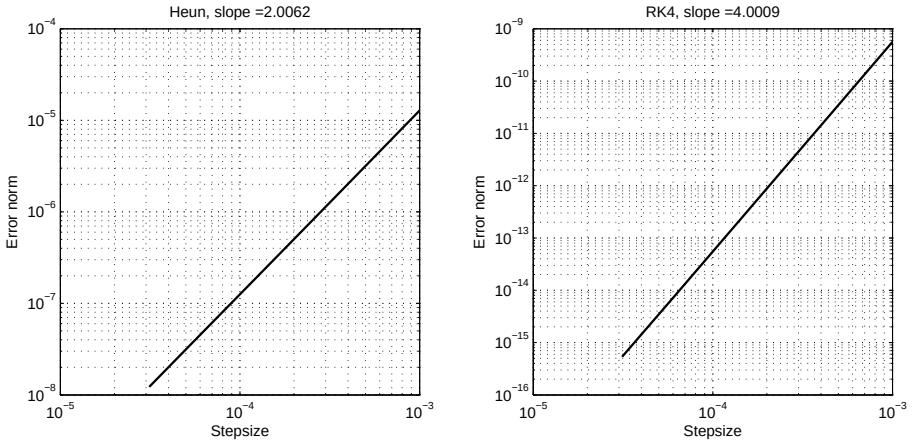

Figure 10: Error norm vs stepsize at t_{end} for Heun and RK4

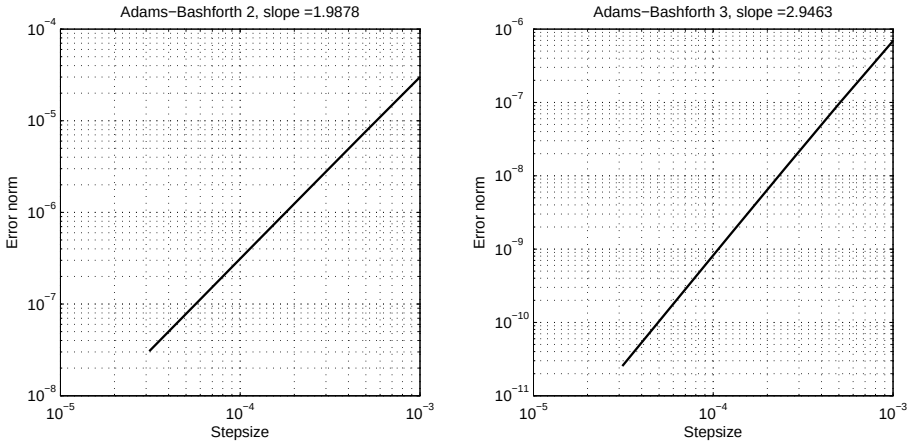

Figure 11: Error norm vs stepsize at t_{end} for AB2 and AB3

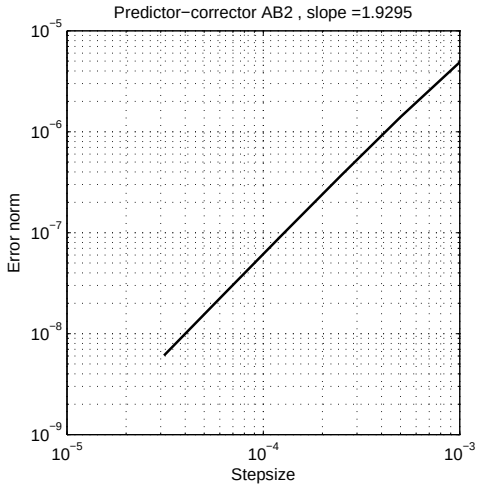

Figure 12: Error norm vs stepsize at t_{end} for PC2

As we can see, indeed the figures 9–12 show that the integrators have the right orders of accuracy.

We also did tests for a problem without known analytical solution (3D heat equation with a constant source term). In this case a solution computed with very small stepsize is used as a reference, but the ratios a computed differently. Moreover, while doing experiments for such problem without analytical solution, we did not exclude the influence of spatial errors, just by using the code from listing 6. Nevertheless, all the integrators showed the desired accuracy, thus we conclude that they are implemented correctly. We do not put the tables and graphs here, because they basically show exactly the same behavior of errors, thus they do not contain any new relevant information.

5.5 Speedup tests for parareal

Since parareal is indeed a parallel algorithm, we need to show that it does provide some speedup. For this purposes, we will use the cluster of System Simulation Chair of FAU Erlangen-Nuremberg. This cluster has 8 computational nodes each of those consists of 4 CPU Intel(R) Xeon(R) CPU E7-4830 with 8 cores and has 256 GB of RAM.

As we stated before, we were not able to use spatial and temporal parallelism at the same time because of some technical problems of HHG. That is why we'll utilize only temporal parallelism. Since parareal has been implemented using openMP, thus the test are to be done on a shared memory machine, or, equivalently, within a single node, otherwise the interconnect will take too much time and there probably won't be any speedup at all. So, we'll use one thread per core, and he number of cores is up to 32 (to stay within 1 node).

For testing the same model problem is used, although the analytical solution is not needed here, such simple form of function allows to exclude the unnecessary computations of Laplacian by replacing it by multiplication by $-3\pi^2$ (see above). In all the test cases, forward Euler integrator is the coarse one, while the fine one is Heun's integrator of 2nd order. We set parameters as follows: $t_{start} = 0$, $t_{end} = 1$, $\Delta T = 0.01$ (100 sub-intervals in total), $\delta T = 0.0002$, the ratio $\frac{\delta T}{\delta t}$ between the coarse and the fine stepsizes is denoted by R and we do test for $R = 10, 20, 40, 80$. One more important parameter is ε, the tolerance, which is used to determine if the two parareal iterations are close enough. We set $\varepsilon = 0.0001$, and with this value parareal does 2 iterations on mesh from meshfile "12el_tet.ugm", on the 4th level of its grid.

The results of numerical experiments on a HPC cluster are presented in the following tables 8–11. By efficiency we denote the fraction $E_P = \frac{S_P}{\#threads}$. We also state the error between so-called serial solution and parareal one at the final time.

