Geosci. Model Dev., 11, 2875–2895, 2018
https://doi.org/10.5194/gmd-11-2875-2018

Geoscientific Model Development
Open Access
EGU

A run control framework to streamline profiling, porting, and tuning simulation runs and provenance tracking of geoscientific applications

Wendy Sharples[1,2,3], Ilya Zhukov[1], Markus Geimer[1], Klaus Goergen[2,4], Sebastian Luehrs[1], Thomas Breuer[1], Bibi Naz[2,4], Ketan Kulkarni[1,4], Slavko Brdar[1,4], and Stefan Kollet[2,4]

[1]Jülich Supercomputing Centre, Forschungszentrum Jülich, Jülich, Germany
[2]Institute of Bio- and Geosciences, Agrosphere (IBG-3), Forschungszentrum Jülich, Jülich, Germany
[3]Meteorological Institute, University of Bonn, Bonn, Germany
[4]Centre for High-Performance Scientific Computing in Terrestrial Systems, Geoverbund ABC/J, Jülich, Germany

Correspondence: Wendy Sharples (w.sharples@fz-juelich.de)

Received: 30 September 2017 – Discussion started: 27 October 2017
Revised: 23 May 2018 – Accepted: 12 June 2018 – Published: 13 July 2018

Abstract. Geoscientific modeling is constantly evolving, with next-generation geoscientific models and applications placing large demands on high-performance computing (HPC) resources. These demands are being met by new developments in HPC architectures, software libraries, and infrastructures. In addition to the challenge of new massively parallel HPC systems, reproducibility of simulation and analysis results is of great concern. This is due to the fact that next-generation geoscientific models are based on complex model implementations and profiling, modeling, and data processing workflows. Thus, in order to reduce both the duration and the cost of code migration, aid in the development of new models or model components, while ensuring reproducibility and sustainability over the complete data life cycle, an automated approach to profiling, porting, and provenance tracking is necessary. We propose a run control framework (RCF) integrated with a workflow engine as a best practice approach to automate profiling, porting, provenance tracking, and simulation runs. Our RCF encompasses all stages of the modeling chain: (1) preprocess input, (2) compilation of code (including code instrumentation with performance analysis tools), (3) simulation run, and (4) postprocessing and analysis, to address these issues. Within this RCF, the workflow engine is used to create and manage benchmark or simulation parameter combinations and performs the documentation and data organization for reproducibility. In this study, we outline this approach and highlight the subsequent developments scheduled for implementation born out of the extensive profiling of ParFlow. We show that in using our run control framework, testing, benchmarking, profiling, and running models is less time consuming and more robust than running geoscientific applications in an ad hoc fashion, resulting in more efficient use of HPC resources, more strategic code development, and enhanced data integrity and reproducibility.

1 Introduction

Geoscientific modeling is constantly evolving, leading to higher demands on high-performance computing (HPC) resources. We distinguish four main developments which increase HPC demands: (i) higher spatial resolution, for which the added value inherent to simulations at high spatial resolutions has been shown, for example, in many studies of regional convection permitting climate simulations (e.g., Prein et al., 2015; Eyring et al., 2016; Heinzeller et al., 2016) and continental hyper-resolution hydrological modeling approaches (e.g., Kollet and Maxwell, 2008; Maxwell et al., 2015; Keune et al., 2013); (ii) increased model domain size, for which models are now being run at larger scales, for example, global-convection-permitting models (Schwitalla et al., 2016), high-resolution continental regional climate models (Leutwyler et al., 2016), and global hydrology and land surface models, which are needed for water resources

modeling (Bierkens et al., 2015); (iii) increased model complexity in which the desire to explore the feedbacks among the surface, subsurface, oceans, and atmosphere have led to fully coupled multi-physics global or regional Earth system models (ESMs) (e.g., Shrestha et al., 2014; Ruti et al., 2016) posing load-balancing issues (Gasper et al., 2014); and (iv) an increasing number of ensemble members in modeling and data assimilation experiments, which means that several instances of a model will need to run simultaneously (e.g., Han et al., 2016; Kurtz et al., 2016). These developments, combined with long climate scenario simulation time spans pose specific challenges in terms of computational resources, data volume, data velocity, data handling, and analysis.

