Using performance analysis tools for a parallel-in-time integrator

Does my time-parallel code do what I think it does?

Robert Speck[1], Michael Knobloch[1], Sebastian Lührs[1], and Andreas Gocht[2]

[1] Jülich Supercomputing Centre
Forschungszentrum Jülich GmbH
52425 Jülich, Germany
{m.knobloch, s.luehrs, r.speck}@fz-juelich.de
[2] Center of Information Services and High Performance Computing
Zellescher Weg 12
01069 Dresden, Germany
andreas.gocht@tu-dresden.de

Abstract. While many ideas and proofs of concept for parallel-in-time integration methods exists, the number of large-scale, accessible time-parallel codes is rather small. This is often due to the apparent or subtle complexity of the algorithms and the many pitfalls awaiting developers of parallel numerical software. One example of such a time-parallel code is `pySDC`, which implements, among others, the parallel full approximation scheme in space and time (PFASST). Inspired by nonlinear multigrid ideas, PFASST allows to integrate multiple time-steps simultaneously using a space-time hierarchy of spectral deferred corrections. In this paper we demonstrate the application of performance analysis tools to the PFASST implementation `pySDC`. We trace the path we took for this work, show examples of how the tools can be applied and explain the sometimes surprising findings we encountered. Although focusing only on a single implementation of a particular parallel-in-time integrator, we hope that our results and in particular the way we obtained them are a blueprint for other time-parallel codes.

1 Motivation

With million-way concurrency at hand, the efficient use of modern high-performance computing systems has become one of the key challenges in computational science and engineering. New mathematical concepts and algorithms are needed to fully exploit these massively parallel architectures. For the numerical solution of time-dependent processes, recent developments in the field of parallel-in-time integration have opened new ways to overcome both strong and weak scaling limit of classical, spatial parallelization techniques. In [14], many of these techniques and their properties are presented, while [32] gives an overview of applications of parallel-in-time integration. Furthermore, the community website[3] provides a comprehensive list of references. We refer to these

[3] `https://www.parallel-in-time.org`

sources for a detailed overview of time-parallel methods and their applications. While many ideas, algorithms and proofs of concept exist in this domain, the number of actual large-scale time-parallel application codes or even stand-alone parallel-in-time libraries showcasing performance gains is still small. In particular, codes which can deal with parallelization in time as well as in space are rare. At the time of this writing, three main, accessible projects targeting this area are `XBraid`, a C/C++ time-parallel multigrid solver [26], `RIDC`, a C++ implementation of the revisionist integral deferred correction method [31], and at least two different implementations of PFASST, the "parallel full approximation scheme in space and time" [10]. One major PFASST implementation is written in Fortran (`libpfasst`, see [28]), another one in Python (`pySDC`, see [42]).

When running parallel simulations, benchmarks or just initial tests, one key question is whether the code actually does what it is supposed to do and/or what the developer thinks it does. While this may seem obvious to the developer, complex codes (like PFASST implementations) tend to introduce complex bugs. To avoid these, one may ask for example: How many messages were sent, how many were received? Is there a wait for each non-blocking communication? Are the number of solves/evaluations/iterations reasonable? Moreover, even if the workflow itself is correct and verified, the developer or user may wonder whether the code is as fast as it can be: Is the communication actually non-blocking or blocking, when it should be? Is the waiting time of the processes as expected? Does the algorithm spend reasonable time in certain functions or are there inefficient implementations causing delays? Then, if all runs well, performing comprehensive parameter studies like benchmarking requires a solid workflow management and it can be quite tedious to keep track of what ran where, when and with what result. In order to address questions like these, advanced performance analysis tools can be used.

The performance analysis tools landscape is manifold. Tools range from node-level analysis tools using hardware counters like LIKWID [44] and PAPI [43] to tools intended for large-scale, complex applications like Scalasca [16]. There are tools developed by the hardware vendors, e.g. Intel VTune [34] or NVIDIA nvprof [5] as well as community driven open source tools and tool-sets like Score-P [25], TAU [39] or HPCToolkit [1]. Choosing the right tool depends on the task at hand and of course on the familiarity of the analyst with the available tools.

It is the goal of this paper to present some of these tools and show their capabilities for performance measurements, workflows and bug detection for time-parallel codes like `pySDC`. Although we will, in the interest of brevity, solely focus on `pySDC` for this paper, our results and in particular the way we obtained them with the different tools can serve as a blueprint for many other implementations of parallel-in-time algorithms. While there are a lot of studies using these tools for many parallelization strategies, see e.g. [22,19], and application areas, see e.g. [38,18], their application in the context of parallel-in-time integration techniques is new. Especially when different parallelization strategies are mixed, these tools can provide invaluable help. We would like to emphasize that this paper is not about the actual results of `pySDC`, PFASST or parallel-in-time inte-

gration itself (like the application, the parallel speedup or the time-to-solution), but on the benefits of using performance tools and workflow managers for the development and application of a parallel-in-time integrator. Thus, this paper is meant as a community service to showcase what can be done with a few standard tools from the broad field of HPC performance analysis. One specific challenge in this regard, however, is the programming language of `pySDC`. Most tools focus on more standard HPC languages like Fortran or C/C++. With the new release of Score-P used for this work, Python codes can now be analyzed as well, as we will show in this paper.

In the next section we will briefly introduce the PFASST algorithm and describe its implementation in some detail. While the math behind a method may not be relevant for performance tools, understanding the algorithms at least in principle is necessary to give more precise answers to the questions the method developers may have. Section 3 is concerned with a more or less brief and high-level description of the performance analysis tools used for this project. Section 4 describes the endeavor of obtaining reasonable measurements from their application to `pySDC`, interpreting the results and learning from them. Section 5 contains a brief summary and an outlook.

2 A Parallel-in-Time Integrator

In this section we briefly review the collocation problem, being the basis for all problems the algorithm presented here tries to solve in one way or another. Then, spectral deferred corrections (SDC, [9]) are introduced, which lead to the time-parallel integrator PFASST. This section is largely based on [4,40].

2.1 Spectral deferred corrections

For ease of notation we consider a scalar initial value problem on the interval $[t_\ell, t_{\ell+1}]$

$$u_t = f(u), \quad u(t_\ell) = u_0,$$

with $u(t), u_0, f(u) \in \mathbb{R}$. We rewrite this in Picard formulation as

$$u(t) = u_0 + \int_{t_\ell}^{t} f(u(s))ds, \quad t \in [t_\ell, t_{\ell+1}].$$

Introducing M quadrature nodes $\tau_1, ..., \tau_M$ with $t_\ell \leq \tau_1 < ... < \tau_M = t_{\ell+1}$, we can approximate the integrals from t_ℓ to these nodes τ_m using spectral quadrature like Gauss-Radau or Gauss-Lobatto quadrature, such that

$$u_m = u_0 + \Delta t \sum_{j=1}^{M} q_{m,j} f(u_j), \quad m = 1, ..., M,$$

where $u_m \approx u(\tau_m)$, $\Delta t = t_{\ell+1} - t_\ell$ and $q_{m,j}$ represent the quadrature weights for the interval $[t_\ell, \tau_m]$ with

$$\Delta t \sum_{j=1}^{M} q_{m,j} f(u_j) \approx \int_{t_\ell}^{\tau_m} f(u(s))ds.$$

We can now combine these M equations into one system of linear or non-linear equations with

$$(\mathbf{I}_M - \Delta t \mathbf{Q} \boldsymbol{f})(\boldsymbol{u}_\ell) = \boldsymbol{u}_0 \tag{1}$$

where $\boldsymbol{u}_\ell = (u_1, ..., u_M)^T \approx (u(\tau_1), ..., u(\tau_M))^T \in \mathbb{R}^M$, $\boldsymbol{u}_0 = (u_0, ..., u_0)^T \in \mathbb{R}^M$, $\mathbf{Q} = (q_{i,j}) \in \mathbb{R}^{M \times M}$ is the matrix gathering the quadrature weights, $\mathbf{I}_M$ is the identity matrix of dimension M and the vector function $\boldsymbol{f}$ is given by $\boldsymbol{f}(\boldsymbol{u}) = (f(u_1), ..., f(u_M))^T \in \mathbb{R}^M$. This system of equations is called the "collocation problem" for the interval $[t_\ell, t_{\ell+1}]$ and it is equivalent to a fully implicit Runge-Kutta method, where the matrix $\mathbf{Q}$ contains the entries of the corresponding Butcher tableau. We note that for $f(u) \in \mathbb{R}^N$, we need to replace $\mathbf{Q}$ by $\mathbf{Q} \otimes \mathbf{I}_N$.

Using SDC, this problem can be solved iteratively and we follow [20,45,35] to present SDC as preconditioned Picard iteration for the collocation problem (1). The standard approach to preconditioning is to define an operator which is easy to invert but also close to the operator of the system. One very effective option is the so-called "LU trick", which uses the LU decomposition of $\mathbf{Q}^T$ to define

$$\mathbf{Q}_\Delta = \mathbf{U}^\mathbf{T} \quad \text{for} \quad \mathbf{Q}^T = \mathbf{L}\mathbf{U},$$

see [45] for details. With this we write

$$(\mathbf{I}_M - \Delta t \mathbf{Q}_\Delta \boldsymbol{f})(\boldsymbol{u}_\ell^{k+1}) = \boldsymbol{u}_0 + \Delta t(\mathbf{Q} - \mathbf{Q}_\Delta)\boldsymbol{f}(\boldsymbol{u}_\ell^k) \tag{2}$$

or, equivalently,

$$\boldsymbol{u}_\ell^{k+1} = \boldsymbol{u}_0 + \Delta t \mathbf{Q}_\Delta \boldsymbol{f}(\boldsymbol{u}_\ell^{k+1}) + \Delta t(\mathbf{Q} - \mathbf{Q}_\Delta)\boldsymbol{f}(\boldsymbol{u}_\ell^k) \tag{3}$$

and the operator $\mathbf{I} - \Delta t \mathbf{Q}_\Delta \boldsymbol{f}$ is then called the SDC preconditioner. Writing (3) line by line recovers the classical SDC formulation found in [9].

