Implementing Parareal – OpenMP or MPI?

Daniel Ruprecht[a,b]

[a]School of Mechanical Engineering, University of Leeds, Woodhouse Lane, Leeds LS2 9JT, UK
[b]Centre for Computational Medicine in Cardiology, Institute of Computational Science, Università della Svizzera italiana, Via Giuseppe Buffi 13, CH-6900 Lugano, Switzerland

Abstract

The paper presents a comparison between MPI and OpenMP implementations of the parallel-in-time integration method Parareal. A special-purpose, lightweight FORTRAN code is described, which serves as a benchmark. To allow for a fair comparison, an OpenMP implementation of Parareal with pipelining is introduced, which relies on manual control of locks for synchronisation. Performance is compared in terms of runtime, speedup, memory footprint and energy-to-solution. The pipelined shared memory implementation is found to be the most efficient, particularly with respect to memory footprint. Its higher implementation complexity, however, might make it difficult to use in legacy codes.

Keywords: Parareal, parallel-in-time integration, pipelining, energy-to-solution, memory footprint
2010 MSC: 68W10, 65Y05, 68N19

1. Introduction

The fields of numerical simulation and computational science face a variety of challenges stemming from the massive increase in parallelism in state-of-the-art high-performance computing systems: projections suggest that exascale machines will require 100-million way concurrency [1]. This development mandates rethinking algorithms with respect to concurrency, fault-tolerance and energy efficiency but also the development of new and inherently parallel numerical methods. As a novel direction for the parallelisation of the solution of initial value problems, parallel-in-time (or time-parallel) integration schemes are attracting a quickly growing amount of attention. Their principle efficacy in extreme-scale parallel computations has been demonstrated [2, 3] and new methods and applications are frequently being introduced. Recently, the first software packages implementing parallel-in-time methods have also started to emerge [4, 5, 6].

Ideas how to parallelise in time go back to the 1960s [7, 8]. One widely studied method, which is also the subject of this paper, is *Parareal* [9]. Parareal is based on ideas from multiple shooting for boundary value problems, which have been adopted for the time dimension [10, 11]. Very similar in spirit is the *parallel implicit time-integrator* (PITA) [12]. A method akin to parallel Runge-Kutta methods is *revisionist deferred corrections* (RIDC) [13]. Based on a combination of Parareal with spectral deferred corrections (SDC) [14, 15] plus the introduction of a FAS correction term, the *parallel full approximation scheme in space and time* (PFASST) has been developed [16]. An extension of Parareal towards a multi-grid in time method is *multi-grid reduction in time* (MGRIT) [17]. A time multi-grid method based on a discontinuous Galerkin discretisation in time has also recently been proposed [18, 19]. An overview of the development of the field over the last 50 years has been given by Martin Gander [20].

Earlier literature has mainly focussed on mathematical properties of parallel-in-time methods [21], but, over the last years, several papers have appeared that study different strategies of implementing them on parallel computers. For example, the following configurations have been investigated: an OpenMP-based Parareal without spatial parallelisation [22]; an MPI-based implementation of PFASST with a hybrid MPI-Pthreads parallelisation in space [2]; a combination of PFASST with parallel multigrid in space, both using MPI [3]; Parareal and mesh decomposition

Email address: `d.ruprecht@leeds.ac.uk` (Daniel Ruprecht) using MPI in space and time [23, 24]; Parareal using OpenMP with parallelisation in space using MPI [25]; RIDC using OpenMP with MPI-based domain decomposition in space [26]; Parareal using MPI with OpenMP or CUDA in space [27]; MGRIT using MPI in space and time [17]; an MPI-based implementation of the time multi-grid DG method [18]. Moreover, scheduling of Parareal has been investigated on HPC systems [28] and cloud platforms [29] as well as an event-based implementation strategy [30]. While all these different configurations have been shown to be feasible and able to provide reasonable performance, to date there are no *comparisons* between different implementation strategies for parallel-in-time methods. Therefore, the question this paper is concerned with is simply: *I want to implement Parareal - should I use MPI or OpenMP?* The answer, of course, is not as straightforward as the question and cannot be addressed exhaustively in a single paper, but the here presented results are a first step to shed some light on the tradeoffs that come with different approaches to mapping parallel-in-time (and, ultimately, space-time parallel) methods to hardware.

To tackle the question, the paper introduces PararealF90, a lightweight, stand-alone and special-purpose FORTRAN code [31], which solves 3D Burgers' equation using Parareal based on a forward Euler as coarse, a SSP Runge-Kutta-3 method [32] as fine integrator as well as different finite difference stencils for the spatial discretisation. PararealF90 is deliberately designed to not use external libraries other than MPI and OpenMP and therefore relies on explicit methods to avoid the need for a nonlinear solver. Using some external solver library would greatly complicate isolating the effect of different implementation strategies for Parareal, because the library's performance would interact with Parareal's performance, which would make it very difficult to isolate the effects of the implementation. Avoiding external packages also makes the used code easily portable, because it only requires an MPI FORTRAN compiler built with thread support. Since the code is open-source and available under a BSD license, the here presented benchmarks can easily be run on other platforms with other compilers or compared against other Parareal implementations. Even though the code is special-purpose solving a specific benchmark, its general layout corresponds to a widely used motif, where explicit integrators call routines that perform stencil operations in space. For example, codes used for numerical weather prediction often have a similar design [33], although of course at a significantly higher level of complexity.

The paper starts out with a brief summary of Parareal and sketches, in pseudo-code, how straightforward implementations in MPI and OpenMP look like. By comparing projected speedup, the importance of using *pipelining* [15] to hide some of the cost of the coarse propagator is illustrated. While this pipelining comes naturally when Parareal is implemented with MPI (preventing it would require to artificially introduce `MPI_BARRIER` statements), a straightforward loop parallelisation with OpenMP does not enable it. Therefore, the paper introduces (apparently for the first time), an OpenMP-based implementation of Parareal with pipelining. While it is shown that this version has clear advantages in terms of performance, it requires manual synchronisation between threads using locks as well as thread-safe functions for the coarse and fine integrator. It is thus more involved to implement than the MPI version. The remainder of the paper then compares the three different versions (MPI, OpenMP without and with pipelining) in terms of runtime and speedup, memory footprint and energy-to-solution on a a single node of a Linux cluster and a Cray XC40 computer. It is found that the OpenMP and MPI based implementations of Parareal with pipelining give almost identical performance in terms of runtime and speedup, but that the OpenMP version has a significantly smaller memory footprint and, depending on the compiler, consumes less energy. This could be particularly interesting in view of the fact that concerns about the memory requirements of parallel-in-time integration methods have been voiced [34]. The conclusions then summarise the results and provide a discussion of future research and generalisation of the obtained findings.

2. The method: Parareal

The starting point for Parareal is an initial value problem of the form

$$\dot{\mathbf{q}} = \mathbf{f}(\mathbf{q}(t), t), \ \mathbf{q}(0) = \mathbf{q}_0, \ t \in (0, T], \tag{1}$$

which, in the numerical examples below, arises from the spatial discretisation of a PDE ("method-of-lines") with $\mathbf{q} \in \mathbb{R}^{N_{\text{dof}}}$ being a vector containing all degrees-of-freedom. Let $\mathcal{F}_{\delta t}$ denote a numerical procedure for the approximate solution of (1), for example a Runge-Kutta method. Denote further by

$$\mathbf{q} = \mathcal{F}_{\delta t}(\tilde{\mathbf{q}}, t_2, t_1) \tag{2}$$

the result of approximately integrating (1) forward in time from some starting value[1] $\tilde{\mathbf{q}}$ at a time t_1 to a time $t_2 > t_1$ using $\mathcal{F}_{\delta t}$. That is,

$$\mathbf{q} = \mathcal{F}_{\delta t}(\mathbf{q}_0, T, 0) \tag{3}$$

would indicate sequentially solving the full initial value problem in serial using the time stepper denoted by $\mathcal{F}_{\delta t}$. In order to parallelise the numerical integration of (1), Parareal and other so-called *parallel-across-the-steps* methods [35] introduce a decomposition of the time interval $[0, T]$ into time-slices $[t_p, t_{p+1}]$, $p = 0, \ldots, P-1$ where P is the number of cores, equal to the number of processes or threads, to be used in time. In the implementations sketched in Section 3, time slice $[t_p, t_{p+1}]$ is assigned to the core running either thread number p (in the OpenMP variants) or the process with rank p (in the MPI version). For simplicity, assume here that all time slices have the same length and that the whole interval $[0, T]$ is covered with a single set of time slices - otherwise, some form of restarting or *moving window* [36] has to be used.