To keep up with these demands, HPC hardware, software, and tools are developing at a rapid pace. For example, heterogeneous HPC architectures that combine multicore CPUs with accelerators on the same compute node (Brodtkorb et al., 2010) are considered a suitable architecture for future exascale systems because of their energy efficiency (i.e., flops per watt), low-latency data management, and peak performance per accelerator (Davis et al., 2012; Langdon et al., 2016; Kandalla et al., 2016). These coprocessors have tens of cores and can host hundreds of threads per chip, including their own memory with very high bandwidth (Liu et al., 2012), and use different memory architectures (e.g., cache coherence) or parallel programming models (e.g., CUDA, OpenCL, OpenACC). Exascale performance and high energy efficiency are also supported by the use of reconfigurable devices into HPC systems such as the field-programmable gate array (FPGA) integrated circuit (Mavroidis et al., 2016). While these HPC developments are instrumental towards next-generation exascale HPC systems, during the next decade (Attig et al., 2011; Keyes, 2011; Davis et al., 2012; Rigo et al., 2017), parallel simulation codes on multi-core shared or distributed memory architectures need a substantial amount of porting, profiling, tuning, and refactoring (Hwu, 2014) to efficiently use such hardware, in particular because a very high level of vectorization is needed to take advantage of the ever increasing number of execution units in each core for single instruction, multiple data (SIMD) architectures. Moreover, offloading compute-intensive code sections to accelerators can also become a performance bottleneck due to excessive data transfers between host and accelerator. Thus, special care needs to be taken with respect to data layout, placement, and reuse.

Therefore, in many cases, complicated legacy codes need substantial investments in model porting, tuning, and refactoring, in order to efficiently use these upcoming and already existing HPC architectures and achieve a high level of performance. Invested effort has already paid off for many codes (Meadows, 2012; Hammond et al., 2014; Leutwyler et al., 2016; Heinzeller et al., 2016), in the form of significantly reduced runtimes, however, at a significant cost in resources. For example in Leutwyler et al. (2016), porting COSMO to graphics processing units (GPUs) brought reduced simulation times (speedup on the order of 3.6) but needed a team of developers to bring this about. In order to reduce both the duration and the cost of code migration, and also to aid in the development of new models or model components, a systematic, rigorous approach is needed to fully analyze and understand the runtime behavior and I/O characteristics in detail, and identify performance bottlenecks. In this context, the use of performance analysis tools is crucial. A run control framework with integrated performance analysis tools can automate a performance engineering approach as well as gather information from the resulting output and analyze that output. However, depending on the current focus of the analysis, different tools and techniques may have to be used – sometimes even in combination. For example, while various tools provide generic information about the runtime behavior of an application, specialized tools exist that focus on a particular aspect such as vectorization, threading, communication and synchronization, or I/O. Likewise, while profiling – i.e., the process-local generation of aggregated performance metrics during the execution – can provide a summarized overview of the performance for the entire application run, it is not able to capture the dynamic runtime behavior. Thus, it can be complemented by using event tracing, which collects performance-related events in chronological order and therefore allows the reconstruction of the dynamic application behavior in detail. However, care has to be taken when using event tracing, as it is more expensive than profiling as the number of data in the trace increases with the runtime of the application (e.g., Geimer et al., 2010; Carns et al., 2011). Thus, it is usually only applied to selected parts of the execution, for example, a few time steps or iterations of a solver, that have been identified using more lightweight techniques such as profiling.

In addition to making efficient use of massively parallel HPC systems, reproducibility of simulation results, based on complex model implementations, profiling, modeling, and data processing workflows, must be a fundamental principle in computational research (Hutton et al., 2016). Recently, Stodden et al. (2016) presented "Reproducibility Enhancement Principles" (REP) to help ensure that the computational steps in data processing and generation are similarly important to access to the data themselves. Hence sharing not only data but also details of software, workflows, and the computational environment via open repositories is likewise important. Similarly, Hutton et al. (2016) recommend for computational hydrology that workflows, which combine data and reusable code, are needed in order to ensure provenance of scientific results. Given that in the weather and climate sciences, data and primary code availability is often ensured, ancillary code availability is addressed in Irving (2016) as one of the root causes for irreproducibility. With this in mind, we consider the aspect of documenting the porting and performance optimization steps as well as provenance tracking during production simulations as highly relevant to ensure reproducibility. Workflow engines such as ecFlow (Bahra,

2011) or cylc (Oliver et al., 2017) can connect all relevant steps of a modeling chain, submit jobs with dependencies, and help with necessary parameter sweeps for application software porting and tuning alike. At the same time they allow for extensive, systematic logging of the processing steps themselves as well as the log outputs from the individual applications.