1. $R = 10$, $T_{serial} = 18.481428688$, $\|u_{parareal}(t_{end}) - u_{serial}(t_{end})\| = 4.40124699692e-11$.

# threads	Parareal time	Speedup	Efficiency
1	40.127931765	0.460562702	0.460562702
2	21.846505402	0.845967277	0.422983638
4	12.475803109	1.481381881	0.370345470
8	8.0680180329	2.290702451	0.286337806
16	5.963918965	3.098873206	0.193679575
32	5.091704462	3.629713551	0.113428548

Table 8: Parareal computational time, speedup and efficiency, $R = 10$

As one can see, although parareal shows some speedup, the efficiency degrades very quickly. This is the major drawback of the algorithm. Let us check how these quantities behave for other values of R.

2. $R = 20,\ T_{serial} = 36.846165626,\ \|u_{parareal}(t_{end}) - u_{serial}(t_{end})\| = 7.09326423678e-12.$

# threads	Parareal time	Speedup	Efficiency
1	77.178353422	0.477415803	0.477415803
2	40.5900477711	0.907763544	0.453881772
4	21.923737832	1.680651625	0.420162906
8	13.115925238	2.809269262	0.351158657
16	8.99460831599	4.096472501	0.256029531
32	6.78620464803	5.429568888	0.169674027

Table 9: Parareal computational time, speedup and efficiency, $R = 20$

3. $R = 40,\ T_{serial} = 73.484222967,\ \|u_{parareal}(t_{end}) - u_{serial}(t_{end})\| = 2.59190980396e-12.$

# threads	Parareal time	Speedup	Efficiency
1	150.891552483	0.487000244	0.487000244
2	77.6575012029	0.946260462	0.473130231
4	40.502308271	1.814321852	0.453580463
8	23.345792586	3.147643100	0.393455387
16	14.863774557	4.943846711	0.308990419
32	10.570647367	6.951723997	0.217241374

Table 10: Parareal computational time, speedup and efficiency, $R = 40$

4. $R = 80,\ T_{serial} = 146.649180216,\ \|u_{parareal}(t_{end}) - u_{serial}(t_{end})\| = 5.67994044537e-12.$

# threads	Parareal time	Speedup	Efficiency
1	297.496455302	0.492944294	0.492944294
2	153.054585356	0.958149537	0.479074768
4	78.300121752	1.872911266	0.468227816
8	43.40629783	3.378523107	0.422315388
16	26.351417798	5.565134344	0.347820896
32	18.265337989	8.028823792	0.250900743

Table 11: Parareal computational time, speedup and efficiency, $R = 80$

From the presented tables 8–11 one can conclude that the greater the ratio between the stepsizes becomes, the greater the speedup becomes and the more efficiently the resources of the HPC system are used. This is exactly what was predicted theoretically (see 3.2.6). Moreover, parareal and "serial" solutions become closer to each other. Also, as it was stated in 3.2, the efficiency is limited by $1/K$, and this is exactly what we got from the numerical tests.

Let us now present the plots (figures 13 and 14) which illustrate the behavior of the speedup and the efficiency.

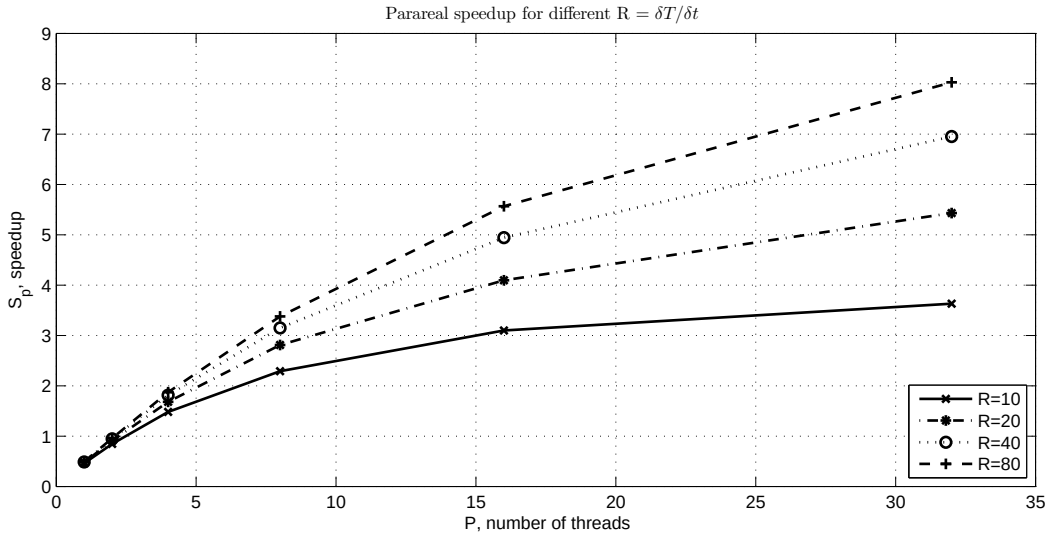

Figure 13: Parareal speedup for different values of $R = \delta T/\delta t$

Here (figure 13), obviously, the greater R is, the more and more linear the speedup becomes. This fact absolutely agrees with what was stated in section 3.2.