Introduce now a second time integrator denoted $\mathcal{G}_{\Delta t}$, which has to be much cheaper to compute but can also be much less accurate (commonly referred to as the "coarse propagator"). Typically, a lower order method with a larger time step is used here, but a lower order spatial discretisation can be used as well. Using fewer degrees of freedom on the coarse level is also possible, but necessitates the introduction of suitable transfer operators (interpolation and restriction) and can degrade convergence [37]. Parareal starts off with a *prediction step*, computing a rough guess of the starting value $\mathbf{q}_p^0$ at the beginning of each time slice by

$$\mathbf{q}_p^0 = \mathcal{G}_{\Delta t}(\mathbf{q}_0, t_p, 0), \quad p = 0, \ldots, P-1. \tag{4}$$

Here, subscript p indicates an approximation of the solution at time t_p. These approximate starting values are used to start the following iteration, run concurrently on each time slice,

$$\mathbf{q}_{p+1}^k = \mathcal{G}_{\Delta t}(\mathbf{q}_p^k, t_{p+1}, t_p) + \mathcal{F}_{\delta t}(\mathbf{q}_p^{k-1}, t_{p+1}, t_p) - \mathcal{G}_{\Delta t}(\mathbf{q}_p^{k-1}, t_{p+1}, t_p), \quad p = 0, \ldots, P-1, \quad k = 1, \ldots, K. \tag{5}$$

As k increases, this iteration converges at the endpoints of the slices to the same solution provided by (3), that is $\mathbf{q}_p^k \to \mathcal{F}_{\delta t}(\mathbf{q}_0, t_p, 0)$ for $p = 0, \ldots, P$. Because the computationally expensive evaluation of the fine propagator (referred to as the *fine integrator step* in Section 3) can be parallelised across time slices, iteration (5) can run in less wall clock time than the direct time-serial integration (3) – provided the coarse method is cheap enough and the number of required iterations K is small. Computation of the fine values is followed by the *correction step*: the updated value $\mathbf{q}_p^k$ from the previous time slice is obtained, the coarse integrator is applied to it and (5) is evaluated to obtain $\mathbf{q}_{p+1}^k$ which is then used to correct the value on the next time slice and so on. Note that, in contrast to the fine integrator, the correction step has to be performed in correct order, going step by step from the first time slice to the last. Also, when using a distributed memory parallelisation, the value $\mathbf{q}_p^k$, required in (5) to compute $\mathcal{G}_{\Delta t}(\mathbf{q}_p^k, t_{p+1}, t_p)$, has to be communicated from the process handling time slice $[t_{p-1}, t_p]$ to the one handling $[t_p, t_{p+1}]$.

2.1. *Performance model*

The expected performance of Parareal, referred to here as *projected speedup*, can be described by a simple theoretical model [15]. Here, the model is briefly repeated to more clearly illustrate the effect and importance of pipelining Parareal, see Subsection 2.2. It is also used as a baseline to compare against the measured speedups reported in Section 4.

Denote, as above, by P the number of cores in time, equal to the number of time slices. Further, denote by c_c the cost of integrating over one time slice using $\mathcal{G}_{\Delta t}$ and by c_f the cost when using $\mathcal{F}_{\delta t}$. Because all time slices are assumed to consist of the same number of steps and an explicit method is used here, it can be assumed that c_f and c_c are identical for all time slices. This does not necessarily hold for an implicit method where differences in the number of iterations required by the nonlinear solver to converge can introduce load imbalances [38]. A simple theoretical model for the speedup of Parareal using K iterations against running the fine method in serial now reads

$$s_\mathrm{np}(P) = \frac{P c_\mathrm{f}}{(1+K) P c_\mathrm{c} + K c_\mathrm{f}} = \frac{1}{(1+K)\frac{c_\mathrm{c}}{c_\mathrm{f}} + \frac{K}{P}}. \tag{6}$$

Equation (6) illustrates the necessity for a cheap coarse method to minimise the ratio $c_\mathrm{c}/c_\mathrm{f}$ while still guaranteeing rapid convergence to minimise the terms K/P and $1+K$.

[1]Here, the prescribed value $\mathbf{q}_0$ at $t = 0$ is referred to as initial value, while the value $\mathbf{q}_p$ the beginning of a time slice is called starting value.

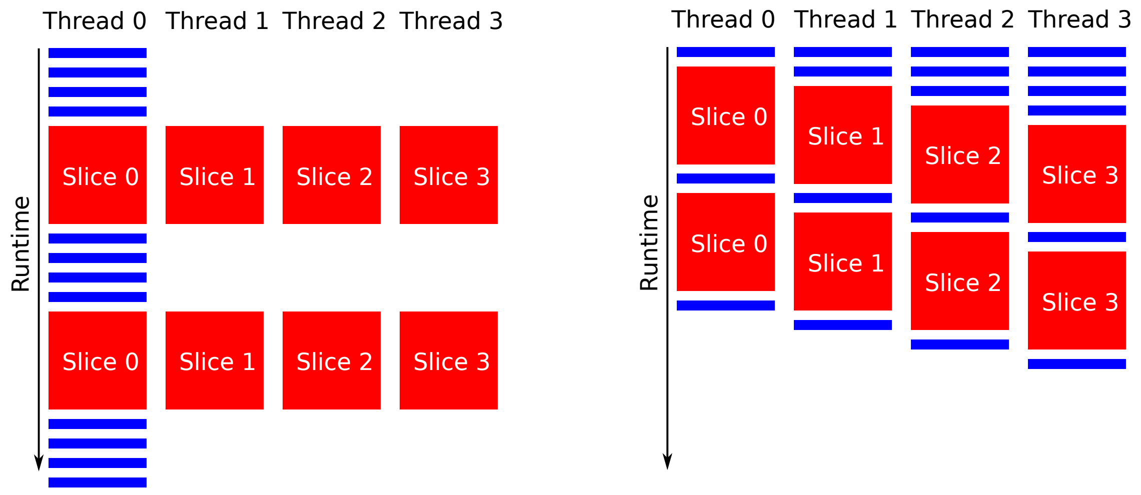


Figure 1: Execution diagram for Parareal without (left) and with (right) pipelining. Red blocks correspond to c_f, the time needed to run the fine integrator $\mathcal{F}_{\delta t}$ over one time slice $[t_p, t_{p+1}]$ while blue blocks correspond to c_c, the time required by the coarse method $\mathcal{G}_{\Delta t}$. Pipelining allows to hide some of the cost of the coarse method. While it comes naturally when using MPI to implement Parareal (there, `Thread` would refer to a `Process`), using simple loop-based parallelism with OpenMP results in the non-pipelined version shown on the left. A pipelined OpenMP Parareal is introduced in Section 3 but requires manual synchronisation between threads.

2.2. *Pipelining Parareal*

It has been pointed out that a proper implementation of Parareal allows to hide some of the cost of the coarse propagator and the name *pipelining* has been coined for this approach [15]. Figure 1 sketches the execution diagrams of both a non-pipelined (left) and pipelined (right) implementation for four time slices. As can be seen, pipelining reduces the effective cost of the coarse correction step in each iteration from $P \times c_\mathrm{c}$ to c_c – but note that the initial prediction step still has cost $P \times c_\mathrm{c}$ as before. For pipelined Parareal, estimate (6) changes to

$$s_\mathrm{p}(P) = \frac{P c_\mathrm{f}}{P c_\mathrm{c} + K c_\mathrm{c} + K c_\mathrm{f}} = \frac{1}{\left(1 + \frac{K}{P}\right)\frac{c_\mathrm{c}}{c_\mathrm{f}} + \frac{K}{P}}. \tag{7}$$

Because $K/P \ll K$, the pipelined version allows for better speedup, that is $s_\mathrm{np}(P) \leq s_\mathrm{p}(P)$. However, because pipelining only hides cost from the coarse integrator, the effect is smaller when the coarse method is very cheap and $c_\mathrm{c}/c_\mathrm{s} \ll 1$. In that case, the term K/P dominates the estimate which is not affected by pipelining. The effect is illustrated in Figure 2. Here, the projected speedup according to (6) and (7) is visualised for two different values of $\sigma := c_\mathrm{c}/c_\mathrm{f}$. For the sake of convenience, the number of iterations is assumed to be constant at $K = 3$ for all values of P, although guaranteeing that Parareal and the fine method ultimately deliver comparable accuracy will typically require more iterations for large values of P [27]. The figure clearly shows the importance of pipelining to increase performance of Parareal: even for a very small ratio of $\sigma = 0.005$ (where the coarse method is two hundred times faster than the fine), pipelining still has a noticeable effect on speedup, at least for larger core counts. Any implementation of Parareal aiming to be reasonable efficient should thus make use of pipelining: however, as discussed in Section 3, pipelining Parareal when using OpenMP is much more involved than with MPI, see corresponding implementation presented in Subsection 3.3. Note that from the performance models one can see that both $s_\mathrm{p}(P)$ and $s_\mathrm{np}(P)$ are always smaller than P/K, independent of the cost of the coarse integrator. This bound is illustrated by the black line.