In this article, we present a run control framework (RCF) as a best practice approach to porting, profiling, and documenting legacy code using the script-based benchmarking framework JUBE (Lührs et al., 2016) as a workflow engine. The framework can be described as the supporting structure to build the geoscience application workflows, whereas the workflow engine is the tool used to automate the workflows. We developed profiling, run control, and testing frameworks which are dynamically built with user input into interdependent tasks and these tasks are run using JUBE. While the use case for this portable run control, profiling, and testing workflow engine system discussed in this paper is the software application ParFlow, an integrated parallel watershed model run on machines at Jülich Supercomputing Center (JSC), the RCF is generic and can be applied to any other simulation software or any other HPC platform. In the remainder of this paper we outline this approach and highlight the subsequent developments scheduled for implementation born out of the extensive profiling of ParFlow. Additionally, we highlight other uses for employing a workflow engine which have enabled us to streamline the run control process for model runs.

2 RCF approach to profiling, portability, and provenance tracking

In this section, the RCF which could be described as a run harness, along with JUBE is introduced, where a harness in this case is used to describe the framework of scripts and other supporting tools that are required to execute a workflow. The RCF is presented as a means to facilitate portability, profiling, and provenance tracking. This is followed by the description of the standard profiling toolset, which is currently built into our RCF to aid in the ParFlow hydrological model development and for production simulation run control as well as the hardware characteristics and the profiling tools available on the supercomputer used in this study. The RCF for the case study in this paper is given in the Supplement as a tar ball file.

2.1 JUBE as a workflow engine

Benchmarking scientific code can assess impacts of changes of the underlying HPC software stack (e.g., compiler or library upgrades) and hardware (e.g., interconnect upgrades), aid in testing as part of software engineering and code refactoring, and aid in finding optimum numerical model configurations. Benchmarking a numerical model system usually involves several runs with different configurations (compiler, domain, physics parameterizations, solver settings, load balancing), including compilation, instrumentation (i.e., the injection of special monitoring "hooks" into the program to enable profiling and/or event tracing), various simulations, profiling, result verification, and analysis. However, with increasing model and HPC environment complexity, the parameter space for benchmarking can be large. To avoid errors in managing benchmark parameter combinations, to reduce the overall temporal effort, and to ensure reproducibility and comparability, benchmarking must be automated. This task can be accomplished using a workflow engine, which is an application for workflow automation, like the JUBE benchmarking environment (Lührs et al., 2016).

JUBE is a script-based framework designed to efficiently and systematically define, set up, run, and analyze benchmarks and production simulations. The current JUBE v2.1.4 is a Python-based implementation released under GNU GPLv3 actively developed at the JSC (https://www.fz-juelich.de/ias/jsc/EN). JUBE allows one to easily define benchmark sets via an XML configuration file, in which the workflow and parameter sweeps are specified. When run, JUBE controls the automatic execution of the designed workflow and takes care of the underlying file structure to allow an individual execution per run. Automatic bookkeeping separates the different runs and parameter combinations and allows reproducible executions. To generate an overview of the overall workflow execution, the user can configure JUBE how to analyze the different output files to extract information such as the overall runtime or other application-specific data. This allows the system to create a combined overview of the underlying parameterization and the application outputs. The features above combined with our described RCF means that users can very quickly get their complicated model workflows up and running without resorting to developing their own specialized bash or Python scripts, which are usually bereft of the features contained in our RCF, to run simulations.

Other workflow engines which are commonly used are ecFlow and cylc. JUBE, ecFlow, and cylc are all written in Python and all have tasks/steps which can be triggered based on dependencies (e.g., a run task would only run on successful compilation) and have variable inheritance in which general variable definitions can be overwritten with specific parameters at runtime. We have chosen to use JUBE as it has been designed to be run on JSC machines and we have large compute projects there; we have direct access to the JUBE developers, and there is a soon-to-be-released version, which can directly interact with the specialized JSC job scheduler at any point in the production run (and can therefore circumvent the 2 h time limitation for a running process on JSC machines). Additionally, we can influence the development of added functionality such as built-in Python scripting for variable declaration, parameter space creation, environment han-