2.2 Parallel full approximation scheme in space and time

We can assemble the collocation problem (1) for multiple time-steps, too. Let $\boldsymbol{u}_1, ..., \boldsymbol{u}_L$ be the solution vectors at time-steps $1, ..., L$ and $\vec{\boldsymbol{u}} = (\boldsymbol{u}_1, ..., \boldsymbol{u}_L)^T$ the full solution vector. We define a matrix $\mathbf{H} \in \mathbb{R}^{M \times M}$ such that $\mathbf{H}\boldsymbol{u}_\ell$ provides the initial value for the $\ell + 1$-th time-step. Note that this initial value has to be used at all nodes, see the definition of $\boldsymbol{u}_0$ above. The matrix depends on the collocation nodes and if the last node is the right interval boundary, i.e. $\tau_M = t_{\ell+1}$ as it is the case for Gauss-Radau or Gauss-Lobatto nodes, then it is simply given by

$$\mathbf{H} = (0, ..., 0, 1) \otimes (1, ..., 1)^T$$

Otherwise, $\mathbf{H}$ would contain weights for extrapolation or the collocation formula for the full interval. Note that for $f(u) \in \mathbb{R}^N$, we again need to replace $\mathbf{H}$ by $\mathbf{H} \otimes \mathbf{I}_N$. With this definition, we can assemble the so-called "composite collocation problem" for L time-steps as

$$\mathbf{C}(\vec{\boldsymbol{u}}) := (\mathbf{I}_{LM} - \mathbf{I}_L \otimes \Delta t \mathbf{Q} \boldsymbol{F} - \mathbf{E} \otimes \mathbf{H})(\vec{\boldsymbol{u}}) = \vec{\boldsymbol{u}}_0, \tag{4}$$

with $\vec{\boldsymbol{u}}_0 = (\boldsymbol{u}_0, \mathbf{0}, ..., \mathbf{0})^T \in \mathbb{R}^{LM}$, the vector of vector functions $\vec{\boldsymbol{F}}(\vec{\boldsymbol{u}}) = (\boldsymbol{f}(\boldsymbol{u}_1), ..., \boldsymbol{f}(\boldsymbol{u}_L))^T \in \mathbb{R}^{LM}$ and where the matrix $\mathbf{E} \in \mathbb{R}^{L \times L}$ has ones on the lower sub-diagonal and zeros elsewhere, accounting for the transfer of the solution from one step to another.

For serial time-stepping each step can be solved after another, i.e. SDC iterations (now called "sweeps") are performed until convergence on $\boldsymbol{u}_1$, move to step 2 via $\mathbf{H}$, do SDC there and so on. In order to introduce parallelism in time, the "parallel full approximation scheme in space in time" (PFASST) makes use of a full approximation scheme (FAS) multigrid approach for solving (4). We present this idea using two levels only, but the algorithm can be easily extended to multiple levels. First, a parallel solver on the fine level and a serial solver on the coarse level are defined as

$$\begin{aligned}
\mathbf{P}_{\text{par}}(\vec{\boldsymbol{u}}) &:= (\mathbf{I}_{LM} - \mathbf{I}_L \otimes \Delta t \mathbf{Q}_\Delta \boldsymbol{F})(\vec{\boldsymbol{u}}), \\
\mathbf{P}_{\text{ser}}(\vec{\boldsymbol{u}}) &:= (\mathbf{I}_{LM} - \mathbf{I}_L \otimes \Delta t \mathbf{Q}_\Delta \boldsymbol{F} - \mathbf{E} \otimes \mathbf{H})(\vec{\boldsymbol{u}}).
\end{aligned}$$

Omitting the term $\mathbf{E} \otimes \mathbf{H}$ in $\mathbf{P}_{\text{par}}$ decouples the steps, enabling simultaneous SDC sweeps on each step.

PFASST uses $\mathbf{P}_{\text{par}}$ as smoother on the fine level and $\mathbf{P}_{\text{ser}}$ as an approximate solver on the coarse level. Restriction and prolongation operators $\mathbf{I}_h^H$ and $\mathbf{I}_H^h$ allow to transfer information between the fine level (indicated with h) and the coarse level (indicated with H). The approximate solution is then used to correct the solution of the smoother on the finer level. Typically, only two levels are used, although the method is not restricted to this choice. PFASST in its standard implementation allows coarsening in the degrees-of-freedom in space (i.e. use $N/2$ instead of N unknowns per spatial dimension), a reduced collocation rule (i.e. use a different $\mathbf{Q}$ on the coarse level), a less accurate solver in space (for solving (2) on each time-step) or even a reduced representation of the problem. The first two strategies directly influence the definition of the restriction and prolongation operators.

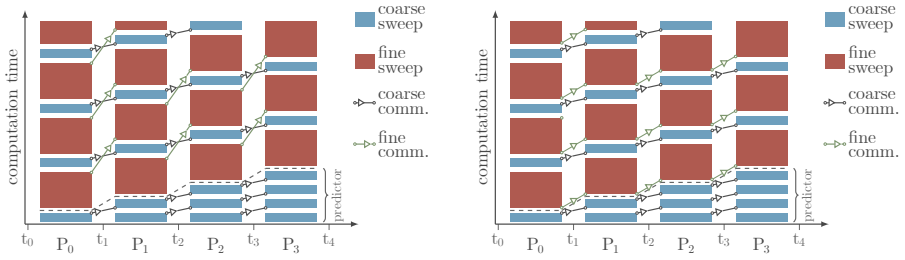
Since the right-hand side of the ODE can be a non-linear function, a τ-correction stemming from the FAS is added to the coarse problem. One PFASST iteration then comprises the following steps:

1. Compute τ-correction as

$$\vec{\boldsymbol{\tau}} = \mathbf{C}_H \left(\mathbf{I}_h^H \vec{\boldsymbol{u}}_h^k \right) - \mathbf{I}_h^H \mathbf{C}_h \left(\vec{\boldsymbol{u}}_h^k \right).$$

2. Compute $\vec{\boldsymbol{u}}_H^{k+1}$ from

$$\mathbf{P}_{\text{ser}}(\vec{\boldsymbol{u}}_H^{k+1}) = \vec{\boldsymbol{u}}_{0,H} + \vec{\boldsymbol{\tau}} + (\mathbf{P}_{\text{ser}} - \mathbf{C}_H)(\mathbf{I}_h^H \vec{\boldsymbol{u}}_h^k).$$



(a) Original algorithm with overlap as described in [10]

(b) Algorithm as described in [3] and implemented in `pySDC`

Fig. 1: Two slightly different workflows of PFASST, on the left with (theoretically) overlapping fine and coarse communication, on the right with multigrid-like communication.

3. Compute $\vec{u}_h^{k+1/2}$ from

$$\vec{u}_h^{k+1/2} = \vec{u}_h^k + \mathbf{I}_H^h \left(\vec{u}_H^{k+1} - \mathbf{I}_h^H \vec{u}_h^k \right).$$

4. Compute $\vec{u}_h^{k+1}$ from

$$\mathbf{P}_{\text{par}}(\vec{u}_h^{k+1}) = \vec{u}_{0,h} + (\mathbf{P}_{\text{par}} - \mathbf{C}_h)\,(\vec{u}_h^{k+1/2}).$$

We note that this "multigrid perspective" [3] does not represent the original idea of PFASST as described in [29,10]. There, PFASST is presented as a coupling of SDC with the time-parallel method Parareal, augmented by the τ-correction which allows to represent fine-level information on the coarse level.

While conceptually the same, there is a key difference in the implementation of these two representations of PFASST. The workflow of the algorithm is depicted in Figure 1, showing the original approach in 1a and the multigrid perspective in 1b. They differ in the way the fine level communication is done. As described in [11], under certain conditions it is possible to introduce overlap of sending/receiving updated values on the fine level and the coarse level computation. More precisely, the "window" for finishing fine level communication is as long as two coarse level sweeps: one from the current iteration, one from the predictor which already introduces a lag of later processors (see Figure 1a). In contrast, the multigrid perspective requires updated fine level values whenever the term $\mathbf{C}_h(\vec{u}_h^k)$ has to be evaluated. This is the case in step 1 and step 2 of the algorithm as presented before. Note that due to the serial nature of step 3, the evaluation of $\mathbf{C}_H(\mathbf{I}_h^H \vec{u}_h^{k+1/2})$ already uses the most recent values on the coarse level in both approaches. Therefore, overlap of communication and computation is somewhat limited: only during the time-span of a single coarse level sweep (introduced by the predictor) the fine level communication has to finish in order to avoid waiting times (see Figure 1b). However, the advantage of the multigrid perspective, besides its relative simplicity and ease of notation,

is that multiple sweeps on the fine level for speeding up convergence, as shown in [4], are now effectively possible. This is one of the reasons this implementation strategy has been chosen for `pySDC`, while the original Fortran implementation `libpfasst` uses the classical workflow. Yet, while the multigrid perspective may simplify the formal description of the PFASST algorithm, the implementation of PFASST can still be quite challenging.

2.3 pySDC

The purpose of the Python code `pySDC` is to provide a framework for testing, evaluating and applying different variants of SDC and PFASST without worrying too much about implementation details, communication structures or lower-level language peculiarities. Users can simply set up an ODE system and run standard versions of SDC or PFASST spending close to no thoughts on the internal structure. In particular, it provides an easy starting point to see whether collocation methods, SDC, and parallel-in-time integration with PFASST are useful for the problem under consideration. Developers, on the other hand, can build on the existing infrastructure to implement new iterative methods or to improve existing methods by overriding any component of `pySDC`, from the main controller and the SDC sweeps to the transfer routines or the way the hierarchy is created. `pySDC`'s main features are [40]:

- available implementations of many variants of SDC, MLSDC and PFASST,
- many ordinary and partial differential equations already pre-implemented,
- tutorials to lower the bar for new users and developers,
- coupling to FEniCS and PETSc, including spatial parallelism for the latter
- automatic testing of new releases, including results of previous publications
- full compatibility with Python 3.6+, runs on desktops and HPC machines

The main website for `pySDC`[4] provides all relevant information, including links to the code repository on Github, the documentation as well as test coverage reports. `pySDC` is also described in much more detail in [40].