3. Implementation using MPI and OpenMP: PararealF90

The code used here for benchmarking is written in Fortran 90 [31] and available under an open-source BSD license. It is special-purpose and tailored to solve a single benchmark problem, 3D Burgers' equation

$$u_t + u \cdot \nabla u = \nu \Delta u \tag{8}$$

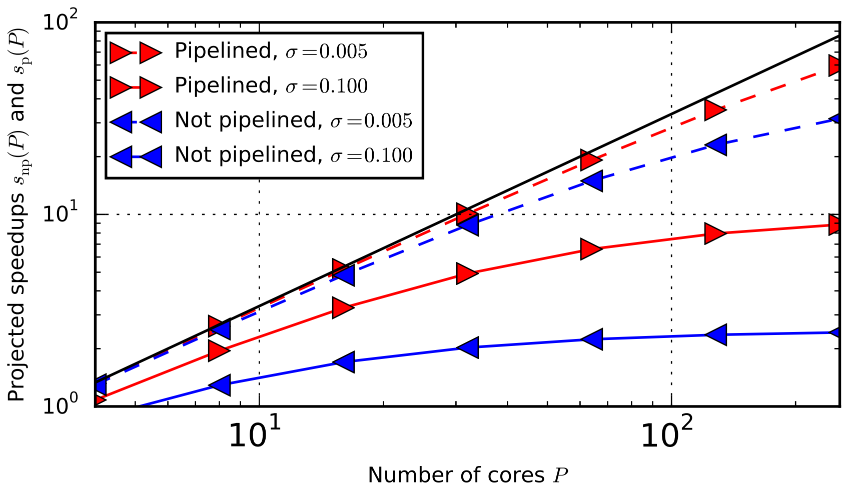


Figure 2: Speedup for Parareal as projected by (6) and (7) for coarse-to-fine ratios of $\sigma = c_c/c_f = 0.005$ (dashed) and $\sigma = c_c/c_f = 0.1$ (solid). The number of iterations is assumed to be $K = 3$, although typically it will increase with P. Decreasing σ improves overall speedup, but the relative benefit of pipelining is diminished. Nevertheless, even for very small coarse-to-fine ratios, pipelining still noticeably improves performance for large core counts. The black line is the upper limit on speedup of K/P.

on $[0,1]^3 \subset \mathbb{R}^3$ with periodic boundary conditions. While this a rather specific setup, finite difference stencils are a widely used motif in computational science: even though specifics and runtimes will be different if more complex problems are solved, the general concepts are used in complex application codes, e.g. by means of domain specific embedded languages (DSEL) [33] and the possibility to use Parareal in such a DSEL has been shown [27]. Thus, conclusions in terms of performance of different implementations of Parareal can be generalised to some extent, in particular because the Parareal routines only operate on linear arrays and do not see any specifics of the underlying time steppers or spatial discretisation. By exchanging the modules implementing the spatial discretisation, PararealF90 could be used to solve and benchmark different equations using stencil-based discretisations.

The time integration module provides two different methods, a strong stability preserving Runge-Kutta method (RK3-SSP) [32] and a first order forward Euler. The module for the spatial discretisation provides a fifth order WENO finite difference discretisation [32] and a simple first order upwind stencil for the advection term as well as a second and fourth order centred stencil for the diffusive term. Three different modules provide three different implementations of Parareal[2]: with pipelining using MPI as described in Subsection 3.1, the simple loop-based parallelisation without pipelining using OpenMP, see Subsection 3.2, and the more involved implementation with pipelining in OpenMP introduced in Subsection 3.3.

For $\mathcal{G}_{\Delta t}$, the forward Euler with upwind and second order centred stencils is used, while $\mathcal{F}_{\delta t}$ uses the RK3-SSP integrator, a 5th order WENO for advection and a fourth order centred stencil for diffusion. Note that all three implementations of Parareal use the same modules to provide the coarse and fine integrator and spatial discretisation, as sketched in Figure 3. Since all three versions rely on the same implementation for the actual integrators and spatial discretisation, differences in performance should therefore solely emerge from the different Parareal routines wrapped around the compute routines. To run coarse and fine serial reference simulations, driver functions are used that directly call the same time stepper routines that are used within Parareal with the same configurations.

The code also comes with a test harness: in particular, the tests guarantee that all three implementations of Parareal produce results that are identical up to a tolerance of $\varepsilon = 10^{-14}$ and thus essentially to round-off error. To detect possible race conditions, the comparison test can be performed multiple times: up to 100 instances of the test were run

[2]In PararealF90, the modules implementing the three different versions are called `parareal_mpi.f90`, `parareal_openmp.f90` and `parareal_openmp_pipe.f90`.

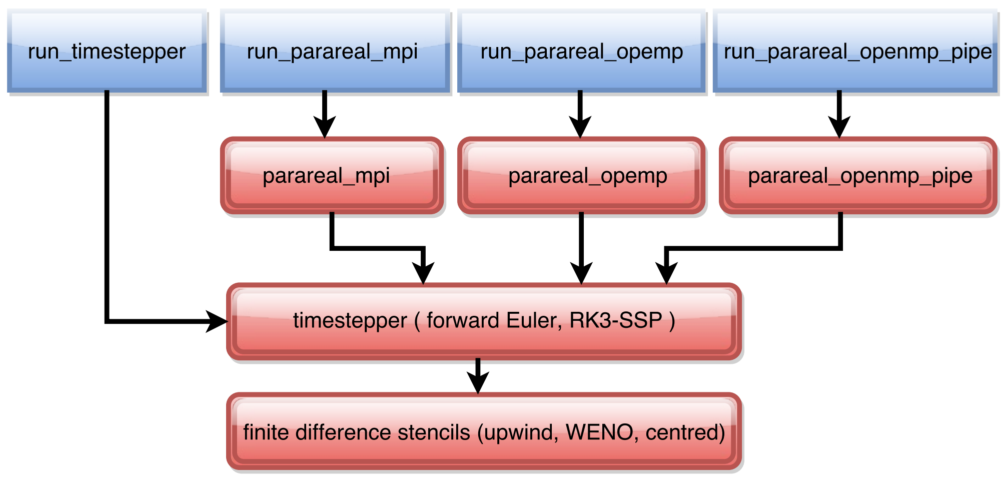


Figure 3: Structure of the PararealF90 code: Four different binaries (blue) are used to run three different versions of Parareal plus serial runs for reference. The three different Parareal versions are implemented in three different modules (red), but all call the same modules for time stepping and spatial discretisation. Differences in performance should therefore be caused by the differences in the three Parareal modules only.

and passed on both used architectures described in Subsection 4.1. The tests also use various randomised parameters (randomised in a small range to avoid too large problems with very long runtimes): diffusion parameter ν, number of time slices and processors P, number of finite difference nodes N (each direction has a different value in the tests) and number of fine and coarse steps per time slice. Furthermore, all three implementations of Parareal use three auxiliary buffers: q to store the fine value and for communication, δq to store the difference $\mathcal{F}_{\delta t}(q) - \mathcal{G}_{\Delta t}(q)$ needed in the correction step and q_c to store the coarse value from the previous iteration. It seems that without introducing additional communication, three copies per time slice is the minimum storage required for Parareal.

Both used systems have nodes with two multi-core CPUs, see the description in Subsection 4.1. For the OpenMP versions to be efficient, they therefore have to take care not only of thread safety but also take into account *non-uniform memory access* (NUMA): fetching data from memory associated with the other CPU requires communication through the intra-node interconnect and incurs overhead. Thus, in PararealF90, solution buffers are extended by one dimension, e.g. `Q(i,j,k,p)`, where i, j, k refer to the three spatial coordinates and p to the thread number. In the discussion below, the first three dimensions are omitted and the buffer corresponding to thread p is indicated simply by `Q(`p`)`. By using first touch initialisation, each thread ensures its respective part of the solution data is allocated locally. This strategy provides thread safety, as long as each thread is only accessing its own part of the buffers, as well as data affinity. All routines at the level of timestepper and below, cf. Figure 3, operate only on the buffers belonging to the respective thread. In the pipelined Parareal version with OpenMP, however, threads have to access buffers assigned to other threads: manual setting and unsetting of locks is required here to avoid race conditions, see Subsection 3.3. [39]

3.1. Parareal with pipelining in MPI

The implementation of Parareal with MPI is straightforward and has been illustrated before [27, 38]. However, it is repeated here for the sake of completeness and sketched in Algorithm 1.

- **Prediction phase:** lines 1.3 and 1.4. Every process generates its own coarse starting value $\mathbf{q}_p^0 = \mathcal{G}_{\Delta t}(\mathbf{q}_0, t_p, 0)$ and computes $\mathcal{G}_{\Delta t}(\mathbf{q}_p^0, t_{p+1}, t_p)$ for use in the correction. Later processes have to perform more time steps and thus take longer to complete the prediction phase, leading to the pipelined execution model sketched in Figure 1, since no global synchronisation points exist.

- **Fine integrator:** lines 1.6 and 1.7. Each process computes the fine value $\mathcal{F}_{\delta t}(\mathbf{q}_p^k, t_{p+1}, t_p)$ in line 1.6 and, in order to free the buffer q for reuse in the receive, computes the difference $\delta q := \mathcal{F}_{\delta t}(\mathbf{q}_p^k, t_{p+1}, t_p) - \mathcal{G}_{\Delta t}(\mathbf{q}_p^k, t_{p+1}, t_p)$ between coarse and fine value in line 1.7 and stores it in δq.