dling, loading files, and substitution. If in the future, ecFlow and cylc prove advantageous to use over JUBE, we could set them up on a continuously running server and tunnel in. Currently JUBE, ecFlow, and cylc workflow engines can interact with a job scheduler and are designed with the purpose of facilitating the automation of workflows. One major benefit of using cylc and ecFlow over JUBE is that they both have a graphical user interface (GUI), which can be very powerful for non-developers. In the case that we could run either cylc or ecFlow continuously on JSC machines as we can JUBE, it would be hard to pick one over the other. Both appear to have the same features and functionality. However as both JUBE and ecFlow can read in XML scripts as input, we would lean towards using ecFlow in the future as it would be easy to swap between the two as we have set up our RCF in XML, while we would need to refactor our RCF for cylc (Bahra, 2011; Oliver et al., 2017). In addition, cylc has been reported to be more complicated when it comes to building workflows (Manubens-Gil et al., 2016).

2.2 RCF description

Whatever workflow engine one decides to use, someone still needs to integrate the workflows or tasks themselves – JUBE, ecFlow, and cylc are simply tools for workflow automation. Leveraging the generic JUBE framework, we developed a run control framework, suitable for a typical geoscience model, from a series of XML files integrated with Python scripts to be executed with JUBE (see Fig. 1). These jobs are usually run with the following modeling chain: (1) preprocess input, (2) compilation of code (includes code instrumentation with profiling tools), (3) simulation run, and (4) postprocessing and analysis. This modeling chain can be thought of as interdependent tasks set up by the RCF, which are then submitted as steps by JUBE. The current run control framework is under version control and can be cloned from GitLab (Hethey, 2013).

The directory structure for the RCF run harness used in the case study in this paper is shown in Fig. 2. A top-level Python script, `jubeRun.py`, combines the custom job specifications (`custom/weakScalingSinusoidal_Job_Juqueen.xml`) with the run control benchmark XML script (`driver/ParFlowRC_Benchmark.xml`) into one XML configuration file `execute.xml`, which can be parsed through JUBE, and then calls the JUBE run command with the newly created file as an argument. Machine-specific profiling and job submission parameter sets are imported from XML structs given in the scripts `templates/platform.xml` and `templates/profiler.xml`, respectively, and the ParFlow model input parameter sets are imported from the structs given in `templates/ParFlow_model_input.xml`, based on the options specified in the custom job. All environment and submission scripts are stored in directories `${machine}_files`, all the profiling specific wrappers and filter files are stored in the directory `profiler_data`, and all ParFlow model input is stored in the directory `model` (see Fig. 2).

The driver script contains the steps to run the modeling chain, where the steps themselves can be dependent on the successful completion of the previous step(s). Compilation parameters are set based on which HPC platform or machine the benchmark suite is run on and the profiling tool(s) chosen. The user can specify the machine, the profiling tool(s), the ParFlow model, the domain size, and the scaling parameters, and they can overwrite the default compilation and job submission parameters via a custom job XML script. The user can also describe an analysis step for the postprocessing of output.

The template scripts are used in our run harness to capture default settings for HPC platform- and model-related parameters. Specific default settings can be overwritten when specified in the custom job XML script. In the case study, the template scripts used contain HPC platform-related default settings for profiling tools (`profiler.xml`), compilers (`platform.xml`), and job submission (`juqueen_files`) (see Fig. 2). In addition there are templates for each type of ParFlow model, for which the idealized overland flow model (see Sect. 3.1) and the type of scaling typically used for a benchmark suite, i.e., weak scaling and strong scaling, is defined. The template ParFlow model XML script sets the default settings for a series of ParFlow models set up over different domains. The template scaling XML script sets the default domain settings for a given ParFlow model. The scaling parameters can be set such that either one subdomain is spawned per thread (weak scaling) or such that the domain size does not change (strong scaling). The custom XML script sets the HPC platform used, the profiling tool, the ParFlow model type, the domain extent and type of scaling, and the postprocessing analysis step. The custom job script can also overwrite default settings such as compiler settings and job submission settings.

Our RCF creates benchmark or production model simulation suites, which can run on multiple computer systems and whose results can be postprocessed and analyzed, via the execution of the driver file using JUBE, in which the driver file ingests custom and template input. All parameters which are comma separated are parsed by JUBE as a parameter sweep, so that each comma-separated variable is iterated over to become a separate run. At the time of writing, our RCF is running on several different machines, namely JUQUEEN (a highly scalable cluster at JSC), JULIA (a prototype KNL cluster at JSC), JURECA (a general purpose cluster at JSC), and JUROPA3 (a prototype test bed system at JSC). All files mentioned in Fig. 2 are available in the tar ball supplied in the Supplement for this paper.