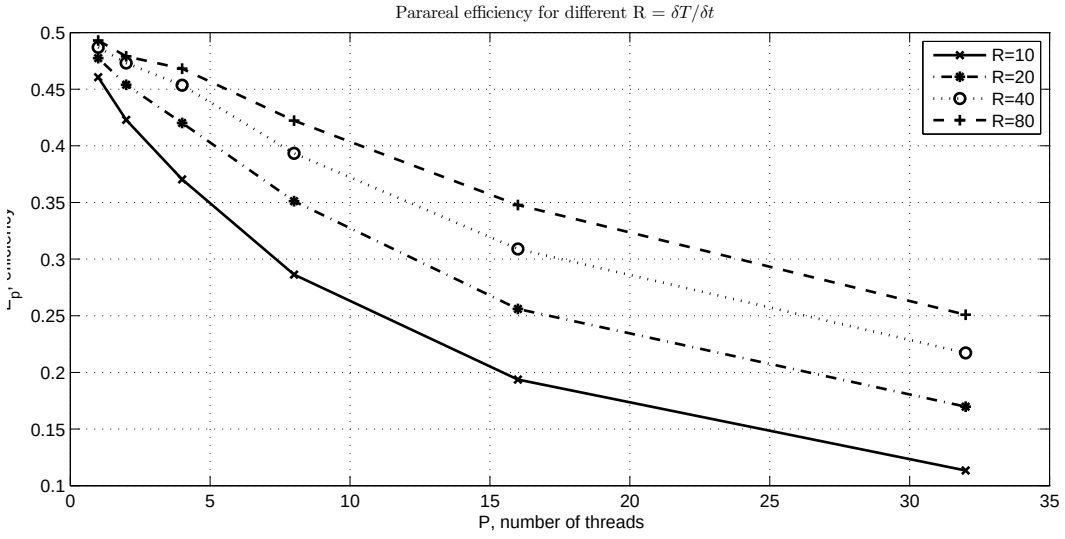

Figure 14: Parareal efficiency for different values of $R = \delta T/\delta t$

As it can be seen in figure 14, the degradation of efficiency does not seem to be a linear process with respect to the number of threads used for computations. What we can predict is that for larger values of R the efficiency degradation will be less significant.

5.6 Parareal applicability tests

We would like to find some kind of limit cases where parareal does not work good enough for the model problem. But, taking into account a large number of parameters involved, the search space is too highly-dimensional, thus we fix most of parameters and vary only one or two.

Let us do as follows: we keep the model problem the same (5.1.1), the ratio $R = \delta T/\delta t = 10$ is also kept the same. Coarse propagator is the forward Euler, the fine one - the Heun's. The parameter ε, which regulates the number of parareal iterations, is set to 0.0001, as before.

What we can do, since the analytical solution (5.2.15) has been computed, is to measure the error between solution generated by parareal and the analytical one at $t = t_{end} = 1$ for different values of amplitude A and frequency ω.

First, we fix $\Delta T = 0.1$, thus we have 10 sub-intervals in total. The following figure 15 shows the behavior of the error norm at final time as function of A and ω.

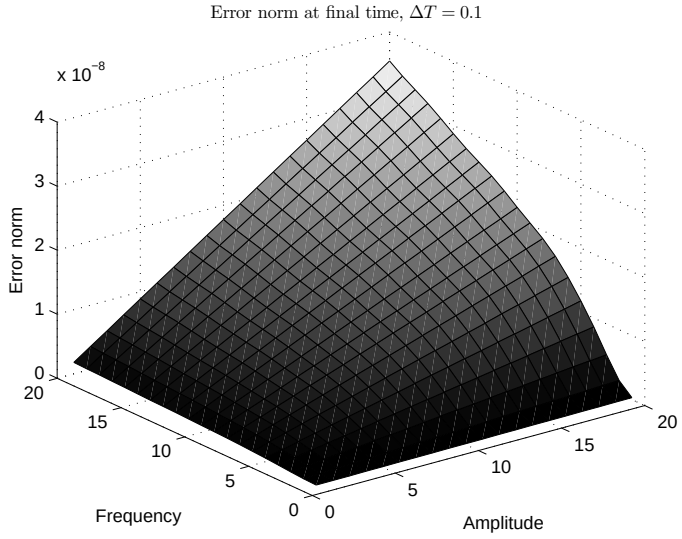

Figure 15: Error norm at final time, $\Delta T = 0.1$

It seems that growth both of the amplitude and the frequency of the source cause the growth of the error, although the error is not too high itself. Let us plot the cases for $\Delta T = 0.05,\ 0.02,\ 0.01$, or, equivalently, $N = 20,\ 50,\ 100$ subintervals (figures 16, 17 and 18 respectively). Other parameters are kept the same.

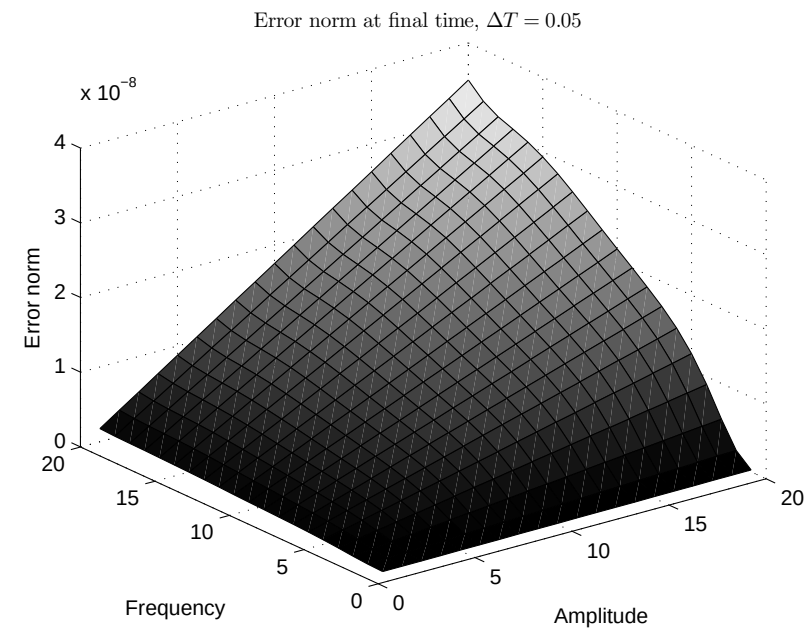

Figure 16: Error norm at final time, $\Delta T = 0.05$

In this case (figure 16) the behavior is basically the same, but one can notice a kind of sinusoidal oscillation at $A = 20$, which was not presented in the previous plot. For other cases the plots show even more interesting results.