Algorithm 1: Parareal using MPI

```
     input: Initial value $q_0$; number of iterations $K$
1.1  $q \leftarrow q_0$
1.2  $p$ = MPI_COMM_RANK()
1.3  $q \leftarrow \mathcal{G}_{\Delta t}(q, t_p, 0)$
1.4  $q_c \leftarrow \mathcal{G}_{\Delta t}(q, t_{p+1}, t_p)$
1.5  for $k = 1, K$ do
1.6      $q \leftarrow \mathcal{F}_{\delta t}(q, t_{p+1}, t_p)$
1.7      $\delta q \leftarrow q - q_c$
1.8      if Process not first then
1.9          MPI_RECV($q$, source $= p - 1$)
1.10     end
1.11     else
1.12         $q \leftarrow q_0$
1.13     end
1.14     $q_c \leftarrow \mathcal{G}_{\Delta t}(q, t_{p+1}, t_p)$
1.15     $q \leftarrow q_c + \delta q$
1.16     if Process not last then
1.17         MPI_SEND($q$, target $= p + 1$)
1.18     end
1.19 end
```

- **Update phase:** lines 1.8–1.18. To receive the updated value from the previous time slice, every process but the first posts an MPI_RECV. The first process simply copies the initial value $\mathbf{q}_0$ into the buffer q instead. After the receive is completed, the coarse value $\mathcal{G}_{\Delta t}(\mathbf{q}_p^{k+1}, t_{p+1}, t_p)$ of the new starting value is computed and the updated end value $\mathbf{q}_{p+1}^{k+1} = \mathcal{G}_{\Delta t}(\mathbf{q}_p^{k+1}, t_{p+1}, t_p) + \delta q$ is computed and send to the process handling the following time slice using MPI_SEND.

Note how this implementation naturally features pipelining as sketched in Figure 1 (right). In order to obtain the non-pipelined execution flow sketched in Figure 1 (left), MPI_BARRIER directives would have to be added artificially after the prediction step, that is after line 1.4, and before the correction, that is before line 1.8. Parareal without pipelining in MPI would therefore be more complex and deliver reduced performance, so that this strategy seems rather senseless and is not explored here. Previous benchmarks did not show a significant difference between blocking and non-blocking communication [27] and for the sake of simplicity, blocking communication is used here.

3.2. *Parareal without pipelining in OpenMp*

Implementing Parareal without pipelining in OpenMP is relatively straightforward as sketched in Algorithm 2 and has been previously described [22]. It requires only parallelisation of the loop between lines 2.8–2.13 using the corresponding OpenMP directives, in case of Fortran `OMP PARALLEL DO` and `OMP END PARALLEL DO`. However, in order to achieve good performance, the implementation of $\mathcal{F}_{\delta t}$ must correctly deal with issues like NUMA-awareness and memory locality. It also has to be thread safe. The code consists of the following three parts:

1. **Prediction step:** lines 2.3–2.6. Here, coarse values $\mathcal{G}_{\Delta t}(\mathbf{q}_p^0, t_{p+1}, t_p)$ are computed for $0, \ldots, P-1$ and stored in buffer $q_c(p)$ for use in the first iteration. These coarse value are also used as initial starting values for the next time slice, that is $\mathbf{q}_{p+1}^0 = \mathcal{G}_{\Delta t}(\mathbf{q}_p^0, t_{p+1}, t_p)$ and, as such, stored in $q(p+1)$ for use in the first instance of the Parareal loop where $k = 1$.
2. **Fine integrator:** lines 2.8–2.13. Computation of fine values $\mathcal{F}_{\delta t}(\mathbf{q}_p^k, t_{p+1}, t_p)$ is parallelised over all time slices using `OMP PARALLEL DO` directives. It is followed by computation of the difference $\delta q := \mathcal{F}_{\delta t}(\mathbf{q}_p^k, t_{p+1}, t_p) - \mathcal{G}_{\Delta t}(\mathbf{q}_p^k, t_{p+1}, t_p)$, where the latter value is still stored in buffer $q_c(p)$ from either the previous iteration (for $k > 1$) or the prediction step (in $k = 1$). Because all instances of the fine integrator are independent of each other, the order in which the loop is executed is irrelevant and no synchronisation is needed here.

3. **Update step:** lines 2.14–2.20. The update is initialised by setting $\mathbf{q}_0^{k+1} = \mathbf{q}_0$. Then, in the loop, the coarse value $\mathcal{G}(\mathbf{q}_p^{k+1}, t_{p+1}, t_p)$ of $\mathbf{q}_p^{k+1}$ is computed and the correction step $\mathbf{q}_{p+1}^{k+1} = \mathcal{G}_{\Delta t}(\mathbf{q}_p^{k+1}, t_{p+1}, t_p) + \delta q$ is performed. The computed value $\mathbf{q}_p^{k+1}$ is the updated starting value for the next time slice and thus written into $q(p+1)$ for use in the next instance of the loop. Note that the update loop is outside of the parallel region: therefore, only the master thread executes it and the loop is automatically performed in the correct order, without danger of race conditions.

The drawback of not pipelining is also clearly visible: of P cores, $P-1$ are idle except for the parallelised loop over the fine method. One advantage of this approach is that solutions from all time-slices readily available, which is useful for variants of Parareal that recycle information from previous iterations, e.g. KSE-Parareal [22, 40].

Algorithm 2: Parareal without pipelining using OpenMP

```
     input: Initial value $q_0$; number of iterations $K$
2.1  $q(0) \leftarrow q_0$
2.2  $N$ = OMP_GET_MAX_THREADS()
2.3  for $p = 0, N-1$ do
2.4      $q_c(p) \leftarrow \mathcal{G}(q(p), t_{p+1}, t_p)$
2.5      $q(p+1) \leftarrow q_c(p)$
2.6  end
2.7  for $k = 1, K$ do
2.8      OMP PARALLEL DO
2.9      for $p = 0, N-1$ do
2.10         $q(p) \leftarrow \mathcal{F}_{\delta t}(q(p), t_{p+1}, t_p)$
2.11         $\delta q(p) \leftarrow q(p) - q_c(p)$
2.12     end
2.13     OMP END PARALLEL DO
2.14     $q(0) \leftarrow q_0$
2.15     for $p = 0, N-1$ do
2.16         $q_c(p) \leftarrow \mathcal{G}_{\Delta}(q(p), t_p, t_{p-1})$
2.17         if $p < N-1$ then
2.18             $q(p+1) \leftarrow q_c(p) + \delta q(p)$
2.19         end
2.20     end
2.21 end
```

3.3. *Parareal with pipelining in OpenMp*

The implementation of Parareal with pipelining in OpenMP introduced here is sketched in Algorithm 3. To implement pipelined Parareal with OpenMP, essentially the whole algorithm is enclosed in one parallel region: threads are spawned by the `OMP PARALLEL` directive in line 3.2 and terminated by `OMP END PARALLEL` in line 3.36. Because this version requires manual synchronisation, a number of OpenMP locks is created using `OMP_INIT_LOCK` (not shown), one for each thread. During the fine integrator and update step, these locks are set and unset using `OMP_SET_LOCK` and `OMP_UNSET_LOCK` to protect buffers during writes and avoid race conditions.

- **Prediction step:** lines 3.4–3.12. Just as in the MPI example, each thread is computing its own coarse prediction of its starting value $\mathbf{q}_p^0$ in a parallelised loop. The coarse value $\mathcal{G}_{\Delta t}(\mathbf{q}_p^0, t_{p+1}, t_p)$ is also computed and stored for use in the first iteration. The later the time slice (indicated by a higher thread number p), the more steps the thread must compute and thus the larger its workload. Therefore, at the end of the coarse prediction loop, the `NO WAIT` clause is required to avoid implicit synchronisation and enable pipelining. [3]

[3]Note that the prediction step could be implemented without a loop, by just using the thread number. However, within the Parareal iteration the

- **Parareal iteration:** lines 3.13–3.35. Here, both the fine integrator and update step are performed inside a single loop over all time slices, parallelised by `OMP DO` directives. Because parts of the loop (the update step) have to be executed in serialised order, the `ORDERERD` directive has to be used in line 3.14. Again, to avoid implicit synchronisation at the end of the loop, the `NO WAIT` clause is required in line 3.34.
  - **Fine integrator:** lines 3.16–3.19. Before the fine integrator is executed, an `OMP_LOCK` is set to indicate that the thread will start writing into buffer $q(p)$. Because thread $p-1$ accesses this buffer in its update step, locks are necessary to prevent race conditions and incorrect solutions. After the lock is set, the thread proceeds with the computation of $\mathcal{F}_{\delta t}(\mathbf{q}_p^k, t_{p+1}, t_p)$ and computation of the difference between coarse and fine value δq. Then, since $q(p)$ is now up to date and δq ready, the lock can be released.
  - **Update step:** lines 3.20–3.32. The update step has to be performed in proper order, from first to last time slice. Therefore, it is enclosed in `ORDERED` directives, indicating that this part of the loop is to be executed in serial order. Then, as in the two other versions, the update step is initialised with $\mathbf{q}_0^{k+1} = \mathbf{q}_0$. For every time slice, the coarse value of the updated initial guess is computed and the update performed. The updated end value is written into buffer $q(p+1)$ to serve as the new starting value for the following time slice. However, to prevent thread p from writing into $q(p+1)$ while thread $p+1$ is still running the fine integrator, thread p sets `OMP_LOCK` number $p+1$ before performing the update.