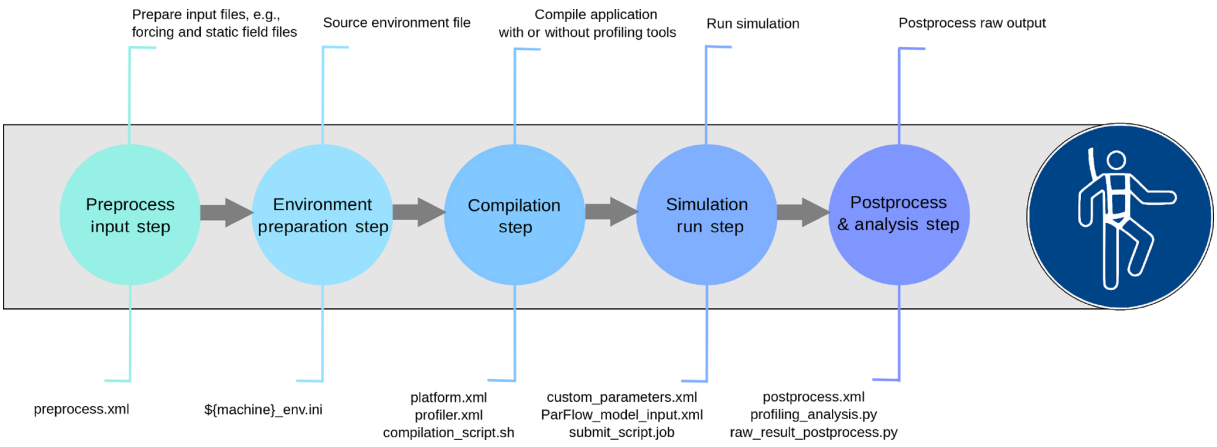

Figure 1. Schematic overview of the modeling chain as supported by our JUBE-based run harness. Each step is annotated with a brief description (at the top) as well as the respective RCF infrastructure (XML files and scripts, at the bottom).

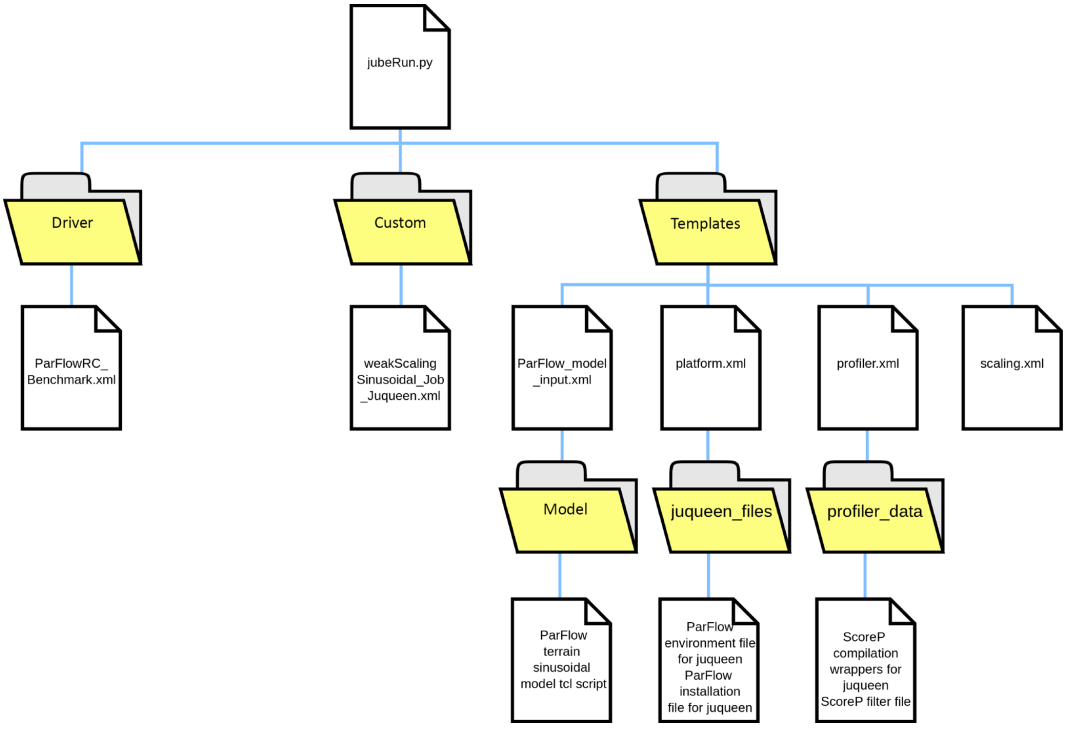

Figure 2. Directory structure of the run control framework used in the case study (Sect. 3).