The algorithms within `pySDC` are implemented using two "controller" classes. One emulates parallelism in time, while the other one uses `mpi4py` [7] for actual parallelization in the time dimension with the Message Passing Interface (MPI). Both can run the same algorithms and yield the same results, but while the first one is primarily used for theoretical purposes and debugging, the latter makes actual performance tests and time-parallel applications possible.

We will use the MPI-based controller for this paper in order to address the questions posed at the beginning. To do that, a number of HPC tools are available which helps users and developers of HPC software to evaluate the performance of their codes and to speed up their workflows.

[4] `https://www.parallel-in-time.org/pySDC`

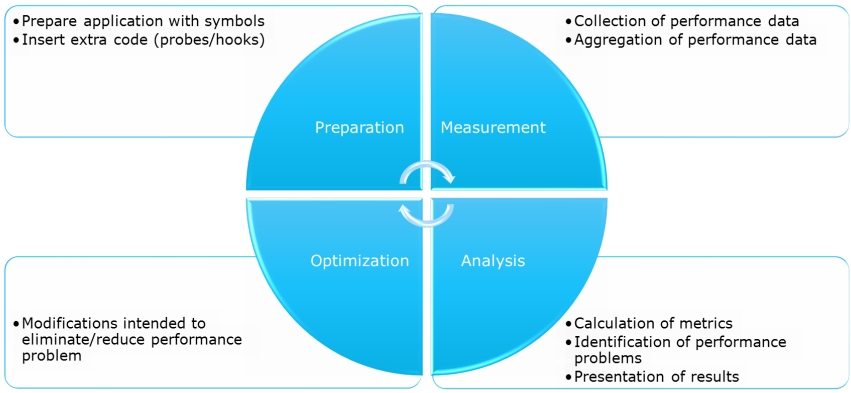


Fig. 2: Performance Engineering Workflow

3 Performance Analysis Tools

Performance analysis plays a crucial part in the development process of an HPC application. It usually starts with simply timing the computational kernels to see where the time is spent. To access more information and to determine tuning potential, more sophisticated tools are required. The typical performance engineering workflow when using performance analysis tools is an iterative process as depicted in Figure 2.

First, the application needs to be prepared and some hooks to the measurement system need to be added. These can be debug symbols, compiler instrumentation or even code changes by the user. Then, during execution of the application, performance data is collected and, if necessary, aggregated. The analysis tools then calculate performance metrics to pinpoint performance problems to the developer. Finally, the hardest part: the developer has to modify the application to eliminate or at least reduce the performance problems found by the tools, ideally without introducing new ones. Unfortunately, tools can only provide little help in this step.

Several performance analysis tools exist, for all kinds of measurement at all possible scales, from a desktop computer to the largest supercomputers in the world. We distinguish two major measurement techniques with different levels of accuracy and overhead – "profiling", which aggregates the performance metrics at runtime and presents statistical results, e.g. how often a function was called and how much time was spend there, and "event-based tracing", where each event of interest, like function enter/exit, messages sent/ received etc. are stored together with a timestamp. Tracing conserves temporal and spatial relationships of events and is the more general measurement technique, as a profile can always be generated from a trace. The main disadvantage of tracing is that trace files can quickly become extremely large (in the order of terabytes) when collecting every event. So usually the first step is a profile to determine the hot-spot of the

application, which then is analyzed in detail using tracing to keep trace-size and overhead manageable.

However, performance analysis tools cannot only be used to identify optimization potential but also to assess the execution of the application on a given system with a specific tool-chain (compiler, MPI library, etc.), i.e. to answer the question "Is my application doing what I think it is doing?". More often than not the answer to that question is "No", as it was in the case we present in this work. Tools can pinpoint the issues and help to identify possible solutions.

For our analysis we used the tools of the Score-P ecosystem, which are presented in this section. A similar analysis is possible with other tools as well, e.g. with TAU [39], Paraver [33], or Intels VTune [34].

3.1 Score-P and the Score-P ecosystem

The Score-P measurement infrastructure [25] is an open source, highly scalable and easy-to-use tool suite for profiling, event tracing, and online analysis of HPC applications. It is a community project to replace the measurement systems of several performance analysis tools and to provide common data formats to improve interoperability between different analysis tools built on top of Score-P. Figure 3 shows a schematic overview of the Score-P ecosystem. Most common HPC programming paradigms are supported by Score-P: MPI (via the PMPI interface), OpenMP (via OPARI2 or the OpenMP tools interface (OMPT) [13]) as well as GPU programming with CUDA, OpenACC or OpenCL. Score-P offers three ways to measure application events:

1. compiler instrumentation, where compiler interfaces are used to insert calls to the measurement system at each function enter and exit,
2. a user instrumentation API, that enables the application developer to mark specific regions, e.g. kernels, functions or even loops, and
3. a sampling interface which records the state of the program at specific intervals.

All this data is handled in the Score-P measurement core where it can be enriched with hardware counter information from PAPI [43], perf or rusage. Further, Score-P provides a counter plugin interface that enables the user to define its own metric sources. The Score-P measurement infrastructure supports two modes of operation, it can generate event traces in the OTF2 format [12] and aggregated profiles in the CUBE4 format [36].

Usage of Score-P is quite straightforward – the compile and link command have to be prepended by `scorep`, e.g. `mpicc app.c` becomes `scorep mpicc app.c`. However, Score-P can be extensively configured via environment variables, so that Score-P can be used in all analysis steps from a simple call-path profile to a sophisticated tracing experiment enriched with hardware counter information. Listing 3 in Section 4.2 will show an example job script where several Score-P options are used.

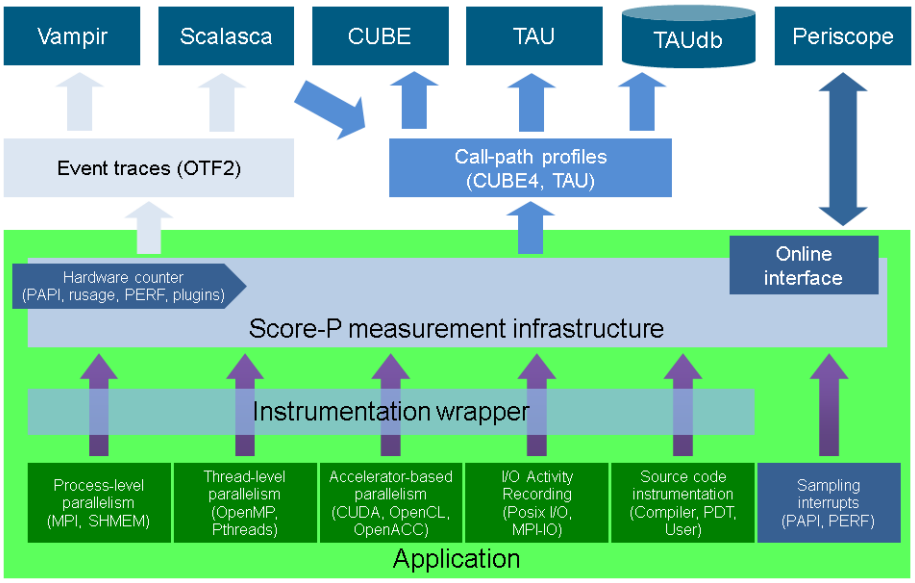


Fig. 3: Overview of the Score-P ecosystem. The green box represents the measurement infrastructure with the various ways of data acquisition. This data is processed by the Score-P measurement infrastructure and stored either aggregated in the CUBE4 profile format or as an event trace in the OTF2 format. On top are the various analysis tools working with these common data formats.

Score-P Python bindings Traditionally the main programming languages for HPC application development have been C, C++ and Fortran. However, with the advent of high-performance Python libraries in the wake of the rise of AI and deep learning, pure Python HPC applications are now a feasible possibility, as `pySDC` shows. Python has two built-in Performance Analysis Tools, called `profile` and `cProfile`. Though they allow profiling Python code, they do not support as detailed application analyses as Score-P does. Therefore, the Score-P Python bindings have been introduced [17], which allow to profile and trace Python applications using Score-P. This technique can analyze all different kinds of applications that use python, including machine learning workflows. This particular aspect will be described in more detail in another paper.

The bindings use the Python built-in infrastructure that generates events for each enter and exit of a function. It is the same infrastructure that is used by the `profile` tool. As the bindings utilize Score-P itself, the different paradigms listed above can be combined and analyzed even if they are used from within a Python application.

Especially the MPI support of Score-P is of interest, as `pySDC` uses `mpi4py` for parallelization in time. Note that `mpi4py` uses matched probes and receives (`MPI_Mprobe` and `MPI_Mrecv`), which ensures thread safety. However, Score-P did not have support for `Mprobe/Mrecv` in the released version, so we had to switch to a development version of Score-P, where the support was added for this project. Full support for matched communication is expected in an upcoming release of Score-P.

Moreover, as not each function might be of interest for the analysis of an application, it is possible to manually enable and disable the instrumentation or to instrument regions manually, see Listing 4 in Section 4.2 for an example. These techniques can be used to control the detail of recorded information and therefore to control the measurement overhead.

3.2 Cube

Cube is the performance report explorer for Score-P as well as for Scalasca (see below). The CUBE data model is a three-dimensional performance space consisting of the dimensions (i) performance metric, (ii) call-path, and (iii) system location. Each dimension is represented in the GUI as a tree and shown in three coupled tree browsers, i.e. upon selection of one tree item the other trees are updated. Non-leaf nodes of each tree can be collapsed or expanded to achieve the desired level of granularity. We refer to Figure 12 for the graphical user interface of Cube. The metrics that are recorded by default contain the time per call, the number of calls to each function and the bytes transferred in MPI calls. Additional metrics depend on the measurement configuration. The Cube-GUI is highly customizable and extendable. It provides a plugin interface to add new analysis capabilities [23] and an integrated domain-specific language called CubePL to manipulate CUBE metrics [37], enabling completely new kinds of analysis.