This implements a pipelined Parareal in OpenMP and achieves runtimes that are essentially identical to the one provided by the MPI version. The necessity to manually control synchronisation between threads, however, makes it the most demanding variant in terms of code complexity. As shown in Section 4, it outperforms the MPI implementation in terms of memory requirements and energy consumption.

4. Numerical results

In this section, the three different implementations of Parareal are compared with respect to runtime in Subsection 4.2, memory footprint in Subsection 4.3 and energy consumption in Subsection 4.4. The parameters for the simulation comprise a viscosity parameter of $\nu = 0.02$ and the spatial discretisation on both levels with $N_x = N_y = N_z = 40$ grid points in every direction. The simulation is run until $T = 1.0$ with a coarse time step of $\Delta t = 1/192$ and a fine step of $\delta t = 1/240$. Because of the quite high computational cost of the WENO-5 method in comparison to a cheap first order upwind scheme and the fact that RK3SSP needs three evaluations of the right hand side per step while the Euler method needs only one, the coarse propagator is about a factor of forty faster, despite the fact that the coarse step is only a factor of 1.25 larger than the fine.

To fix the number of iterations to a meaningful value which guarantees comparable accuracy from Parareal and serial fine integrator [27], we estimate the discretisation error of $\mathcal{F}_{\delta t}$ by comparing against a reference solution with time step $\delta t/10$. This gives estimates for the fine relative error at $T = 1$ of about $e_{\text{fine}} \approx 5.9 \times 10^{-5}$ and for the coarse error of about $e_{\text{coarse}} \approx 7.3 \times 10^{-2}$. For $P = 24$ time slices, after three iterations, the defect between Parareal and the fine solution is approximately 1.4×10^{-4}, after four iterations 1.5×10^{-5}. We therefore fix the number of iterations to $K = 4$ so that for all values of P Parareal produces a solution with the same accuracy as the fine integrator.

4.1. Hardware

Benchmarks are run on a single node of two different systems, respectively. The first one is CUB, a commodity Linux cluster at the Institute of Computational Science in Lugano, consisting of 3x14 IBM Blade nodes. Each node has two quad-core Opteron (Barcelona) CPUs, for a total of 8 cores per node, and 16 GigaByte main memory. Nodes are connected through an Infiniband interconnect. The used MPI implementation is OpenMPI-1.4.2[4], compiled with GCC-4.5.0[5] with flags `-O3` for maximum optimisation and `-fopenmp` to enable OpenMP. Note that, as the code is stand alone, no external libraries have to be linked.

`ORDERED` directive is required to guarantee a serialised execution of the update step and this directive is only available as part of a parallelised loop. Therefore, to keep the code consistent, the prediction step is also written as a parallel loop.

[4]`http://www.open-mpi.org/`

[5]`https://gcc.gnu.org/`

Algorithm 3: Parareal with pipelining using OpenMP

```
input: Initial value $q_0$; number of iterations $K$
3.1  $q(0) \leftarrow q_0$
3.2  OMP PARALLEL
3.3  $N$ = OMP_GET_MAX_THREADS()
3.4  OMP DO
3.5  for $p = 0, N-1$ do
3.6      $q(p) \leftarrow q(0)$
3.7      if Thread not first then
3.8          $q(p) \leftarrow \mathcal{G}(q(p), t_p, 0)$
3.9      end
3.10     $g_c(p) \leftarrow \mathcal{G}(q(p), t_{p+1}, t_p)$
3.11 end
3.12 OMP END DO NOWAIT
3.13 for $k = 1, K$ do
3.14     OMP DO ORDERED
3.15     for $p = 0, N-1$ do
3.16         OMP_SET_LOCK(p)
3.17         $q(p) \leftarrow \mathcal{F}_{\delta t}(q(p), t_{p+1}, t_p)$
3.18         $\delta q(p) \leftarrow q(p) - q_c(p)$
3.19         OMP_UNSET_LOCK(p)
3.20         OMP_ORDERERD
3.21         if Thread is first then
3.22             OMP_SET_LOCK(0)
3.23             $q(0) \leftarrow q_0$
3.24             OMP_UNSET_LOCK(0)
3.25         end
3.26         $q_c(p) \leftarrow \mathcal{G}_{\Delta t}(q(p), t_{p+1}, t_p)$
3.27         if Thread not last then
3.28             OMP_SET_LOCK(p+1)
3.29             $q(p+1) \leftarrow q_c(p) + \delta q(p)$
3.30             OMP_UNSET_LOCK(p+1)
3.31         end
3.32         OMP END ORDERED
3.33     end
3.34     OMP END DO NOWAIT
3.35 end
3.36 OMP END PARALLEL
```

The second system is Piz Dora at the Swiss National Supercomputing Centre.[6] Dora is a Cray XC40 with a total of 1,256 compute nodes. Each node contains two 12-core Intel Haswell CPUs and has 64 or 128 GigaByte of RAM and nodes are connected through a Cray Aries interconnect, using a dragonfly network topology. Two compilers are tested, the GCC-4.9.2 and the Cray Fortan compiler version 8.3.12. Both use the MPICH MPI library version 7.2.2. As on cub, compiler flags `-O3` and `-fopenmp` (GCC) or `omp` (Cray compiler) are used. Performance data for each completed job is generated using the Cray *Resource Utilisation Reporting* tool RUR [41]. RUR collects compute node statistics before and after each job and provides data on user and system time, maximum memory used, amount of I/O operations, consumed energy and other metrics. However, it only collects data for a full node and not for individual CPUs or cores.

[6]Detailed specification can be found here `http://www.cscs.ch/computers/piz_daint/index.html`

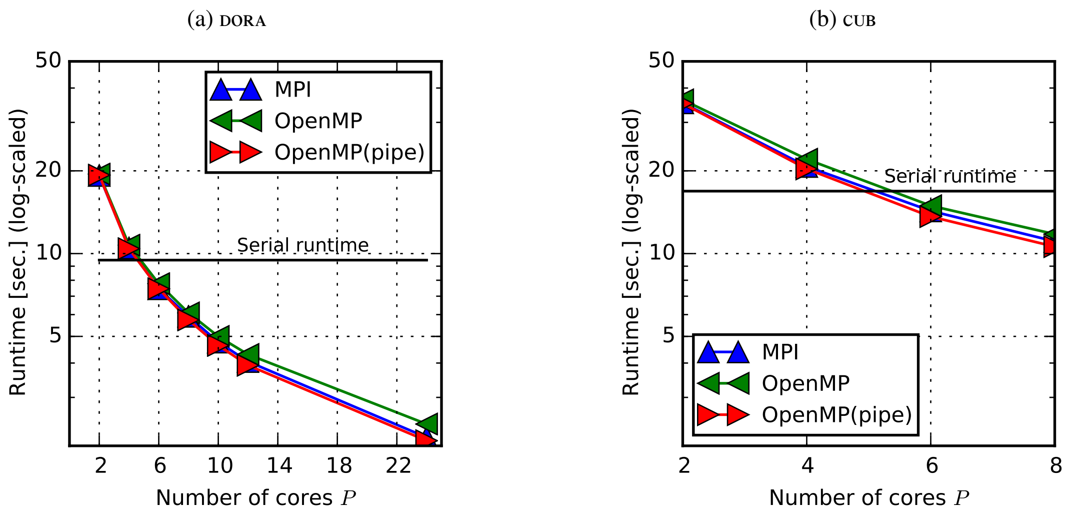


Figure 4: Five run averages of runtime with relative standard deviation below 0.01 on DORA (4a) and 0.025 on CUB (4b).

4.2. *Wall clock time and speedup*

At first, runtime and speedup compared to the serial execution of the fine integrator are assessed for the three different versions of Parareal. On both Cub and Dora, for each variant of Parareal and each value of P, five runs are performed and the average runtime is reported here. Measured runtimes are quite stable across different runs: the largest relative standard deviation of all performed five-run ensembles is smaller than 0.025 on CUB and smaller than 0.01 on DORA. Therefore, plots show only the average values without error bars, because they are hardly recognisable and clutter the figure.

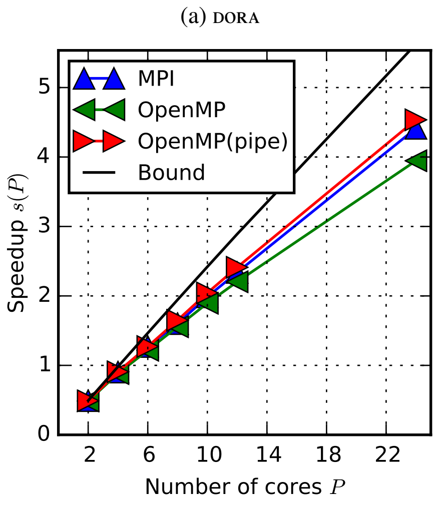
Figure 4 shows runtimes in seconds depending on the number of cores on Dora (left) and CUB (right). For Dora, only results from the Cray compiler are shown, which tends to generate slightly faster code than the GCC, but speedups for both compilers are very similar. Values from runs using MPI are marked by blue upward pointing triangles, from runs using OpenMP and no pipelining by leftward pointing green triangles and from runs using OpenMP and pipelining by rightward pointing red triangles. The runtime of the serial fine integrator is indicated by a horizontal black line. Note that the y axis is scaled logarithmically, so the distance from 5 to 10 seconds is the same as from 10 to 20. Because the more modern CPUs on DORA are faster, runtimes are generally smaller on DORA than on CUB. For $P = 8$, for example, Parareal runtimes on CUB are around 10 seconds but only around 7 seconds on DORA.