2.3 RCF: aiding code portability

Within the RCF, we have separated all the information pertinent to the existing compilers, required environment modules, and workload manager job submission specifics for a given system into a single XML file (see Fig. 1, `platform.xml`). This `platform.xml` file can easily be extended to include any new system. When compilers, environment modules, and workload managers are updated or new features or functionality are added, the `platform.xml` can be easily altered to include these updates. For example, as new C and Fortran compiler versions are released with improved code generation and potentially new optimization flags, it is useful to reassess which compilation flags give the best runtime without compromising the accuracy of the result. In the case of our RCF, this is as simple as altering the compiler flags parameter in `platform.xml` with comma-separated values for each different compiler flag, which produces a benchmark suite. In order to ensure accuracy is not compromised, we built in a result comparison test, in which the user can compare the result with previously generated output. However, some thought needs to be

taken in setting up the test as ultimately it is up to the user to decide which models and previously generated output is accurate enough to be the gold standard to test against. If the user is uncomfortable in using their own models for this test, most geoscience applications (ParFlow included) have a plethora of tests with previously generated output to use as a gold standard to test against.

2.4 RCF: facilitating code profiling

In order to analyze ParFlow's runtime behavior, determine optimal runtime settings, and identify performance bottlenecks during model development, we use several complementary performance analysis tools. Setup, compilation wrappers, and analysis profiling steps were built into our RCF (Fig. 1: `profiler.xml`, Appendix A) with support for the following tools: Score-P v3.1 (Knüpfer et al., 2012) and Scalasca v2.3.1 (Geimer et al., 2010; Zhukov et al., 2015), where results collected with Score-P and Scalasca can be examined using the interactive analysis report explorer Cube v4.3.5 (Saviankou et al., 2015), Allinea Performance Reports v7.0.4 (January et al., 2015), Extrae v3.4.3 (Alonso et al., 2012), Paraver v4.6.3 (Labarta et al., 2006), Intel Advisor 2015 (Rane et al., 2015), and Darshan v3.0.0 (Carns et al., 2011) (see Table A1 in Appendix A for a more detailed description of each performance analysis tool listed above). The modeling chain for the profiling workflow is as follows:

1. prepare the input data;
2. load environment modules and set up parameters specific to the performance analysis tool;
3. compile or link ParFlow using scripts and wrappers, depending on what is required by the profiling tool; e.g., Score-P requires compilation and linking using wrappers, whereas Darshan requires only linking after compilation;
4. ParFlow execution with the necessary tool flags (e.g., Scalasca has various runtime measurement, collection, and analysis flags which can be turned on or off);
5. parse and analyze the results interactively (e.g., using interactive visual explorers like Paraver, Cube) or generate a textual performance metric report via a postprocessing step.

Note that code instrumentation with performance analysis tools – that is, the insertion of tool-specific measurement calls, which are executed at relevant points (events) during runtime, into the application code – can introduce significant overhead, which can be assessed by comparing to an uninstrumented reference run. If the runtime of the instrumented version of the code under inspection is much longer than the reference run (more than 10–15 %), it is recommended to reduce instrumentation overhead as the measurement may no longer reflect the actual runtime behavior of the application. Typical measures to reduce the runtime overhead include turning off automatic compiler instrumentation, filtering out short but frequently called functions, and applying manual instrumentation using specific application programming interfaces (APIs) provided by the tools.

Health check protocol

A typical workflow when performing an initial "health examination" on a scientific code can be described as follows.

1. "Which function(s) or code region(s) in my program consume(s) the most wall clock time?" This question can usually be answered using a flat profile, which breaks down the application code into separate functions or manually annotated source code regions (e.g., initialization vs. solver phase) and aggregates the wall clock time spent in each function or region. This ascertains the area(s) of interest in order to streamline performance analysis efforts.

   Typical diagnosis tools include Allinea Performance Reports, Score-P + Cube, and Extrae + Paraver.