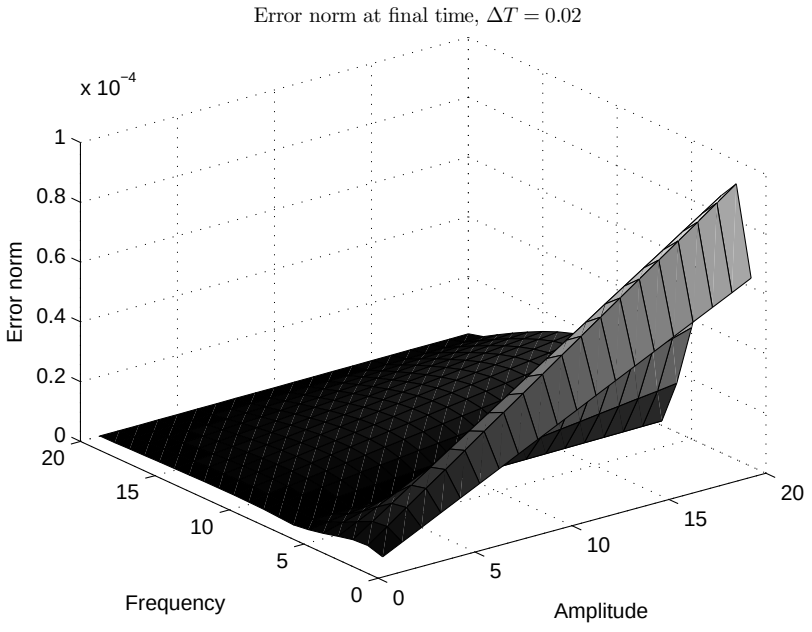

Figure 17: Error norm at final time, $\Delta T = 0.02$

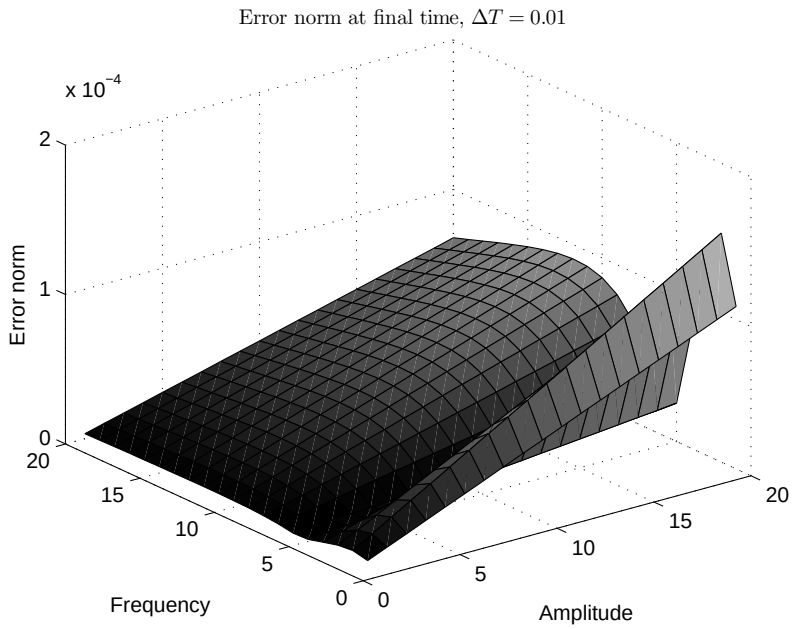

Figure 18: Error norm at final time, $\Delta T = 0.01$

As we can see in figures 17 and 18, there is a fast jump of error norm when ω grows, and then there is a decay and a kind of stabilization. At first glance, the error norm always grows when A grows, but the relation between the frequency ω and the error norm is more complicated. Let us plot all 4 cases at semilog scale at $A = 20$:

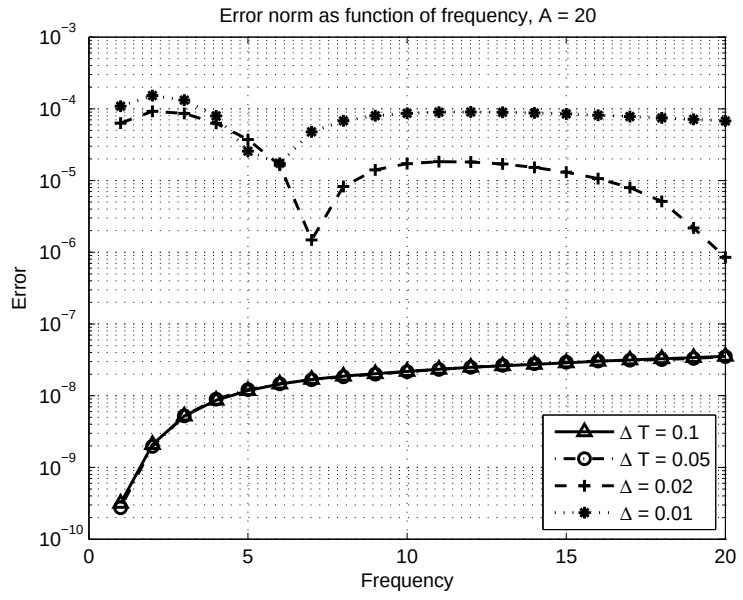

Figure 19: Error norm at final time as function of frequency, $A = 20$

As one can see in figure 19, the error norm seems to be bounded, but the studied range from 1 to 20 for each of parameters is not too broad and might not cover more interesting

cases. Note that on the last figure first two curves almost coincide, but both of them are presented.

Let us do the same computations, but now let A and ω be the powers of 2 from 1 to 2048 (figures 20 and 21):