Both OpenMP and MPI with pipelining give almost identical performance: on both DORA and CUB, OpenMP(pipe) is only minimally faster than the MPI version. The missing pipelining in the simple OpenMP version leads to a small performance decrease, particularly for larger core counts on DORA. Note that as pipelining hides some of the cost of the coarse propagator, in a scenario with a more expensive $\mathcal{G}_{\Delta t}$, the performance penalty would be more pronounced, see also the discussion in Subsection 2.2.

Figure 5 shows speedup over the serial fine integrator, including the theoretical bound $s_{\mathrm{p}}(P)$ for Parareal with pipelining given in Equation (7) as a black line. On both machines (DORA and CUB) all configurations fall below the theoretically possible speedup, particularly when almost the full node is used. The reason is that running P instances of the fine propagator on P cores does not exactly take the same amount of time as running one instance of $\mathcal{F}$ on a single core, but is more costly – because the used spatial parallelisation is based on finite difference stencils and thus likely memory bound, the most probable reason seems to be memory access congestion. However, this affects both MPI and OpenMP alike and the conclusion here is that there is very little difference in terms of runtimes and speedup between using MPI and OpenMP for Parareal with pipelining.

4.3. *Memory footprint*

The memory footprint of the code is measured only on DORA where RUR is available. In contrast to runtime and energy, the memory footprint, as expected, does not vary between runs. Therefore Figure 6 shows a visualisation of



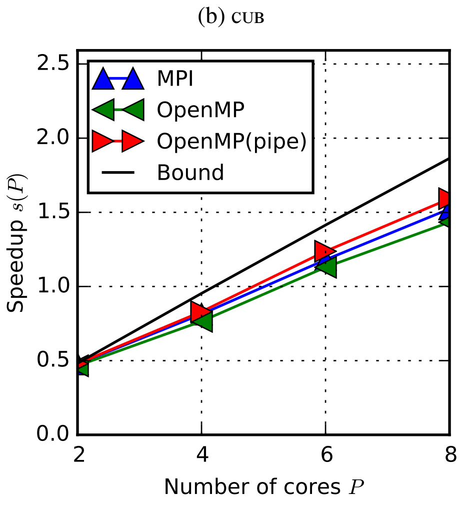


Figure 5: Speedup computed from average runtimes shown in Figure 4 on DORA (5a) and CUB (5b).

the data from a single run with no averaging. The bars indicate the maximum required memory in MegaByte while the black line indicates the expected memory consumption using P cores computed as

$$m(P) = P \times m_{\text{serial}} \tag{9}$$

where m_{serial} is the value measured for a reference run of the fine integrator. Because copies of the solution have to be stored for every time slice, the total memory required for Parareal can be expected to increase linearly with the number of cores in time. Note, however, that memory required *per core* stays constant if it follows (9).

For the GCC compiler, this is indeed the case for the OpenMP variants, both of which exactly match the expected values. The Cray compiler leads to a smaller than expected memory footprint, but memory requirements still increase linearly with the number of time slices. For both compilers, the MPI version causes a noticeable overhead in terms of memory footprint, most likely because of internal allocation of additional buffers for sending and receiving [42]. Interestingly, the total memory footprint for the MPI version is *larger* for the Cray than for the GCC compiler.

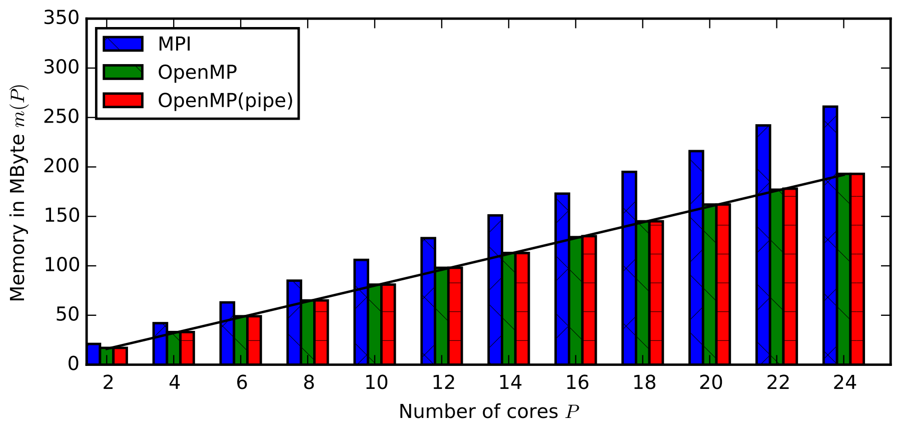
It is important to note that the three FORTRAN modules implementing the different Parareal versions (`parareal_mpi`, `parareal_openmp` and `parareal_openmp_pipe`, cf. Figure 3) all allocate three auxiliary buffers. The overhead in terms of memory in MPI does thus not simply stem from allocating an additional buffer for communication, but comes from somewhere within the MPI library. The OpenMP implementations provide direct access through shared memory and can thus avoid this overhead. Given that memory will be a much more precious resource on future supercomputers and that the memory requirements of "across-the-steps" parallel-in-time methods have raised concerns [34], these savings might be important. Since the memory overhead grows as more and more cores are used in time, savings from OpenMP will be especially pronounced when Parareal is used with many time slices on nodes with a large number of cores.

4.4. Energy-to-solution

The RUR tool also reports the energy-to-solution for every completed job. Because RUR can only report energy usage for a full node, only results are reported here from runs using the full number of $P = 24$ cores available on a DORA node. In contrast to runtimes and memory footprint, energy measurements show significant variations between runs due to random fluctuations: thus, the presented values are averages over ensembles of 50 runs for each version of Parareal. This number of runs has been sufficient to reduce the relative standard deviation to below 0.09 in all three configurations (MPI, OpenMP, OpenMP(pipe)) and therefore gives a robust indication of actual energy requirements.

For the GCC, the MPI version consumes the most energy, followed by OpenMP without pipelining, while OpenMP with pipelining is the most energy efficient variant: the resulting 95% confidence intervals (assuming energy-to-solution is normally distributed), are 844.04 ± 15.89 Joule for MPI, 801.14 ± 13.18 Joule for OpenMP without pipelining and 783.72 ± 11.53 Joule for OpenMP with pipelining. Figure 7 gives a graphical representation of these values

(a) GCC compiler



(b) Cray compiler

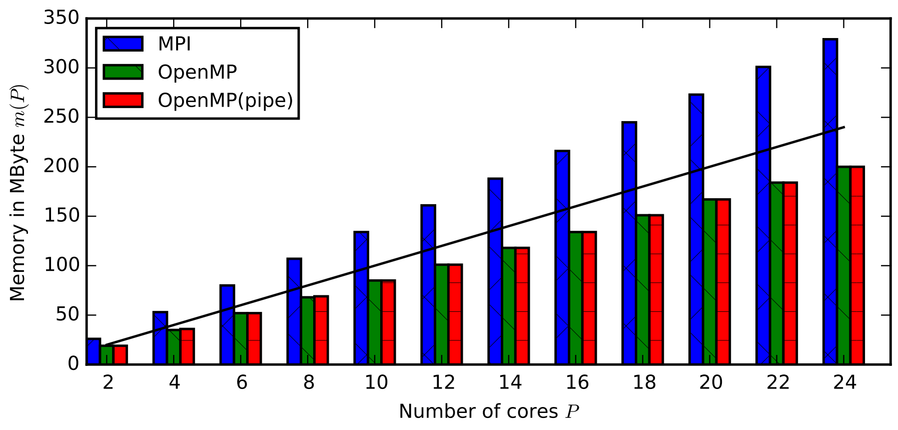
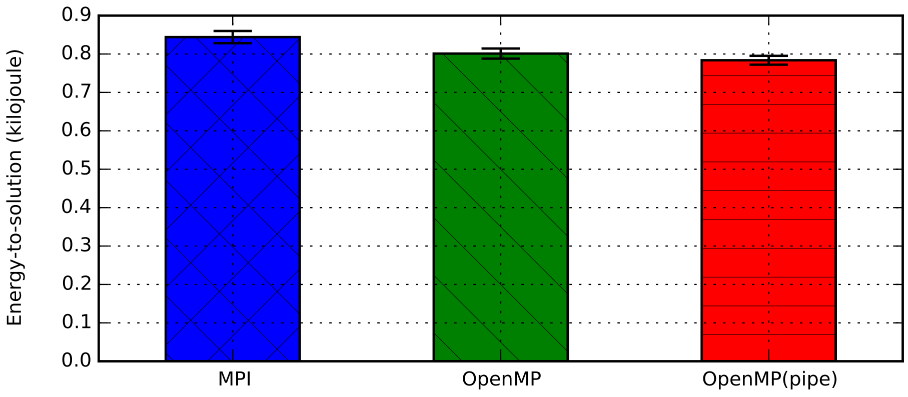