3.3 Scalasca

Scalasca [16] is an automatic analyzer of OTF2 traces generated by Score-P. The idea of Scalasca, as outlined in Figure 4, is to perform an automatic search for patterns indicating inefficient behavior. The whole low-level trace data is considered and only a high-level result in the form of a CUBE report is generated. This report has the same structure as a Score-P profile report, but contains additional metrics for the patterns that Scalasca detected. Scalasca performs three major tasks: (i) an identification of wait states, like the Late Receiver pattern shown in Figure 5 and their respective root-causes [47], (ii) a classification of the behaviour and a quantification of its significance and (iii) a scalable identification of the critical path of the execution [2]. As Scalasca is primarily targeted at large-scale applications, the analyzer is a parallel program itself, typically running on the same resources as the original application. This enables a unique scalability to the largest machines available [15]. Scalasca offers convenience commands to start the analysis right after measurement in the same job. Unfortunately,

Fig. 4: The Scalasca approach for a scalable parallel trace analysis. The entire trace date is analyzed and only a high-level result is stored in the form of a Cube report.

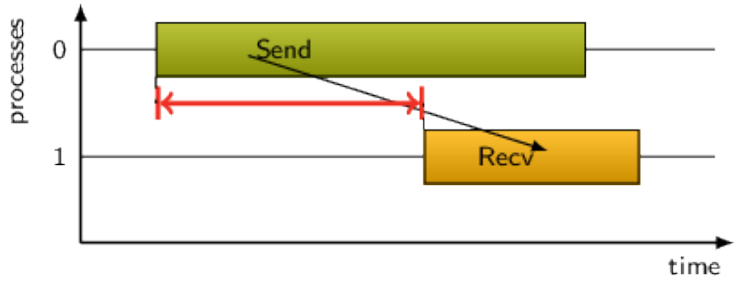


Fig. 5: Example of the Late Receiver pattern as detected by Scalasca. Process 0 post the Send before process 1 posts the Recv. The red arrow indicates waiting time and thus a performance inefficiency.

this does not work with Python yet, in this case the analyzer has to be started separately, see line 21 in Listing 3.

3.4 Vampir

Complementary to the automatic trace analysis with Scalasca - and often more intuitive to the user - is a manual analysis with Vampir. Vampir [24] is a powerful trace viewer for OTF2 trace files. In contrast to traditional profile viewers, which only visualize the call hierarchy and function runtimes, Vampir allows the investigation of the whole application flow. Any metrics collected by Score-P, from PAPI or counter plugins, can be analyzed across processes and time with either a timeline or as a heatmap in the Performance Radar. Recently added was the functionality to visualize I/O-events like reads and writes from and to the hard drive [30]. It is possible to zoom into any level of detail, which automatically updated all views and shows the information from the selected part of the trace. Besides opening an OTF2 file directly, Vampir can connect to VampirServer, which uses multiple MPI processes on the remote system to load the traces. This approach improves scalability and removes the necessity to copy the trace file. VampirServer allows the visualisation of traces from large-scale application runs with multiple thousand processes. The size of such traces is typically in the order of several Gigabyte.

3.5 JUBE

Managing complex workflows of HPC applications can be a complex and error-prone task and often results in significant amounts of manual work. Application parameters may change at several steps in these workflows. In addition, reproducibility of program results is very important but hard to handle when parametrizations change multiple times during the development process. Usually application-specific, hardly documented script based solutions are used to accomplish these tasks.

In contrast, the JUBE benchmarking environment provides a lightweight, command line based, configurable framework to specify, run and monitor the parameter handling and the general workflow execution. This allows a faster integration process and easier adoption of necessary workflow mechanics [27].

Parameters are the central JUBE elements and can be used to configure the application, to replace parts of the source code or to be even used within other parameters. Also the workflow execution itself is managed through the parameter setup by automatically looping through all available parameter combinations in combination with a dependency driven step structure. For reproducibility, JUBE also takes care of the directory management to provide a sandbox space for each execution. Finally, JUBE allows to extract relevant patterns from the application output to create a single result overview to combine the input parametrization and the extracted output results.

To port an application workflow into the JUBE framework, its basic compilation (if requested) and execution command steps have to be listed within a JUBE configuration file. To allow the sandbox directory handling, all necessary external files (source codes, input data and configuration files) have to be listed as well. On top, the user can add the specific parametrization by introducing named key/value pairs. These pairs can either provide a fixed one to one key/value mapping or, in case of a parameter study, multiple values can be mapped to the same key. In such a case JUBE starts to spawn a decision tree, by using every available value combination for a separate program step execution. Figure 6 shows a simple graph example where three different program steps (pre-processing, compile and execution) are executed in a specific order and three different parameters (`const`, `p1` and `p2`) are defined. Once the parameters are defined, they can be used to substitute parts of the original source files or to directly define certain options within the individual program configuration list. Typically, an application-specific template file is designed to be filled by JUBE parameters afterwards. Once the templates and the JUBE configuration file is in place, the JUBE command line tools are used to start the overall workflow execution. JUBE automatically spawns the necessary parameter tree, creates the sandbox directories and executes the given commands multiple times based on the parameter configuration.

To take care of the typical HPC environment, JUBE also helps with the job submission part by providing a set of job scheduler-specific script templates. This is especially helpful for scaling tests by easily varying the amount of compute devices using a single parameter within the JUBE configuration file. JUBE itself

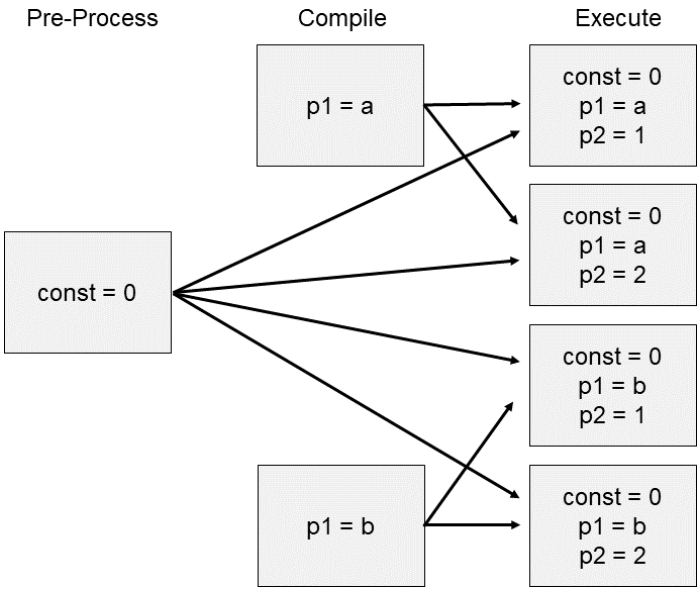


Fig. 6: JUBE workflow example

is not aware of the different types of HPC schedulers, therefore it uses a simple marker file mechanic to recognize if a specific job was finally executed. In Sect. 4.1 we show detailed examples for a configuration file and a jobscript template.

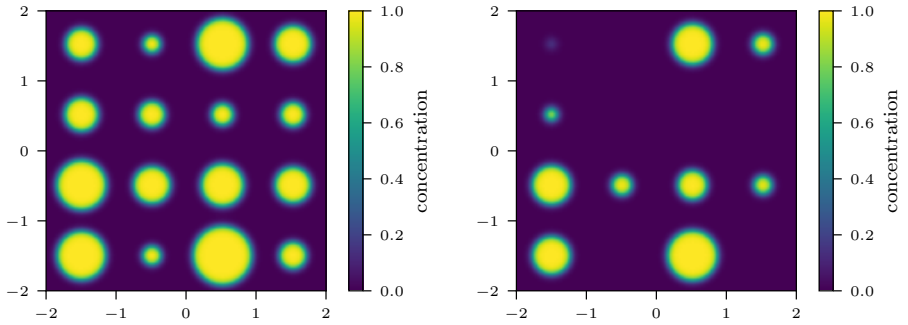
The generic approach of JUBE allows it to easily replace any manual workflow. For example, to use JUBE for an automated performance analysis, using the highlighted performance tools, the necessary Score-P and Scalasca command line options can be directly stored within a parameter, which can then be used during compilation and job submission. After the job execution, even the performance metric extraction can be automated, by converting the profiling data files within an additional performance tool specific post-processing step into a JUBE parsable output format. This approach allows to easily rerun a specific analysis or even combine performance analysis together with a scaling run, to determine individual metric degradation towards scaling capabilities.

4 Results and Lessons Learned

In the following we consider the two-dimensional Allen-Cahn equation

$$u_t = \Delta u - \frac{2}{\epsilon^2} u(1-u)(1-2u) \tag{5}$$

$$u(x,0) = \sum_{i=1}^{L} \sum_{j=1}^{L} u_{i,j}(x)$$



(a) Initial conditions (b) System at time-step 24

Fig. 7: Evolution of the Allen-Cahn problem used for this analysis.

with periodic boundary conditions, scaling parameter $\epsilon > 0$ and $x \in \mathbb{R}^N$, $N \in \mathbb{N}$. Note that as a slight abuse of notation $u(x, 0)$ is the initial condition for the initial value problem, whereas in Sect. 2.1 u_0 represents the initial value for the individual time slabs. The domain in space $[-L/2, L/2]^2$, $L \in \mathbb{N}$, consists of L^2 patches of size 1×1 and in each patch we start with a circle

$$u_{i,j}(x) = \frac{1}{2}\left(1 + \tanh\left(\frac{R_{i,j} - \|x\|}{\sqrt{2}\epsilon}\right)\right),$$

of initial radius $R_{i,j} > 0$ which is chosen randomly between 0.5ϵ and 3ϵ for each patch. For $L = 1$ and this set of parameters, this is precisely the well-known shrinking circle problem, where the dynamics is known and which can be used to verify the simulation [46]. By increasing the parameter L, the simulation domain can be increased without changing the evolution of the simulation fundamentally. For the test shown here we use $L = 4$, finite differences in space with $N = 576$ and $\epsilon = 0.04$, so that initially about 6 points resolve the interfaces, which have a width of about 7ϵ. We furthermore use $M = 3$ Gauss-Radau nodes and $\Delta t = 0.001 < \epsilon^2$ for the collocation problem and stop the simulation after 24 time-steps at $T = 0.024$. We split the right-hand side of (5) and treat the linear diffusion part implicitly using the LU trick [45] and the nonlinear reaction part explicitly using the explicit Euler preconditioner. This has been shown to be the fastest SDC variant in [40] and allows us to use the `mpi4py-fft` library [8] for solving the implicit system, for applying the Laplacian and for transferring data between coarse and fine levels in space. The iterations are stopped when a residual tolerance of 10^{-8} is reached. For coarsening, only 96 points in space were used on the coarse level and, following [4], 3 sweeps are done on the fine level and 1 on the coarse one. All tests were run on the JURECA cluster at JSC [21] using Python 3.6.8 with the Intel compiler and (unless otherwise stated) Intel MPI. The code can be found in the `projects/Performance` folder of `pySDC` [41]. Figure 7 shows the evolution of the system with $L = 4$ from the initial condition in 7a to the 24th time-step in 7b.