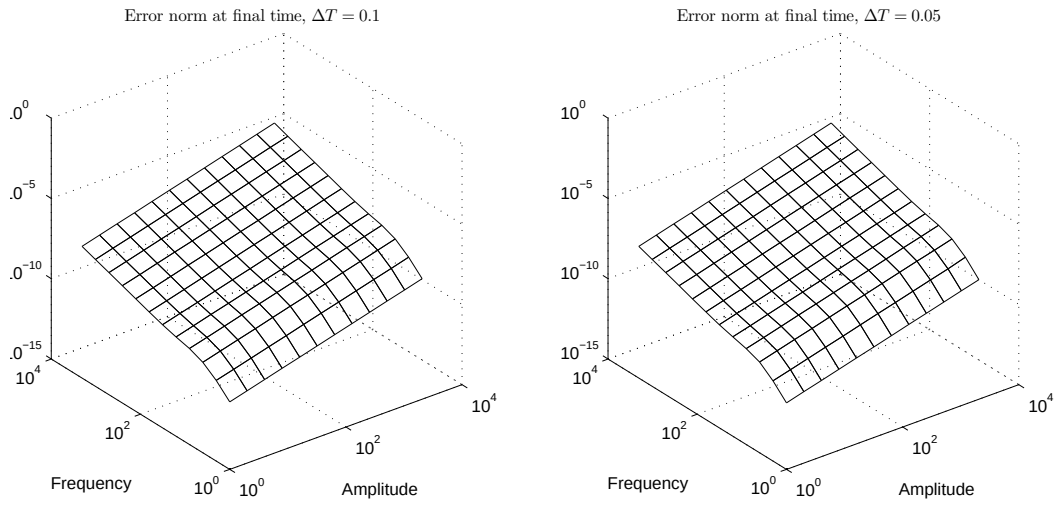

Figure 20: Error norm at final time, powers of 2, $\Delta T = 0.1$ and 0.05

Here, exactly as before, for $\Delta T = 0.1$ and 0.05 both amplitude and frequency are the sources of error growth. Let us look at two rest cases:

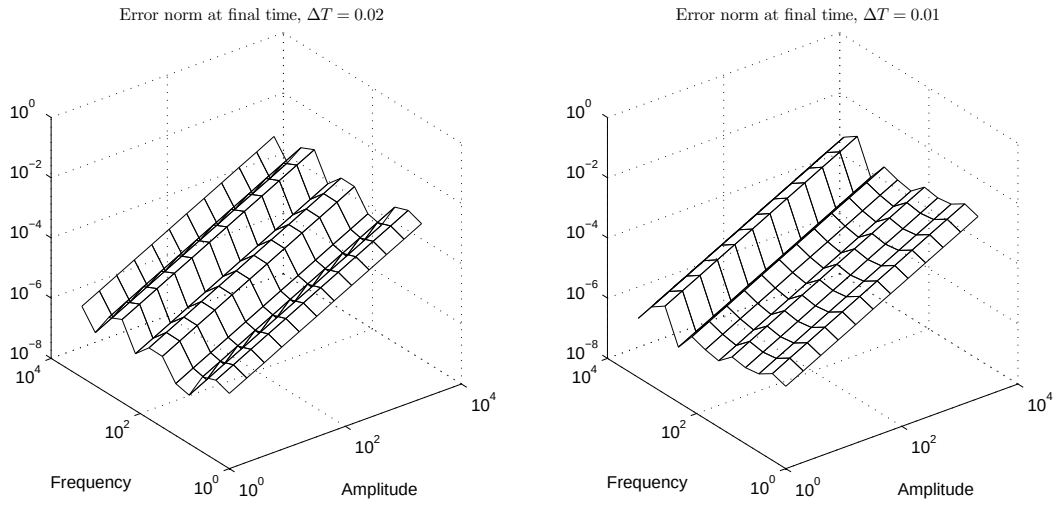

Figure 21: Error norm at final time, powers of 2, $\Delta T = 0.02$ and 0.01

As we can see even on a loglog plot, there is a kind of oscillation at the edge $A = 2048$.

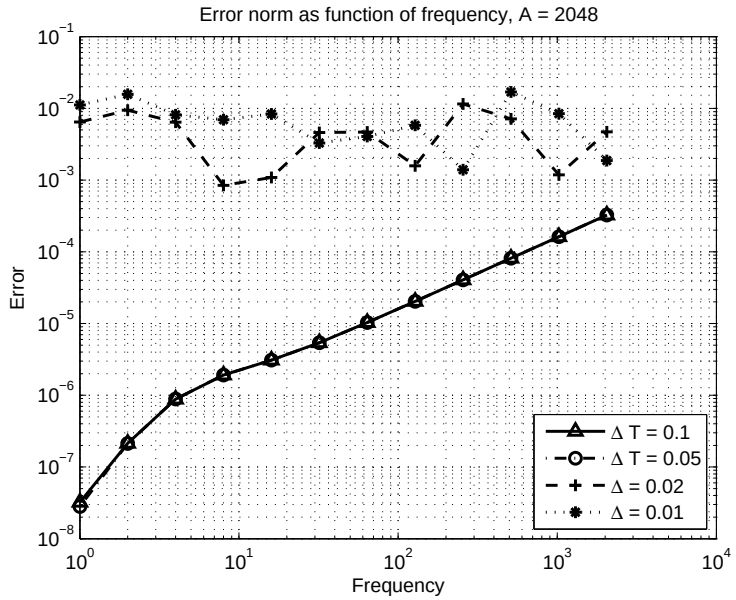

Figure 22: Error norm at final time as function of frequency, $A = 2048$

In figure 22, obviously, the first two cases produce the growing error, and there is no guarantee that the corresponding curves will meet a horizontal asymptote later. So, we can make a hypothesis that for large enough number of subintervals parareal produces a discrete solution with a bounded error. We should also point out that even for such values of parameters, the number of parareal iterations to reach the desired tolerance $\varepsilon = 0.0001$ does not exceed 3.

We also did a second experiment: fix $A = 1000$ (it plays a role of a factor according to previous results), and then plot the error norm, the analytical solution norm and the relative error $\frac{\|u_{parareal}(t_{end}) - u_{analytical}(t_{end})\|}{\|u_{analytical}(t_{end})\|}$ as a function of frequency ω, when $\omega = 1, \ldots, 500$.