2. "Does my application scale as expected?" Typically all scientific applications aim to perform well at scale. To address this question, profiles need to be collected with varying numbers of processes, and the scalability of functions and code regions within the areas of interest need to be examined.

   There are two types of scaling: strong and weak scaling. In the case of strong scaling, the overall problem size (workload) stays fixed but the number of processes increases. Here, the runtime is expected to decrease with increasing number of processes. By contrast, in the case of weak scaling, the workload assigned to each process remains constant with an increase in processors and, thus, the runtime is ideally expected to be constant as well.

   The strong scaling efficiency, E_{ss}, is computed as a relation of speedup to the number of processes. Speedup is computed as a relation of the amount of time to complete a work unit with one process to the amount of time to complete N of the same work units with N processes:

$$E_{\mathrm{ss}} = \frac{T_1}{NT_N}, \tag{1}$$

   where T_1 is the time taken to complete a work unit with one process and T_N is the time taken to complete a work unit with N processes.

   The weak scaling efficiency, E_{ws}, is computed as a relation of the amount of time to complete a work unit with one process to the amount of time to complete N of the

same work units with N processes:

$$E_{\mathrm{ws}} = \frac{T_1}{T_N}. \tag{2}$$

E_{ss} and E_{ws} are very common metrics in HPC to quantify and qualify the scalability of the application. These metrics indicate how efficient an application is when using increasing numbers of processes.

Typical diagnosis tools include Allinea Performance Reports, Score-P + Cube, and Extrae + Paraver.

3. "Does my program suffer from load imbalance?" If this is the case, some processes will perform significantly more or less work than the others. Load balance is an indication of how well the load is distributed across processors. If a code is not well balanced, HPC resources will be used inefficiently as imbalances usually materialize as wait states in communication–synchronization operations among processes and threads. Thus, this may be an area on which to concentrate code refactoring efforts.

   To characterize load imbalance Rosas et al. (2014) invented the load balance efficiency metric, E_{lb}, which is defined as a relation between average computation, $\overline{T}$, and maximal computation time, $T_{\max}$:

$$E_{\mathrm{lb}} = \frac{\overline{T}}{T_{\max}}. \tag{3}$$

   Note that load imbalance can either be static or dynamic. While the former can usually be easily identified in profiles, pinpointing the latter may require more heavyweight measurement and analysis techniques such as event tracing, as imbalances may cancel each other out in aggregated profile data.

   Typical diagnosis tools include Score-P + Cube, Score-P + Scalasca + Cube, and Extrae + Paraver.

4. "Is there a disproportionate time spent in communication or synchronization?" Communication and synchronization overheads can be caused by network latencies (e.g., due to an inefficient process placement onto the compute resources) or wait states and other inefficiency patterns (e.g., caused by load or communication imbalances). If these overheads increase significantly with an increase in resources, this can be a further barrier to scalability.

   To quantify and qualify disproportionate time spent in communication or synchronization, Rosas et al. (2014) developed the communication efficiency metric, E_{com}, which is defined as a relation between $T_{\max}$ and total execution time, T_{tot}:

$$E_{\mathrm{com}} = \frac{T_{\max}}{T_{\mathrm{tot}}}. \tag{4}$$

   Typical diagnosis tools include Allinea Performance Reports, Score-P + Scalasca + Cube, and Extrae + Paraver.

5. "Is my parallelization strategy efficient?" To answer this question, Rosas et al. (2014) developed an auxiliary efficiency metric, parallel efficiency, which quantifies and qualifies the parallelization strategy as a whole.

   Parallel efficiency is computed as the product of the previously defined metrics load balance efficiency (step no. 3) and communication efficiency (step no. 4):

$$E_{\mathrm{par}} = E_{\mathrm{lb}} E_{\mathrm{com}} = \frac{\overline{T}}{T_{\mathrm{tot}}}, \tag{5}$$

   where a minor value reduction of any component will result in a significant reduction of parallel efficiency.

   Efficiency values range from zero to 1, where a value of 1 is the most efficient. Using Cube's derived metric feature (Zhukov et al., 2015), we can derive these efficiency metrics from the Score-P profile data automatically.

6. "Is my application limited by resource bounds?" There are several bounds one can reach, such as

   (a) CPU bound; i.e., the rate at which processes operate is limited by the speed of the CPU. For example, a tight loop that can be vectorized and operates only on a few values held in CPU registers is likely to be CPU bound.