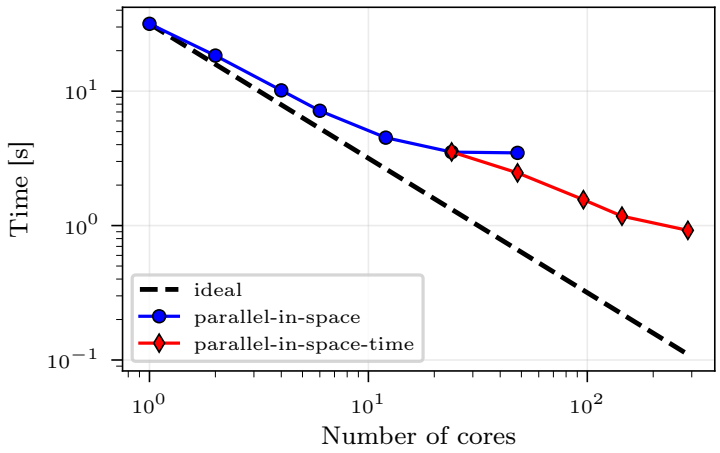


Fig. 8: Time vs. number of cores in space and time.

4.1 Scalability test with JUBE

In Figure 8 the scalability of the code in space and time is shown. While spatial parallelization stagnates at about 24 cores, adding temporal parallelism with PFASST allows to use 12 times more processors for an additional speedup of about 4. Note that using even more cores in time increases the runtime again due to a much higher number of iterations. Also, using more than 48 cores in space is not possible due to the size of the problem. We do not consider larger-scale problems and parallelization here, since a detailed a performance analysis in this case is currently work in progress together with the EU Centre of Excellence "Performance Optimisation and Productivity" (POP CoE, see [6] for details).

The runs were set up and executed using JUBE. The corresponding XML file is shown in Listings 1 and 2. The first listing contains the input and operations part of the file and consists of four blocks:

1. the parameter set (lines 6-16),
2. the rules for substituting the parameter values in the template to build the executable (lines 18-27),
3. the list of files to copy over to the run directory (lines 29-33),
4. and the operations part, where the shell command for submitting the job is given (lines 35-42).

While the last two are rather straightforward and do not require too much of the user's attention, the first two are the ones where the simulation and run parameters find their way into the actual execution. In lines 8-12, the number of compute nodes and the number of tasks (or cores) are set up. Using the python mode in lines 9 and 11, the variable i from line 8 is taken to step simultaneously through the number of nodes and tasks. Without this, for each number of nodes, all number of tasks would be used in separate runs, i.e. instead of 10 runs,

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="pySDC AC scaling test" outpath="bench_run_SPxTP">
    <comment>Scaling test with pySDC</comment>

    <!-- Parameters -->
    <parameterset name="param_set">
      <parameter name="i">0, 1, 2, 3, 4, 5, 6, 7, 8, 9</parameter>
      <parameter name="nnodes" mode="python" type="int">
            [1, 1, 1, 1,  1,  1,  2,  4,  6, 12][$i]</parameter>
      <parameter name="ntasks" mode="python" type="int">
            [1, 2, 4, 6, 12, 24, 24, 24, 24, 24][$i]</parameter>
      <parameter name="space_size" mode="python" type="int">
            $ntasks</parameter>
      <parameter name="mpi" type="str">intel, parastation</parameter>
    </parameterset>

    <!-- Substitute -->
    <substituteset name="substitute">
      <!-- Substitute files -->
      <iofile in="run_pySDC_AC.tmpl" out="run_pySDC_AC.exe" />
      <!-- Substitute commands -->
      <sub source="#NNODES#" dest="$nnodes" />
      <sub source="#NTASKS#" dest="$ntasks" />
      <sub source="#SPACE_SIZE#" dest="$space_size" />
      <sub source="#MPI#" dest="$mpi" />
    </substituteset>

    <!-- Files -->
    <fileset name="files">
      <copy>run_pySDC_AC.tmpl</copy>
      <copy>run_benchmark.py</copy>
    </fileset>

    <!-- Operation -->
    <step name="sub_step">
      <use>param_set</use>    <!-- use existing parameterset -->
      <use>files</use>        <!-- use existing fileset -->
      <use>substitute</use>   <!-- use existing substituteset -->
      <!-- shell command -->
      <do done_file="ready">sbatch run_pySDC_AC.exe</do>
    </step>

    ...
```

Listing 1: XML input file for JUBE running space-parallel and space-and-time-parallel runs (part 1, input and operations).

```
    ...

    <!-- Regex pattern -->
    <patternset name="pattern">
      <pattern name="timing_pat" type="float">
            Time to solution: $jube_pat_fp sec.</pattern>
      <pattern name="niter_pat" type="float">
            Mean number of iterations: $jube_pat_fp</pattern>
    </patternset>

    <!-- Analyze -->
    <analyser name="analyze">
      <use>pattern</use> <!-- use existing patternset -->
      <analyse step="sub_step">
        <file>run.out</file> <!-- file which should be scanned -->
      </analyse>
    </analyser>

    <!-- Create result table -->
    <result>
      <use>analyze</use> <!-- use existing analyser -->
      <table name="result" style="pretty" sort="space_size">
        <column>nnodes</column>
        <column>ntasks</column>
        <column>space_size</column>
        <column>mpi</column>
        <column>timing_pat</column>
        <column>niter_pat</column>
      </table>
    </result>

  </benchmark>
</jube>
```

Listing 2: XML input file for JUBE running space-parallel and space-and-time-parallel runs (part 2, output and analysis).

we would end up with 100 runs, most of them irrelevant. Then, in lines 13-14, the simulation parameter `space_size` is defined as being equal to the number of tasks. This specifies the number of cores for the spatial parallelization. In line 15, two different MPI versions are requested, where the parameter `mpi` is then handled appropriately in the jobscript. For each combination of these parameters, JUBE creates a separate directory with all necessary files and folders. The template jobscript `run_pySDC_AC.tmpl` is replaced by an actual jobscript `run_pySDC_AC.exe`, see line 21, with all parameters in place. An example of a template jobscript can be found in Listing 3.

The second listing 2 continues the XML file with the output and analysis blocks. We have:

1. the pattern block (lines 3-9), which will be used to extract data from the output files of the simulation,
2. the analyzer (lines 11-17), which simply applies the pattern to the output file,
3. and the result block (lines 19-30) to create a "pretty" table with the results, based on the parameters and the extracted results.

Using a simple Python script, this table can be read in again and processed into Figure 8. With JUBE, this workflow can be completely automated using only a few configuration files and a post-processing script. All relevant configuration files can be found in the project folder.

4.2 Performance analysis with Score-P, Scalasca and Vampir

Performance analysis of a parallel application is not an easy task in general and with non-traditional HPC applications in particular. Python applications are still very rare in the HPC landscape and performance analysis tools (and performance analysts for that matter) are often not yet fully prepared for this scenario. In this section we present the challenges we faced and the solutions we found to show what tools can do.

We also would like to encourage other application developers to seek assistance from the tool developers and their system administrators when obstacles are encountered in order to get reasonable and satisfactory results.

First measurement attempts The first obstacle we encountered was that the Score-P Python bindings did not build for the tool-chain of Intel compilers and IntelMPI due to an issue with the Intel compiler installation on JURECA. We thus switched to GNU compilers and ParaStationMPI[5]. Using that we were able to obtain a first analysis result.

The workflow to get these results is as follows: After setting up the runs with JUBE XML files as described above, the job is submitted via JUBE using the jobscript generated from the template.

[5] `https://www.par-tec.com/products/parastation-mpi.html`

```
1  #!/bin/bash -x
2  #SBATCH --nodes=#NNODES#
3  #SBATCH --ntasks-per-node=#NTASKS#
4  #SBATCH --output=run.out
5  #SBATCH --error=run.err
6  #SBATCH --time=00:05:00
7  #SBATCH --partition=batch
8
9  export MPI=#MPI#
10
11 if [ "$MPI" = "intel" ];
12 ... # logic to distiguish MPI libraries
13 fi
14
15 export SCOREP_EXPERIMENT_DIRECTORY=data/scorep-$MPI
16 export SCOREP_PROFILING_MAX_CALLPATH_DEPTH=90
17 export SCOREP_ENABLE_TRACING=1
18 export SCOREP_METRIC_PAPI=PAPI_TOT_INS
19
20 srun python -m scorep --mpp=mpi run_benchmark.py -n #SPACE_SIZE#
21 srun scout.mpi --time-correct $SCOREP_EXPERIMENT_DIRECTORY/traces.otf2
22 touch ready
```

Listing 3: Jobscript template to run the simulation with profiling and tracing enabled.

Listing 3 shows such a template, where all variables of the form `#NAME#` will be replaced by actual values for the specific run. Lines 2-7 provide the allocation and job information for the Slurm Workload Manager. In lines 9-13, the distinction between different MPI libraries is implemented, using different modules and virtual Python environments (not shown here). Lines 15-18 define flags for the Score-P infrastructure, e.g. tracing is enabled (line 17). Then, line 20 contains the run command, where the Score-P infrastructure is passed using the `-m` switch. This generates both a profile report (profiling is enabled by default) for an analysis with CUBE and OTF2 trace files, which can be analyzed manually with Vampir or automatically with Scalasca. The Scalasca trace analyzer is called on line 21. As `pySDC` is a pure MPI application, `scout.mpi` is used here (there is also a `scout.omp` for OpenMP and a `scout.hyb` for hybrid programs). Note that tracing is enabled manually here, but could be part of the parameter input as described in Sect. 3.5. Finally, line 22 marks this particular run as completed for JUBE. The resulting files can then be read by tools like Vampir and CUBE.

Using this setup we were able to get a first usable measurement. We used filtering and Score-P's manual instrumentation API to mark the interesting parts of the application. In Listing 4, a mock-up of a PFASST implementation is shown. Here, after importing the Python module `scorep.user`, separate regions