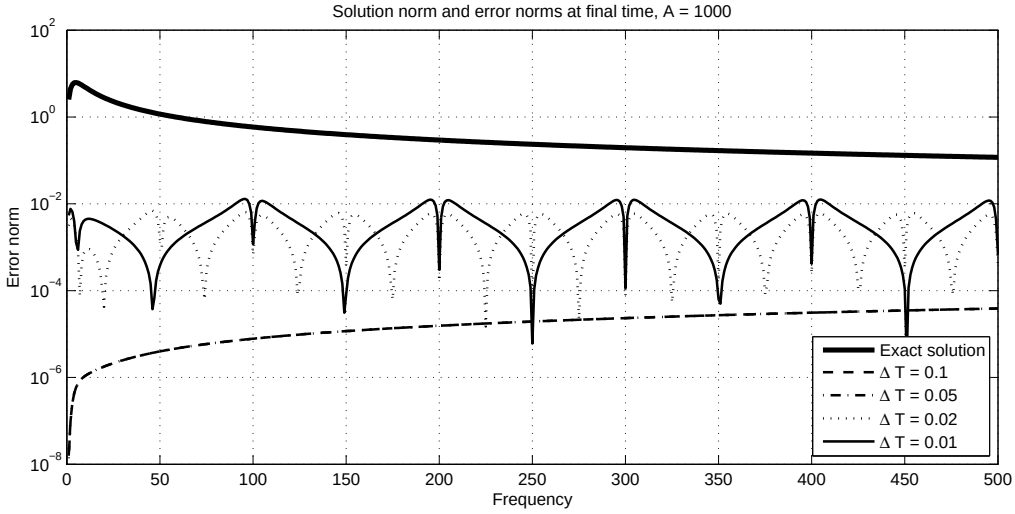

Figure 23: Analytical solution and error norms as functions of frequency, $A = 1000$

Here (see figure 23) we can observe oscillations of unusual type for two finest cases. Although these oscillations seem to have a finite upped boundary, the norm of the exact (i.e. analytical) solution slowly tends to zero as ω grows. Such behavior of analytical solution determines the behavior of relative error, which does grow, but seem to have an asymptotic behavior (see figure 24):

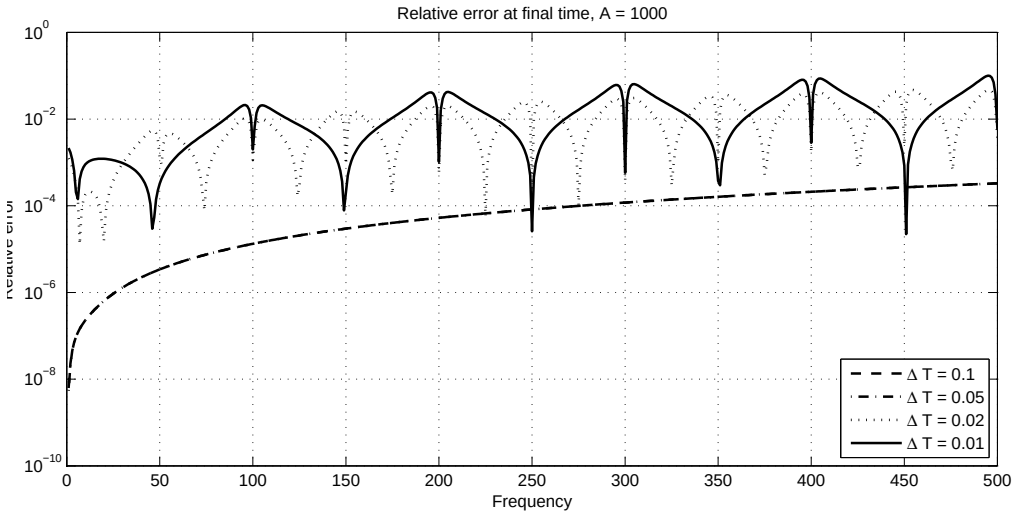

Figure 24: Relative error as functions of frequency, $A = 1000$

Anyway, even taking into account the fact of probably limited growth for the finest cases, one should note that this is a relative error, and it does not exceed 0.1 for relatively high frequency $\omega = 500$ and high amplitude $A = 1000$. That is why we can conclude that the parareal algorithm does cope quite well with such high-frequency model problem. The main error source is the amplitude, while the frequency is handled sufficiently well. The number of parareal iterations to reach the desired tolerance does not exceed 3 as well as in the previous experiments.

Although we have shown experimentally that the error seems to be bounded, this remains a hypothesis, which might be proved or disproved in later related works.

6 Conclusion and future work

6.1 Conclusion

The goal of the thesis was to study different time-integration schemes, to implement the concept of time-integrators within the HHG framework and to add the parareal algorithm on top of integrators.

In the first chapter we gave a motivation for adding several time integrators and briefly discussed them from a historical point of view. Moreover, we gave a basis for exploiting temporal parallelism and made an overview of several classes of time-parallel methods.

The second chapter contains derivation of several explicit time-integration schemes, of both one-step and multi-step classes. The global order of accuracy for each method is computed. One of adaptive methods, namely the Kutta-Merson scheme, is considered. Attention is also paid to starting procedures of multi-step methods, which are very important to preserve the order of accuracy. Stability concept is studied on a model ODE IVP, for all fixed-step methods so-called stability regions are computed and visualized.

In the 3rd chapter the parareal algorithm is described. First, it is given in a simple form using common notation of coarse and fine integrators. Then, a pseudo-code is presented, which can be easily translated into any procedural language. Computations complexity of the parareal algorithm is estimated in terms of complexities of both integrators involved. The parallel speedup is computed under assumption of a memory-shared parallel programming model. Some observation how to get the best speedup an parallel efficiency are given. Then, the stability issue is studied on the same model ODE IVP, the stability function is derived from stability functions of both integrators involved. Stability conditions for several (but not all) cases are presented. Convergence rates are studied and the a fact of superlinear convergence for fixed number of iterations is stated. Some results regarding the application of parareal to PDEs are also given. Then, the algebraic representation is considered, from which it becomes obvious where the parallelism comes from.

The 4th chapter includes the description of implementation issues with which the author has encountered during the work. First of all, we describe how the object oriented approach can be fit into the problem. Then, the implementations of two concepts - a time dependent problem and an integrator are discussed in details. Second of all, the parareal algorithm is implemented gradually. The first, prototype version, is done in Matlab. Then, a graphical illustration of how parareal works is given on a simple ODE IVP. Third of all, an implementation of parareal in HHG is described. The particular parallel programming model is chosen and the reasoning for that is also given. Some major drawbacks of HHG are covered, like the lack of full support of standard "rule of three" by some basic classes of HHG. Then, a list of formal requirements for the implementation is presented. These requirements are satisfied one by one by introducing pieces of source code with rich explanations. In the end, some comments regarding the unsuccessful attempt to do a hybrid MPI-openMP version are given.