Figure 6: Maximum memory allocated in MegaByte for GCC (6a) and Cray compiler (6b) for the three different versions of Parareal depending on the number of used cores P. The black line indicates expected memory consumption computed as number of cores time memory footprint of serial fine integrator.

including the confidence intervals. Note that the average for OpenMP(pipe) is well outside the confidence interval for the MPI version, so this is very unlikely just a chance result. Moreover, because runtimes for both MPI and OpenMP(pipe) are almost identical, the differences in energy-to-solution cannot simply be attributed to differences in runtime. The lower power uptake by OpenMP without pipelining is most likely because most of the cores of the node are idle over large parts of the code, compare for Subsection 3.2. It is also noteworthy that OpenMP without pipelining, even though it is *slower* than the MPI version, still requires less energy: this further reinforces that the better energy efficiency of the OpenMP versions of Parareal is due to some intrinsic cause. Tracking down the precise reason for the differences in energy-to-solution and power requirement will require detailed tracing of power uptake, which is only possible on specially prepared machines [43] and thus left for future work.

Interestingly, for code generated with the Cray compiler, both OpenMP(pipe) and MPI lead to almost identical energy requirements: here, confidence intervals are 784.24 ± 11.93 Joule for MPI, 833.12 ± 20.99 Joule for OpenMP without and 784.24 ± 12.18 Joule for OpenMP with pipelining. Most likely, the compiler optimises the message passing to take advantage of the shared memory on the single node. Supposedly, the MPI version handles communication in a similar way as what is explicitly coded in the OpenMP(pipe) version. However, as shown in Subsection 4.3, this

(a) GCC compiler



(b) Cray compiler

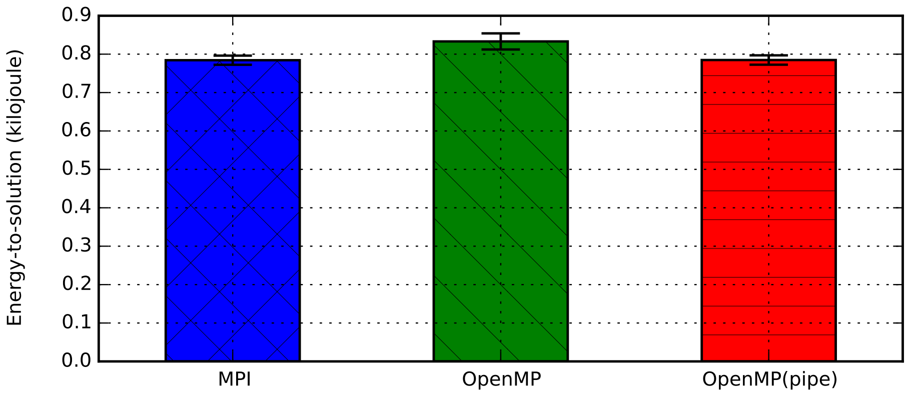


Figure 7: Energy-to-solution in Kilojoule for GCC in (7a) and Cray compiler (7b) for the three versions of Parareal, averaged across 50 samples, each using all 24 cores of the node. The largest relative standard deviation is 0.07, error bars indicate 95% confidence intervals.

automatic optimisation comes at the expense of a significantly larger memory footprint.

Note that the energy consumption of Parareal has previously been studied [27]. By comparing against a simple theoretical model, it has been shown that the energy overhead of Parareal (defined as energy-to-solution of Parareal divided by energy-to-solution of the fine serial integrator), is mostly due to Parareal's intrinsic suboptimal parallel efficiency. Since the GCC compiler has been used there, the measurements presented here refine this result by showing that a portion of the overhead is also likely to come from the use of MPI to communicate data in time. While improving parallel efficiency of parallel-in-time integration remains the main avenue for improving energy efficiency, the results here suggest that in some cases a shared memory approach can provide non-trivial additional savings.

5. Summary, Discussion and Outlook

Summary. The paper compares three different implementations of the parallel-in-time integration method Parareal. As benchmark, a stand-alone lightweight FORTRAN code is used, which solves 3D Burgers' equation with a fixed set of discretisations for both coarse and fine integrator [31]. The code provides three different versions of Parareal: one with pipelining using MPI, one with pipelining using OpenMP and one without pipelining using OpenMP. Because pipelining for Parareal comes naturally with MPI, the pipelined OpenMP implementation of Parareal is introduced here to allow for a fair comparison. Only the implementations of Parareal are different, though, and all three versions use the same modules to provide the coarse and fine integrator and underlying spatial discretisation.

The three versions are compared with respect to runtime and speedup on two different systems, a Linux cluster and a Cray XC 40 and, on the latter, using two different compilers. Differences in runtime between the MPI and OpenMP with pipelining are found to be small on both systems: the here proposed strategy for a shared memory version of Parareal with manual control of locks is minimally faster than its MPI counterpart. Using measurements provided through *resource utilisation reporting* on the Cray, memory footprint and energy-to-solution are also compared between MPI and OpenMP. It is demonstrated that the pipelined shared memory version is the most efficient: it has a significantly smaller memory footprint and, for the GCC compiler, requires less energy-to-solution.

Discussion. While the results presented here, investigating performance of pure parallelisation in time on a single compute node, clearly illustrate the possible benefits of using shared memory in time, there are a couple of possible pitfalls to consider in the generalisation of these results. Firstly, the manual control of synchronisation between threads and the need to deal with issues like non-uniform memory access (NUMA) make the pipelined OpenMP version more involved to implement. While this is not a big issue for the lightweight benchmark code used here, it might lead to unfeasible levels of complexity when integrating a parallel-in-time integrator into a complex legacy code. In contrast, for MPI, both the parallel-in-time methods Parareal and PFASST have been successfully used with such legacy codes [2, 24]. Nevertheless, as parallel-in-time integration becomes more widely used, new software projects will most certainly emerge that are from scratch designed to exploit space-time parallelism – here, considering shared memory time parallelisation for optimised performance could be a viable strategy.

Secondly, the OpenMP implementation of Parareal requires thread-safe routines for the coarse and fine integrator. In actual applications, parallelisation in time will almost always be considered *on top* of a large-scale spatial parallelisation, typically relying on MPI. For Parareal without pipelining, MPI communication in space can be funnelled through the `MASTER` thread [25], in order to avoid the necessity of thread-safe MPI. When combining shared memory Parareal with pipelining with spatial MPI, however, there is no way around using a fully thread compliant MPI library, corresponding to the level provided by the flag `MPI_THREAD_MULTIPLE` in the initialisation routine. While implementations of MPI exist that provide this level of thread-safety, the performance of such a space-time parallel code is hard to gauge a priori.

Thirdly, using shared memory to parallelise in time limits the number of cores in the time direction to the number available in a single computer node. Given the trend to nodes containing more and more cores (consider e.g. the Xeon Phi) and the fact that parallel-in-time methods typically suffer from degrading convergence if the time domain is distributed over too many cores, this is probably not going to be a very serious issue, but still mentioned for the sake of completeness.

Outlook. An important next step would be to make a similar comparison while also including spatial parallelisation. In particular, this would allow to assess whether the good performance of OpenMP-based Parareal is retained in combination with MPI-based parallelisation in space. Also, rerunning the here presented benchmarks on a special system equipped to perform detailed profiling of energy consumption would allow to more precisely determine the origin of the observed differences in energy-to-solution. Because the code used here is freely available, it (or its components) could also serve as a basis for further comparisons between methods or different libraries. For example, its time integrators could be included in other implementations of Parareal like e.g. LIB4PRM [38] and compared to the results presented here. They could also be included in other methods with similar interfaces like e.g. MGRIT [17]. Finally, the modules implementing the spatial routines could be integrated into libraries for other parallel-in-time methods, e.g. PFASST++ [6], in order to attempt a fair comparison between different methods.

Acknowledgments

I would like to thank the Centre for Interdisciplinary Research (ZIF) at the University of Bielefeld, Germany, for inviting me for a research visit in August 2015. The writing of this article greatly benefited from the tranquil and productive atmosphere at ZIF. I gratefully acknowledge support by Andrea Arteaga with the energy measurements.

References

[1] J. Dongarra, P. Beckman, al., The international exascale software roadmap, Int. Journal of High Performance Computer Applications 25 (1). doi:10.1177/1094342010391989.

[2] R. Speck, D. Ruprecht, R. Krause, M. Emmett, M. L. Minion, M. Winkel, P. Gibbon, A massively space-time parallel N-body solver, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 92:1–92:11. doi:10.1109/SC.2012.6.