```python
from mpi4py import MPI
from pySDC.core.Controller import controller

import scorep.user as spu

...

def run_pfasst(*args, **kwargs):
    ...

    while not done:
        ...
        name = f'REGION -- IT_FINE -- {my_rank}'
        spu.region_begin(name)
        controller.do_fine_sweep()
        spu.region_end(name)
        ...
        name = f'REGION -- IT_DOWN -- {my_rank}'
        spu.region_begin(name)
        controller.transfer_down()
        spu.region_end(name)
        ...
        name = f'REGION -- IT_COARSE -- {my_rank}'
        spu.region_begin(name)
        controller.do_coarse_sweep()
        spu.region_end(name)
        ...
        name = f'REGION -- IT_UP -- {my_rank}'
        spu.region_begin(name)
        controller.transfer_up()
        spu.region_end(name)
        ...
        name = f'REGION -- IT_CHECK -- {my_rank}'
        spu.region_begin(name)
        controller.check_convergence()
        spu.region_end(name)
        ...

    ...

...
```

Listing 4: Pseudo code of a PFASST implementation using Score-P regions

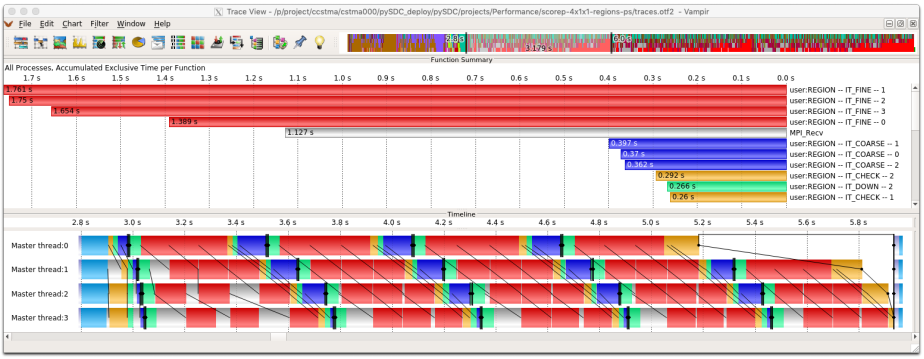


Fig. 9: Vampir visualization: user-defined regions, only a single iteration (ParaStation MPI, 4 processes in time, 1 in space).

can be defined using `region_start` and `region_end`, see e.g. lines 14 and 16. This information will then be available e.g. for filtering in Vampir.

Analysis then showed that the algorithm outlined in Figure 1b worked as expected, at least in principle. This can be seen in Figure 9: the bottom part shows exactly a transposed version of the original communication and workflow structure as expected from Figure 1b. The middle part shows the amount of time spend in the different regions: the vast majority of the computation time (70 %) is spent in the fine sweep, and only about 3 % in the coarse sweep.

Another, more high-level overview of the parallel performance can be gained with the Advisor plugin of Cube [23]. This prints the efficiency metrics developed in the POP project[6] for the entire execution or an arbitrary phase of the application. Figure 10a shows a screenshot of the Advisor result for the computational part of `pySDC`, i.e. omitting initialization and finalization.

The main value to look for is "Parallel Efficiency", which reveals the inefficiency in splitting computation over processes and then communicating data between processes. In this case the "Parallel Efficiency", which is defined as the product of "Load Balance" and "Communication Efficiency", is 79 %, which is worse than what we expected for this small test case. We know from Sect. 2.2 that due to the sequential coarse level and the predictor, PFASST runs will always show slight load imbalances, so the "Load Balance" value of 89 % is understood.

However, the "Communication Efficiency" of 88 % is way below our expectations. A "Serialisation Efficiency" of 98 % indicates that there is hardly any waiting time. The "Transfer Efficiency" of 90 % means we lose significant time due to data transfers. This was not expected so we assumed either an issue with the implementation of the algorithm or the system environment. A Scalasca analysis showed that the slight serialisation inefficiency originates from a "Late Receiver pattern" (see Figure 5) in the fine sweep phase and a "Late Broadcast"

[6] `https://pop-coe.eu/node/69`

after each time step, but did not reveal the reason for the loss in transfer efficiency. A closer look with Vampir at just a single iteration, as shown in Figure 9, finally reveals the issue.

The implementation of `pySDC` uses non-blocking MPI communication in order to overlap computation and communication. However, Figure 9 clearly shows that this does not work as expected.

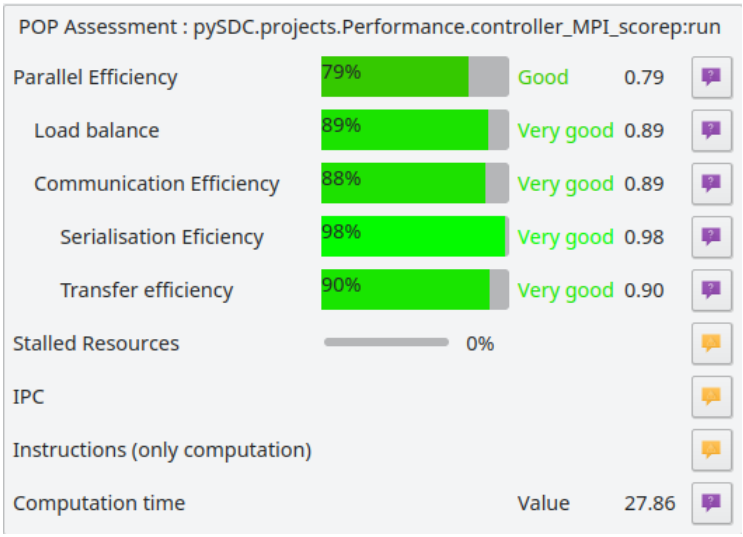
In the time of the analysis of the ParaStationMPI runs there was an update of the JURECA software environment which finally enabled the support of the Score-P Python wrappers for the Intel compilers and IntelMPI. So we performed the same analysis again for this constellation, the one we originally intended to analyze anyway. Surprisingly, the results looked much better now. The Cube Advisor analysis now showed nearly perfect Transfer Efficiency and subsequently a much improved Parallel Efficiency, see Figure 10b.

Vampir further confirms a very good overlap of computation and communication, the way the implementation intended it to be, see Figure 11.

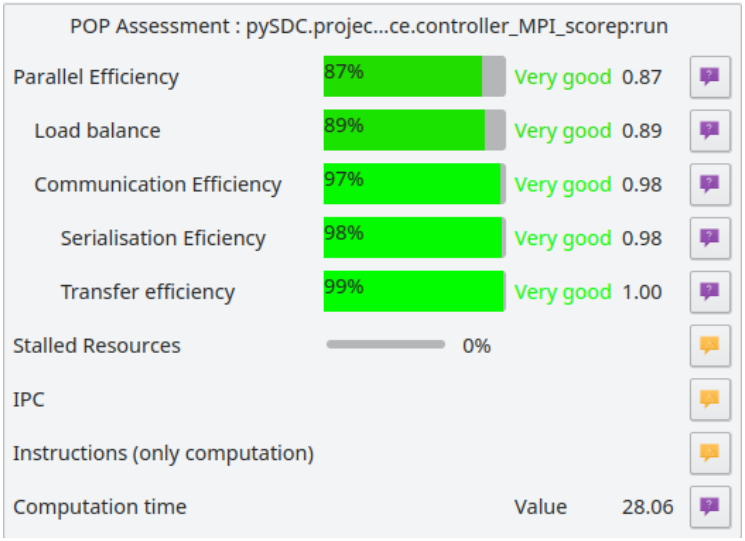
Eye for the detail Thus, the natural question to ask is where these differences between the exact same code running on two different tool-chains come from. Further investigation showed that the installation of ParaStationMPI provided on JURECA does not provide an MPI progress thread, i.e. MPI communication cannot be performed asynchronously and thus overlapping computation and communication is not possible. IntelMPI on the other hand always uses a progress thread if not explicitly disabled via an environment variable. With a newly installed test version of ParaStationMPI, where an MPI progress thread has been enabled, the overlap of computation and communication is possible there, too. We then see an similar performance of `pySDC` using the new ParaStationMPI and IntelMPI.

Even though the overlap problem does not seem to be that much of an issue for this small test case, where just 8% efficiency could be gained, we want to emphasize that these little issues can become severe ones when scaling up. Figure 12 shows the average time per call of the fine sweep, as calculated by CUBE. In the Intel case with overlap we see that the fine sweep time is very balanced across the processes (Figure 12b). In the ParaStationMPI case we see that the fine sweep time increases with the process number (Figure 12a). This problem will likely become worse when the problem size is increased, thus limiting the maximum number of processes that can be utilized.

The scaling tests as well as the performance analysis made for this work are rather small compared to what joined space and time parallelism can do. The difference when using space-parallel solvers can be quite substantial for the analysis ranging from larger datasets for the analysis and visualization to more complex communication patterns. In addition, the issues experienced can differ, as we already see for the test case at hand. In Figure 13, we now use 2 processes in space and 4 in time. There is still unnecessary waiting time, but its impact is much smaller. This is due to the fact that progress of MPI calls does not depend on the MPI communicator, i.e. for each application of the space-parallel FFT



(a) Cube Advisor showing the POP metrics for pySDC with ParaStationMPI.



(b) Cube Advisor showing the POP metrics for pySDC with IntelMPI.

Fig. 10: Cube Advisor showing the POP metrics for pySDC

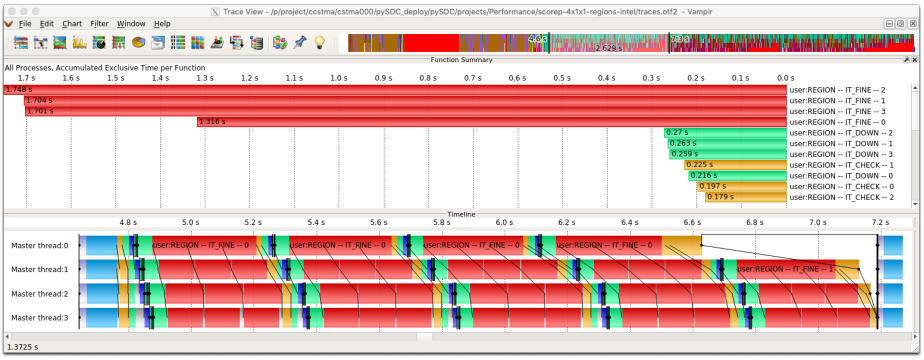


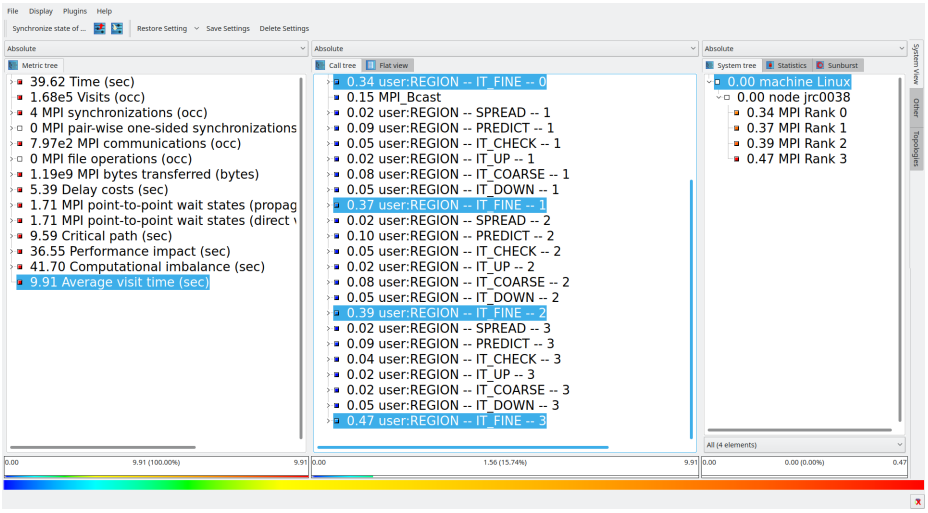
Fig. 11: Vampir visualization: user-defined regions, only a single iteration (Intel MPI, 4 processes in time, 1 in space).

solver progress does happen even in the time-communicator. A more thorough and in-depth analysis of large-scale runs is currently under way together with the POP CoE and we will report on the outcome of this in a future publication.