The 5th chapter introduces the setting for testing the implemented time-integrator and parareal algorithm. First, a particular model problem of 3D inhomogeneous heat equation is chosen. Then, under certain assumptions, an analytical solution is computed. An imple-

mentation of the particular equation in terms of HHG is described. Then, in order to verify the implementation, all the time-integrators are tested to find their actual order of accuracy numerically. All the results are presented in tables and graphs, and all of them coincide with theoretical expectations. Thus a conclusion is made that all of them are implemented correctly. The next step done is the speedup check of the parareal algorithm. The computations are performed on LSS cluster, but, due to the assumptions of shared memory machine, only a single node (up to 32 cores) is used. The setting for the testing is given, and the test are done for 4 different ratios between the coarse and the fine stepsizes. All the results are put into tables, the speedup and the efficiency are computed. Then, the plots of the speedup and the efficiency as functions of the number of threads are presented. The numerical results absolutely agree with theoretical expectations from chapter 3, thus the conclusion is made that parareal is implemented correctly.

In the last part of the 5th chapter an attempt to find some limit cases of applicability of the parareal algorithm is made. The algorithm is tested on the model problem with varying parameters in the source term, namely the frequency and the amplitude. Then, for each case the norm of the error is computed at the final time. At first glance, both parameters seemed to be the sources of error growth. But it turns out that a high frequency can be handled much better than a high amplitude. A behavior of the error is studied for different numbers of sub-intervals, and it seems that if there are enough time-intervals, then the error is somehow bounded for an increasing frequency. Also the relative error for a fixed high amplitude is computed, and it is found out that this error does not exceed 0.1. Thus we conclude that our hypothesis that for varying frequency the error is bounded is found to be plausible, but still not proven.

6.2 Suggestions for future work

Although this thesis contains a detailed descriptions and discussions of several selected topics, it is not possible to cover all question related to time-integration and the parareal algorithm in HHG. That is why we can think of how the presented work can be continued in the future.

From our point of view, the most important direction of the future work is probably to combine the spatial parallelism implemented with MPI, which is already in HHG, with the parareal algorithm implemented with openMP. Such combination of temporal and spatial parallelism can be applied to extreme scale problems like the one in Terra-Neo. Moreover, this approach can be competitive with other algorithms which are parallel both in space and time.

Another direction might be to adopt parareal to distribute memory machines, which is currently not done because of the internal architecture of the HHG framework. In case of successful implementation, it will become possible to run really long-time simulations with large number of computational nodes used.

One more direction is to implement implicit integrators within HHG framework. Implicit integrators are important for PDEs with convection domination character. Currently they are to be re-implemented for each particular problem because of operator-based manner of HHG.

Since it is not possible to cover many test cases in one paper, the forth direction of the work might be to study the behavior of the parareal algorithm on more problems with extreme values of parameters, even without usage of spatial parallelism some unexpected results might be obtained, like in chapter five.

7 Bibliography

[1] Terra-neo project. `https://www10.informatik.uni-erlangen.de/Research/Projects/terraneo/`.

[2] E. Aubanel. Scheduling of tasks in the parareal algorithm. *Parallel Computing*, 37(3):172–182, 2011.

[3] L. Baffico, S. Bernard, Y. Maday, G. Turinici, and G. Zérah. Parallel-in-time molecular-dynamics simulations. *Physical Review E*, 66(5):057701, 2002.

[4] F. Bashforth and J. C. Adams. *An attempt to test the theories of capillary action: by comparing the theoretical and measured forms of drops of fluid*. University Press, 1883.

[5] A. Bellen and M. Zennaro. Parallel algorithms for initial-value problems for difference and differential equations. *Journal of Computational and applied mathematics*, 25(3):341–350, 1989.

[6] B. Bergen, G. Wellein, F. Hülsemann, and U. Rüde. Hierarchical hybrid grids: achieving teraflop performance on large scale finite element simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):311–329, 2007.

[7] K. Burrage. *Parallel and sequential methods for ordinary differential equations*. Clarendon Press, 1995.

[8] F. Chouly and M. A. Fernández. An enhanced parareal algorithm for partitioned parabolic-hyperbolic coupling. In *NUMERICAL ANALYSIS AND APPLIED MATHEMATICS: International Conference on Numerical Analysis and Applied Mathematics 2009: Volume 1 and Volume 2*, volume 1168, pages 1517–1520. AIP Publishing, 2009.

[9] L. Euler. *Institutionum calculi integralis*, volume 1. imp. Acad. imp. Saènt., 1768.

[10] C. Farhat and M. Chandesris. Time-decomposed parallel time-integrators: theory and feasibility studies for fluid, structure, and fluid–structure applications. *International Journal for Numerical Methods in Engineering*, 58(9):1397–1434, 2003.

[11] M. J. Gander and S. Vandewalle. Analysis of the parareal time-parallel time-integration method. *SIAM Journal on Scientific Computing*, 29(2):556–578, 2007.

[12] M. J. Gander and S. Vandewalle. *On the superlinear and linear convergence of the parareal algorithm*. Springer, 2007.

[13] M. L. Ghrist, B. Fornberg, and J. A. Reeger. Stability ordinates of adams predictor-corrector methods. *BIT Numerical Mathematics*, pages 1–18, 2012.

[14] W. Hackbusch. Parabolic multi-grid methods. In *Proc. of the sixth int'l. symposium on Computing methods in applied sciences and engineering, VI*, pages 189–197. North-Holland Publishing Co., 1985.

[15] E. Hairer, S. Nørsett, and G. Wanner. *Solving ordinary differential equations I. Nonstiff problems*. Springer-Verlag, Berlin, 1987.

[16] K. Heun. Neue methoden zur approximativen integration der differentialgleichungen einer unabhängigen veränderlichen. *Z. Math. Phys*, 45:23–38, 1900.

[17] B. Khalaf and D. Hutchinson. Parallel algorithms for initial value problems: parallel shooting. *Parallel computing*, 18(6):661–673, 1992.

[18] M. Kiehl. Parallel multiple shooting for the solution of initial value problems. *Parallel computing*, 20(3):275–295, 1994.

[19] W. Kutta. Beitrag zur näherungweisen integration totaler differentialgleichungen. 1901.