[3] D. Ruprecht, R. Speck, M. Emmett, M. Bolten, R. Krause, Poster: Extreme-scale space-time parallelism, in: Proceedings of the 2013 Conference on High Performance Computing Networking, Storage and Analysis Companion, SC '13 Companion, 2013.
URL http://sc13.supercomputing.org/sites/default/files/PostersArchive/tech_posters/post148s2-file3.pdf

[4] XBraid: Parallel multigrid in time, http://llnl.gov/casc/xbraid.

[5] RIDC, https://github.com/ongbw/ridc.

[6] M. Emmett, D. Ruprecht, R. Speck, S. Terzi, PFASST++: MPI Bugfix (v0.5.0) (2015). doi:10.6084/m9.figshare.1431794.

[7] J. Nievergelt, Parallel methods for integrating ordinary differential equations, Commun. ACM 7 (12) (1964) 731–733. doi:10.1145/355588.365137.

[8] W. L. Miranker, W. Liniger, Parallel methods for the numerical integration of ordinary differential equations, Mathematics of Computation 21 (99) (1967) 303–320. doi:10.1090/S0025-5718-1967-0223106-8.

[9] J.-L. Lions, Y. Maday, G. Turinici, A "parareal" in time discretization of PDE's, Comptes Rendus de l'Acadmie des Sciences - Series I - Mathematics 332 (2001) 661–668. doi:10.1016/S0764-4442(00)01793-6.

[10] P. Chartier, B. Philippe, A parallel shooting technique for solving dissipative ODE's, Computing 51 (3-4) (1993) 209–236. doi:10.1007/BF02238534.

[11] M. Kiehl, Parallel multiple shooting for the solution of initial value problems, Parallel Computing 20 (3) (1994) 275–295. doi:10.1016/S0167-8191(06)80013-X.

[12] C. Farhat, 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) (2003) 1397–1434. doi:10.1002/nme.860.

[13] A. J. Christlieb, C. B. Macdonald, B. W. Ong, Parallel high-order integrators, SIAM Journal on Scientific Computing 32 (2) (2010) 818–835. doi:10.1137/09075740X.

[14] M. L. Minion, S. A. Williams, Parareal and spectral deferred corrections, in: AIP Conference Proceedings, Vol. 1048, 2008, p. 388. doi:10.1063/1.2990941.

[15] M. L. Minion, A Hybrid Parareal Spectral Deferred Corrections Method, Communications in Applied Mathematics and Computational Science 5 (2) (2010) 265–301. doi:10.2140/camcos.2010.5.265.

[16] M. Emmett, M. L. Minion, Toward an Efficient Parallel in Time Method for Partial Differential Equations, Communications in Applied Mathematics and Computational Science 7 (2012) 105–132. doi:10.2140/camcos.2012.7.105.

[17] R. D. Falgout, S. Friedhoff, T. V. Kolev, S. P. MacLachlan, J. B. Schroder, Parallel time integration with multigrid, SIAM Journal on Scientific Computing 36 (2014) C635–C661. doi:10.1137/130944230.

[18] M. J. Gander, M. Neumueller, Analysis of a Time Multigrid Algorithm for DG-Discretizations in Time (2014).
URL http://arxiv.org/abs/1409.5254

[19] M. J. Gander, M. Neumueller, Analysis of a new space-time parallel multigrid algorithm for parabolic problems (2014).
URL http://arxiv.org/abs/1411.0519

[20] M. J. Gander, 50 years of Time Parallel Time Integration, in: Multiple Shooting and Time Domain Decomposition, Springer, 2015.
URL http://www.unige.ch/%7Egander/Preprints/50YearsTimeParallel.pdf

[21] M. J. Gander, S. Vandewalle, On the Superlinear and Linear Convergence of the Parareal Algorithm, in: O. B. Widlund, D. E. Keyes (Eds.), Domain Decomposition Methods in Science and Engineering, Vol. 55 of Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, 2007, pp. 291–298. doi:10.1007/978-3-540-34469-8_34.

[22] D. Ruprecht, R. Krause, Explicit parallel-in-time integration of a linear acoustic-advection system, Computers & Fluids 59 (0) (2012) 72–83. doi:10.1016/j.compfluid.2012.02.015.

[23] J. M. F. Trindade, J. C. F. Pereira, Parallel-in-Time Simulation of Two-Dimensional, Unsteady, Incompressible Laminar Flows, Numerical Heat Transfer, Part B: Fundamentals 50 (1) (2006) 25–40. doi:10.1080/10407790500459379.

[24] R. Croce, D. Ruprecht, R. Krause, Parallel-in-Space-and-Time Simulation of the Three-Dimensional, Unsteady Navier-Stokes Equations for Incompressible Flow, in: H. G. Bock, X. P. Hoang, R. Rannacher, J. P. Schlder (Eds.), Modeling, Simulation and Optimization of Complex Processes – HPSC 2012, Springer International Publishing, 2014, pp. 13–23. doi:10.1007/978-3-319-09063-4_2.

[25] R. Krause, D. Ruprecht, Hybrid Space-Time Parallel Solution of Burgers' Equation, in: Domain Decomposition Methods in Science and Engineering XXI, Vol. 98 of Lecture Notes in Computational Science and Engineering, Springer International Publishing, 2014, pp. 647–655. doi:10.1007/978-3-319-05789-7_62.

[26] R. D. Haynes, B. W. Ong, MPI-OpenMP algorithms for the parallel space-time solution of time dependent PDEs, in: Domain Decomposition Methods in Science and Engineering XXI, Vol. 98 of Lecture Notes in Computational Science and Engineering, Springer International Publishing, 2014, pp. 179–187. doi:10.1007/978-3-319-05789-7_14.

[27] A. Arteaga, D. Ruprecht, R. Krause, A stencil-based implementation of Parareal in the C++ domain specific embedded language STELLA, Applied Mathematics and Computation 267 (2015) 727–741. doi:10.1016/j.amc.2014.12.055.

[28] E. Aubanel, Scheduling of Tasks in the Parareal Algorithm, Parallel Computing 37 (2011) 172–182. doi:10.1016/j.parco.2010.10.004.

[29] H. Xiao, E. Aubanel, Scheduling of Tasks in the Parareal Algorithm for Heterogeneous Cloud Platforms, in: Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International, 2012, pp. 1440–1448. doi:10.1109/IPDPSW.2012.181.

[30] L. A. Berry, W. R. Elwasif, J. M. Reynolds-Barredo, D. Samaddar, R. S. Snchez, D. E. Newman, Event-based parareal: A data-flow based implementation of parareal, Journal of Computational Physics 231 (17) (2012) 5945–5954. doi:10.1016/j.jcp.2012.05.016.

[31] D. Ruprecht, PararealF90: Implementing Parareal – OpenMP or MPI?, release v1.0 (2015). doi:10.5281/zenodo.31288.

[32] C.-W. Shu, S. Osher, Efficient implementation of essentially non-oscillatory shock-capturing schemes II, Journal of Computational Physics 83 (1989) 32–78. doi:10.1016/0021-9991(89)90222-2.

[33] T. Gysi, O. Osuna, Carlos anad Fuhrer, M. Bianco, T. C. Schulthess, Stella: A domain-specific tool for structured grid methods in weather and climate models, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2015.

[34] D. E. Keyes, Exaflop/s: The why and the how, Comptes Rendus Mécanique 339 (23) (2011) 70 – 77. doi:10.1016/j.crme.2010.11.002.

[35] K. Burrage, Parallel methods for ODEs, Advances in Computational Mathematics 7 (1997) 1–3. doi:10.1023/A:1018997130884.

[36] M. Schreiber, A. Peddle, T. Haut, B. Wingate, A decentralized parallelization-in-time approach with parareal (2015).
URL http://arxiv.org/abs/1506.05157

[37] D. Ruprecht, Convergence of Parareal with spatial coarsening, PAMM 14 (1) (2014) 1031–1034. doi:10.1002/pamm.201410490.

[38] A. Kreienbuehl, A. Naegel, D. Ruprecht, R. Speck, G. Wittum, R. Krause, Numerical simulation of skin transport using Parareal, Computing and Visualization in Sciencedoi:10.1007/s00791-015-0246-y.

[39] M. Hermanns, Parallel programming in Fortran 95 using OpenMP, http://www.openmp.org/presentations/miguel/F95_OpenMPv1_v2.pdf.

[40] M. J. Gander, M. Petcu, Analysis of a Krylov Subspace Enhanced Parareal Algorithm for Linear Problem, ESAIM: Proc. 25 (2008) 114–129. doi:10.1051/proc:082508.

[41] A. Barry, Resource utilization reporting: Gathering and evaluating HPC system usage, in: CUG2013 Proceedings, 2013.
URL https://cug.org/proceedings/cug2013_proceedings/includes/files/pap103-file2.pdf

[42] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: 17th Euromicro International Conference on Parallel, Distributed and Network-based processing, 2009, pp. 427–436. doi:10.1109/PDP.2009.43.

[43] C. Isci, M. Martonosi, Runtime power monitoring in high-end processors: Methodology and empirical data, in: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, IEEE Computer Society, Washington, DC, USA, 2003, pp. 93–.
URL http://dl.acm.org/citation.cfm?id=956417.956567