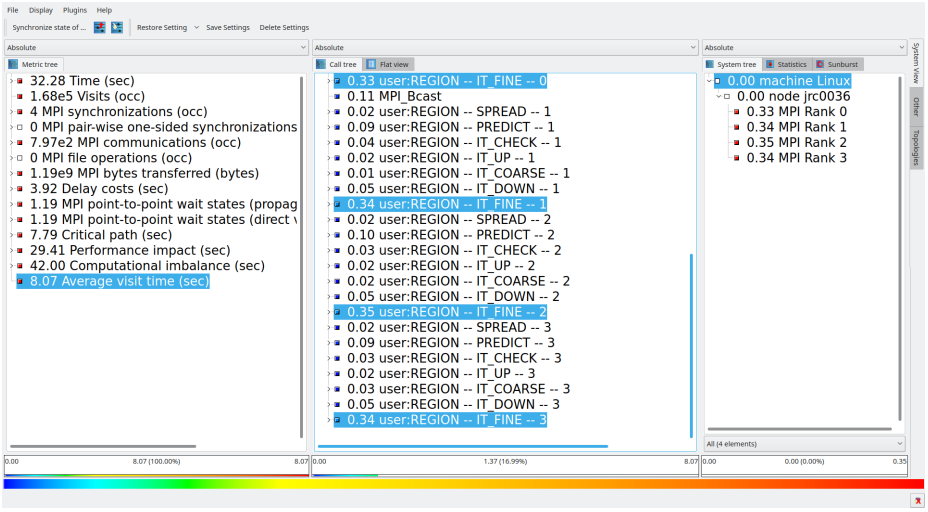
5 Conclusion and Outlook

In this paper we performed and analyzed parallel runs of the PFASST implementation `pySDC` using the performance tools Score-P, CUBE, Scalasca and Vampir as well as the benchmarking environment JUBE. While the implementation complexity of a time-parallel method may vary, with standard Parareal being on one side of the spectrum and methods like PFASST on the other, it is crucial to check and analyze the actual performance of the code. This is particularly true for most time-parallel methods with their theoretically grounded low parallel efficiency, since here problems in the communication can easily be mistaken for method-related shortcomings.

As we have shown, the performance analysis tools in the Score-P ecosystem cannot only be used to identify tuning potential but also allow to easily check for bugs and unexpected behavior, without the need to do “print”-debugging. While methods like Parareal may be straightforward to implement, PFASST is not, in particular due to many edge cases which the code needs to take care of. For example, in the standard PFASST implementation the residual is checked locally for each time-step individually, so that a process working on a later time-step could, erroneously, decide to stop although the iterations on previous time-steps still run. Vice versa, when previous time-steps did converge, the processes dealing with later ones should not expect to receive new data. Depending on the implementation, those cases could lead to deadlocks (the “good” case) or to unexpected results (the “bad” case), e.g. when one-sided communication is used, or other unwanted behavior. Many of these issues can be checked by looking at the gathered data after an instrumented run. This does not, however, replace a



(a) Cube screenshot showing the average time per call of the fine sweep for ParaStationMPI. Time increases with process number.



(b) Cube screenshot showing the average time per call of the fine sweep for IntelMPI. Time is well balanced across the processes.

Fig. 12: Cube screenshots showing the average time per call of the fine sweep

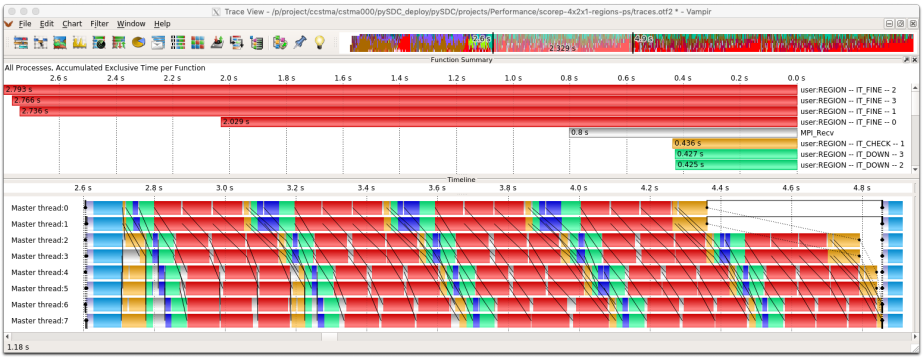


Fig. 13: Vampir visualization: user-defined regions, only a single iteration (ParaStation MPI, 4 processes in time, 2 in space)

careful design of the code, testing, benchmarking, verification and, sometimes, rethinking.

We saw for `pySDC` that already the choice of the MPI implementation can influence the performance quite severely, let alone the unexpected deviation from the intended workflow of the method. Performance tools as the ones presented here help to verify (or falsify) that the implementation of an algorithm actually does what the developers thinks it does. While there is a lot of documentation on these tools available, it is extremely helpful and productive to get in touch with the core developers, either directly or by attending one of the tutorials e.g. provided by the VI-HPS through the Tuning Workshop series[7]. This way, many of the pitfalls and sources of frustration can be avoided and the full potential of these tools becomes visible.

In order to set up experiments using parallel codes in a structured way, be it for performance analysis, parameter studies or scaling tests, tools like JUBE can be used to ease the management of submission, monitoring and post-processing of the jobs. Besides parameters for the model, the methods in space and time, the iteration and so on, the application of time-parallel methods in combination with spatial parallelism adds another level of complexity, which becomes manageable with tools like JUBE.

Acknowledgements

Parts of this work have received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No 676553 and 824080. RS thankfully acknowledges the financial support by the German Federal Ministry of Education and Research through the ParaPhase project within the framework "IKT 2020 - Forschung für Innovationen" (project number 01IH15005A).

[7] `https://www.vi-hps.org/training/tws/tuning-workshop-series.html`

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience **22**(6), 685–701 (2010)
2. Böhme, D., Wolf, F., de Supinski, B.R., Schulz, M., Geimer, M.: Scalable critical-path based performance analysis. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp. 1330–1340. IEEE (2012)
3. Bolten, M., Moser, D., Speck, R.: A multigrid perspective on the parallel full approximation scheme in space and time. Numerical Linear Algebra with Applications **24**(6), e2110–n/a (2017). DOI 10.1002/nla.2110. URL `http://dx.doi.org/10.1002/nla.2110`. E2110 nla.2110
4. Bolten, M., Moser, D., Speck, R.: Asymptotic convergence of the parallel full approximation scheme in space and time for linear problems. Numerical linear algebra with applications **25**(6), e2208 – (2018). DOI 10.1002/nla.2208. URL `https://juser.fz-juelich.de/record/857114`
5. Bradley, T.: GPU Performance Analysis and Optimisation. NVIDIA Corporation (2012)
6. Center, B.S.: Website for POP CoE (2019). URL `https://pop-coe.eu/`. [Online; accessed August 13, 2019]
7. Dalcin, L.D., Paz, R.R., Kler, P.A., Cosimo, A.: Parallel distributed computing using python. Advances in Water Resources **34**(9), 1124 – 1139 (2011). DOI https://doi.org/10.1016/j.advwatres.2011.04.013. URL `http://www.sciencedirect.com/science/article/pii/S0309170811000777`. New Computational Methods and Software Tools
8. Dalcin, Lisandro and Mortensen, Mikael and Keyes, David E: Fast parallel multidimensional FFT using advanced MPI. Journal of Parallel and Distributed Computing (2019). DOI 10.1016/j.jpdc.2019.02.006
9. Dutt, A., Greengard, L., Rokhlin, V.: Spectral deferred correction methods for ordinary differential equations. BIT Numerical Mathematics **40**(2), 241–266 (2000). DOI 10.1023/A:1022338906936. URL `http://dx.doi.org/10.1023/A:1022338906936`
10. Emmett, M., Minion, M.L.: Toward an efficient parallel in time method for partial differential equations. Communications in Applied Mathematics and Computational Science **7**, 105–132 (2012). URL `http://dx.doi.org/10.2140/camcos.2012.7.105`
11. Emmett, M., Minion, M.L.: Efficient implementation of a multi-level parallel in time algorithm. In: Domain Decomposition Methods in Science and Engineering XXI, *Lecture Notes in Computational Science and Engineering*, vol. 98, pp. 359–366. Springer International Publishing (2014). DOI 10.1007/978-3-319-05789-7_33. URL `http://dx.doi.org/10.1007/978-3-319-05789-7_33`
12. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In: Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium, August 30 – September 2 2011, *Advances in Parallel Computing*, vol. 22, pp. 481–490. IOS Press (2012). DOI 10.3233/978-1-61499-041-3-481
13. Feld, C., Convent, S., Hermanns, M.A., Protze, J., Geimer, M., Mohr, B.: Score-p and ompt: Navigating the perils of callback-driven parallel runtime introspection. In: X. Fan, B.R. de Supinski, O. Sinnen, N. Giacaman (eds.) OpenMP: Conquering

the Full Hardware Spectrum, pp. 21–35. Springer International Publishing, Cham (2019)
14. Gander, M.J.: 50 years of Time Parallel Time Integration. In: Multiple Shooting and Time Domain Decomposition. Springer (2015). URL http://dx.doi.org/10.1007/978-3-319-23321-5_3