[20] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli. The waveform relaxation method for time-domain analysis of large scale integrated circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1(3):131–145, 1982.

[21] J.-L. Lions, Y. Maday, and G. Turinici. Résolution d'edp par un schéma en temps «pararéel». *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, 332(7):661–668, 2001.

[22] Y. Maday. The parareal in time algorithm. 2008.

[23] Y. Maday, E. Ronquist, and G. Staff. The parareal-in-time algorithm: basics, stability and more, 2006.

[24] Y. Maday and G. Turinici. A parareal in time procedure for the control of partial differential equations. *Comptes Rendus Mathematique*, 335(4):387–392, 2002.

[25] D. Mercerat, L. Guillot, and J.-P. Vilotte. Application of the parareal algorithm for acoustic wave propagation. In *AIP Conference Proceedings*, volume 1168, pages 1521–1524, 2009.

[26] A. S. Nielsen. *Feasibility study of the parareal algorithm*. PhD thesis, Master's thesis, Technical University of Denmark, DTU Informatics, E-mail: reception@ imm. dtu. dk, Asmussens Alle, Building 305, DK-2800 Kgs. Lyngby, Denmark. http://www. imm. dtu. dk/English. aspx, DTU supervisor: Allan P. Engsig Karup, apek@ imm. dtu. dk, DTU Informatics, 2012.

[27] J. Nievergelt. Parallel methods for integrating ordinary differential equations. *Communications of the ACM*, 7(12):731–733, 1964.

[28] C. Runge. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895.

[29] G. A. Staff and E. M. Rønquist. Stability of the parareal algorithm. In *Domain decomposition methods in science and engineering*, pages 449–456. Springer, 2005.

[30] M. Stämpfle. Mathematical methods. 2013.

[31] V. M. Verzhbitsky. *Foundations of numerical methods.* Vyschaya shkola, Moscow, 2005. [Russian].

8 Appendix

8.1 Matlab implementation of parareal

```
function [ U_next ] = G( rhs, t_current, dt_coarse, U_current )
%G coarse propagator
   U_next = U_current + dt_coarse * rhs(t_current, U_current);
end
```

```
function [ U_next ] = F( rhs, t_current, dt_fine, U_current, N_steps )
%F fine propagator
    for k = 1:N_steps
       %forward euler
       %U_next = U_current + dt_fine * rhs(t_current, U_current);

       % explicit midpoint
       U_next = U_current + dt_fine * rhs(t_current + 0.5 * dt_fine , U_current + 0.5 *
           dt_fine * rhs(t_current, U_current)  );

       U_current = U_next;
       t_current = t_current + dt_fine;

    end
end
```

```
clear all;
clc;
close all;
format long;

% rhs of the problem
rhs = @(t, u)(sin(t) .* cos(u));

% initial value
u_0 = 1;

% time-domain
T_start = 0;
T_end = 20;

% ratio between stepsizes
R = 10;

% stepsizes
dt_coarse = 0.5;
dt_fine = dt_coarse / R;

T = T_start:dt_coarse:T_end;

% number of parareal iterations
K_max = 4;
K_break = K_max; % to keep when the loop broke

% arrays to keep values
U = zeros(K_max, size(T,2));
U_tilde = zeros(K_max, size(T,2));
U_hat = zeros(K_max, size(T,2));

N = size(T,2);

% tolerance for convergence check
eps = 10^-6;
```

```
38
39 tic;
40 %================================ PARAREAL ================================
41 for k = 1:K_max
42    U(k,1) = u_0;
43    U_hat(k,1) = u_0;
44    U_tilde(k,1) = u_0;
45 end
46
47 % iteration 0
48 for n = 1:N-1
49     U_tilde(1,n+1) = G(rhs, T(n), dt_coarse, U_tilde(1,n));
50     U(1,n+1) = U_tilde(1,n+1);
51 end
52
53 %iteration 1...K_max-1
54 for k = 1:K_max-1
55
56     %loop to parallelize
57     parfor n = 1:N-1
58         U_hat(k,n+1) = F(rhs,T(n), dt_fine, U(k,n), R);
59     end
60
61     for n = 1:N-1
62         %prediction
63         U_tilde(k+1,n+1) =  G(rhs, T(n), dt_coarse, U(k+1,n));
64
65         %correction
66         U(k+1,n+1) = U_tilde(k+1,n+1) + U_hat(k,n+1) - U_tilde(k,n+1);
67     end
68
69     if(norm(U(k+1,:)-U(k,:),Inf) < eps)
70         K_break = k;
71         break;
72     end
73 end
74
75 parareal_elapsed_time = toc
76 %=========================TRUE FINE SOLUTION ==============================
77 T_fine = T_start:dt_fine:T_end;
78 U_fine = zeros(1,size(T_fine,2));
79 U_fine(1) = u_0;
80 tic;
81 for n = 1:size(T_fine,2)-1
82     U_fine(n+1) = F(rhs, T_fine(n), dt_fine, U_fine(n), 1);
83 end
84 serial_elapsed_time = toc;
85
86 speedup = serial_elapsed_time / parareal_elapsed_time
87 %==========================================================================
88 for k = 1:K_break
89     figure;
90     plot(T_fine, U_fine, 'r-','LineWidth',1.5);
91     hold on;
92     plot(T, U(k,:), 'b-x','LineWidth',1.5);
93     grid on;
94     xlabel('t');
95     ylabel('u');
96     title(strcat('Fine and parareal solution on iteration k = ',num2str(k)));
97     legend('Fine solution','Parareal');
98 end
```

Listing 8: Parareal in Matlab

9 Biography

I was born in Syktyvkar, Komi Republic, Russia, on the 10th of May, 1990. After finishing school with a silver medal, I entered Saint Petersburg State University in 2009, the faculty of Applied Mathematics and Control Processes. There I got a bachelor's degree in Information Technologies (with distinction) in 2013. The same year I was accepted to Erasmus Mundus Computer Simulations for Science and Engineering (COSSE) master programme. I moved to Sweden and spent one year there as a student of KTH Royal Institute of Technology in Stockholm. The second year of my master studies is at Friedrich-Alexander-Universität Erlangen-Nürnberg, where this master theses is done.