15. Geimer, M., Saviankou, P., Strube, A., Szebenyi, Z., Wolf, F., Wylie, B.J.N.: Further improving the scalability of the scalasca toolset. In: K. Jónasson (ed.) Applied Parallel and Scientific Computing, pp. 463–473. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
16. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The SCALASCA performance toolset architecture. In: International Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece, pp. 51–65 (2008)
17. Gocht, A., Schöne, R., , Frenzel, J.: Advanced Python Performance Monitoring with Score-P. In: Tools for High Performance Computing 2019, p. to appear. Springer International Publishing (2019)
18. Harlacher, M., Calotoiu, A., Dennis, J., Wolf, F.: Analysing the Scalability of Climate Codes Using New Features of Scalasca. In: K. Binder, M. Müller, M. Kremer, A. Schnurpfeil (eds.) Proc. of the John von Neumann Institute for Computing (NIC) Symposium 2016, Juelich, Germany, *NIC Series*, vol. 48, pp. 343–352. Forschungszentrum Jülich, John von Neumann-Institut for Computing (2016)
19. Hermanns, M.A., Geimer, M., Mohr, B., Wolf, F.: Trace-based detection of lock contention in MPI one-sided communication. In: C. Niethammer, J. Gracia, T. Hilbrich, A. Knüpfer, M.M. Resch, W.E. Nagel (eds.) Tools for High Performance Computing 2016, Proc. of the 10th Parallel Tools Workshop, Stuttgart, Germany, October 2016, pp. 97–114. Springer (2017). DOI 10.1007/978-3-319-56702-0_6. URL http://juser.fz-juelich.de/record/830159
20. Huang, J., Jia, J., Minion, M.: Accelerating the convergence of spectral deferred correction methods. Journal of Computational Physics **214**(2), 633 – 656 (2006)
21. Jülich Supercomputing Centre: JURECA: General-purpose supercomputer at Jülich Supercomputing Centre. Journal of large-scale research facilities **2**(A62) (2016). DOI 10.17815/jlsrf-2-121. URL http://dx.doi.org/10.17815/jlsrf-2-121
22. Knobloch, M., Mohr, B.: Tools for GPU Computing – Debugging and Performance Analysis of Heterogenous HPC Applications. Supercomputing Frontiers and Innovations **7**(1) (2020). URL https://superfri.org/superfri/article/view/311
23. Knobloch, M., Saviankou, P., Schlütter, M., Visser, A., Mohr, B.: A picture is worth a thousand numbers – Enhancing Cube's analysis capabilities with plugins. In: Tools for High Performance Computing 2019 (tbp)
24. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir Performance Analysis Tool-Set. In: M. Resch, R. Keller, V. Himmler, B. Krammer, A. Schulz (eds.) Tools for High Performance Computing, pp. 139–155. Springer Berlin / Heidelberg (2008). DOI 10.1007/978-3-540-68564-7_9
25. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S.S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Proc. of the 5th Int'l Workshop on Parallel Tools for High Performance Computing, September 2011, Dresden, pp. 79–91. Springer (2012). DOI 10.1007/978-3-642-31476-6_7. URL http://dx.doi.org/10.1007/978-3-642-31476-6_7

26. LLNL: Website for `XBraid` (2018). URL `https://www.llnl.gov/casc/xbraid`. [Online; accessed July 30, 2018]
27. Lührs, S., Rohe, D., Schnurpfeil, A., Thust, K., Frings, W.: Flexible and Generic Workflow Management. In: Parallel Computing: On the Road to Exascale, *Advances in parallel computing*, vol. 27, pp. 431 – 438. International Conference on Parallel Computing 2015, Edinburgh (United Kingdom), 1 Sep 2015 - 4 Sep 2015, IOS Press, Amsterdam (2016). DOI 10.3233/978-1-61499-621-7-431. URL `http://juser.fz-juelich.de/record/808798`
28. Minion, M., Emmett, M.: Website for `libpfasst` (2019). URL `https://github.com/libpfasst/LibPFASST`. [Online; accessed August 13, 2019]
29. Minion, M.L.: A hybrid parareal spectral deferred corrections method. Communications in Applied Mathematics and Computational Science **5**(2), 265–301 (2010). URL `http://dx.doi.org/10.2140/camcos.2010.5.265`
30. Mix, H., Herold, C., Weber, M.: Visualization of Multi-layer I/O Performance in Vampir. In: Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2018 IEEE International (2018)
31. Ong, B.W., Haynes, R.D., Ladd, K.: Algorithm 965: RIDC Methods: A Family of Parallel Time Integrators. ACM Trans. Math. Softw. **43**(1), 8:1–8:13 (2016). DOI 10.1145/2964377. URL `http://doi.acm.org/10.1145/2964377`
32. Ong, B.W., Schroder, J.B.: Applications of time parallelization. Computing and Visualization in Science **23**(1), 1–15 (2020)
33. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: transputer and occam developments, vol. 44, pp. 17–31. Citeseer (1995)
34. Reinders, J.: Vtune performance analyzer essentials. Intel Press (2005)
35. Ruprecht, D., Speck, R.: Spectral deferred corrections with fast-wave slow-wave splitting. SIAM Journal on Scientific Computing **38**(4), A2535–A2557 (2016)
36. Saviankou, P., Knobloch, M., Visser, A., Mohr, B.: Cube v4: From performance report explorer to performance analysis tool. In: Proceedings of the International Conference on Computational Science, ICCS 2015, Computational Science at the Gates of Nature, Reykjavík, Iceland, 1-3 June, 2015, pp. 1343–1352 (2015). DOI 10.1016/j.procs.2015.05.320. URL `https://doi.org/10.1016/j.procs.2015.05.320`
37. Saviankou, P., Knobloch, M., Visser, A., Mohr, B.: Cube v4: From performance report explorer to performance analysis tool. Procedia Computer Science **51**, 1343–1352 (2015)
38. Sharples, W., Zhukov, I., Geimer, M., Goergen, K., Luehrs, S., Breuer, T., Naz, B., Kulkarni, K., Brdar, S., Kollet, S.: A run control framework to streamline profiling, porting, and tuning simulation runs and provenance tracking of geoscientific applications. Geoscientific Model Development **11**(7), 2875–2895 (2018). DOI 10.5194/gmd-11-2875-2018. URL `https://gmd.copernicus.org/articles/11/2875/2018/`
39. Shende, S.S., Malony, A.D.: The TAU parallel performance system. The International Journal of High Performance Computing Applications **20**(2), 287–311 (2006)
40. Speck, R.: Algorithm 997: pySDC - Prototyping Spectral Deferred Corrections. ACM Transactions on Mathematical Software **45**(3) (2019). URL `https://doi.org/10.1145/3310410`
41. Speck, R.: Parallel-in-time/pysdc: The performance release (2019). DOI 10.5281/zenodo.3407254. URL `https://doi.org/10.5281/zenodo.3407254`
42. Speck, R.: Website for `pySDC` (2019). URL `https://parallel-in-time.org/pySDC/`. [Online; accessed August 13, 2019]

43. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with papi-c. In: Tools for High Performance Computing 2009, pp. 157–173. Springer (2010)
44. Treibig, J., Hager, G., Wellein, G.: Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: 2010 39th International Conference on Parallel Processing Workshops, pp. 207–216. IEEE (2010)
45. Weiser, M.: Faster SDC convergence on non-equidistant grids by DIRK sweeps. BIT Numerical Mathematics **55**(4), 1219–1241 (2014)
46. Zhang, J., Du, Q.: Numerical Studies of Discrete Approximations to the Allen-Cahn Equation in the Sharp Interface Limit. SIAM Journal on Scientific Computing **31**(4), 3042–3063 (2009). DOI 10.1137/080738398. URL `https://doi.org/10.1137/080738398`
47. Zhukov, I., Feld, C., Geimer, M., Knobloch, M., Mohr, B., Saviankou, P.: Scalasca v2: Back to the future. In: Proc. of Tools for High Performance Computing 2014, pp. 1–24. Springer (2015). DOI 10.1007/978-3-319-16012-2_1