   (b) cache bound; i.e., the simulation is limited by the amount and the speed of the cache available. For example, a kernel operating on more data than can be held in registers but which fits into the cache is likely to be cache bound.

   (c) memory bound; i.e., the simulation is limited by the amount of memory available and/or the memory access bandwidth. For example, a kernel operating on more data than fit into the cache is likely to be memory bound.

   (d) I/O bound; i.e., the simulation is limited by the speed of the I/O subsystem. For example, counting the number of lines in a file is likely to be I/O bound.

   Typical diagnosis tools include Score-P + PAPI + Cube, Extrae + Paraver, Intel Advisor, and Darshan.

7. There are additional questions one can add to the survey, for example, "how many pipeline stalls, cache misses, and mis-predicted branches are occurring?", "how can we assess serial performance?", etc.

   To describe serial performance, for example, we use the instructions per cycle metric (IPC), i.e., the ratio of total

instructions executed to the total number of CPU cycles. Potential reasons for low IPC values are pipeline stalls, cache misses, and mis-predicted branches (John and Rubio, 2008). Therefore, additional measurements with hardware counters should be made to determine the number of cache misses in L1, stalls, and mis-predicted branches when a low IPC value is computed.

Typical diagnosis tools include Score-P + PAPI + Cube, Extrae + Paraver, Intel Advisor, Darshan, etc.

As can be seen, different tools can be used to answer the various questions mentioned above. However, they usually employ different measurement and analysis techniques, which may prohibit the use of a particular tool under certain circumstances. For example, the combination Extrae + Paraver uses event tracing in conjunction with a manual–visual analysis, which is only applicable to selected parts of the execution (i.e., the current region of interest). In contrast, Score-P profiling + Cube uses a more lightweight measurement technique that can handle arbitrarily long executions but requires a second measurement if there is a need for a focused in-depth analysis based on event tracing. Moreover, the level of detail provided by the various tools usually differs. For example, Allinea Performance Reports can provide a very coarsely grained initial performance overview. Such an overview can be sufficient to already rule out certain classes of performance issues but does not provide enough detail to track down the root causes of the issues being identified. Thus, it can only give an indication on which more in-depth analysis to carry out in the next step(s). Also, not all performance analysis tools are available on every platform. For example, Allinea Performance Reports and Intel Advisor are only available on x86 architectures, but not for the IBM Blue Gene/Q platform used in this case study. Finally, if two tools provide comparable functionality, users are inclined to use the tool(s) they feel most comfortable with. Thus, our RCF implements support for all of the diagnosis tools mentioned in the section above (see Appendix A for more details), covering many different use cases and preferences.

In this case study, we followed the health check using the diagnosis tools (co)developed by JSC and available on the JUQUEEN platform, namely (1) Score-P profile measurements, including hardware performance counters collected via PAPI (Moore et al., 2003), (2) Score-P trace measurements followed by a subsequent automatic Scalasca trace analysis, and (3) manual analysis of measurements from the abovementioned steps with an interactive visual browser, i.e., Cube. Where Score-P is a community-maintained scalable instrumentation and performance measurement infrastructure for parallel codes that can collect both profiles and event traces, Scalasca Trace Tools are a collection of scalable trace-based tools for in-depth analyses of concurrent behavior, and Cube is an interactive analysis report explorer for Score-P profiles and Scalasca trace analysis reports. Additionally we used Darshan, a tool to capture and characterize the I/O behavior of an application, for I/O profiling. Both Score-P and Scalasca output their results in CUBE4 format, which can be processed by the Cube GUI and command-line tools. The latter are used by the RCF to process the result files and to collate specific information from each run in tabular format.

```
result:
              time[s] |       4
           time_io[s] |     0.1
          time_mpi[s] |     0.7
          mem_vs_cmp  |     1.0
    load_imbalance[%] |     9.0
       io_volume[MB]  |   183.1
            io_calls  |       0
  io_throughput[MB/s] |  1624.4
   avg_io_ac_size[kB] |     0.0
        num_p2p_calls |   14128
     p2p_comm_time[s] |     0.4
 p2p_message_size[kB] |     8.1
       num_coll_calls |    1008
    coll_comm_time[s] |     0.2
coll_message_size[kB] |     0.4
         delay_mpi[s] |     0.4
   delay_mpi_ratio[%] |    59.3
          time_omp[s] |
         omp_ratio[%] |     nan
         delay_omp[s] |
   delay_omp_ratio[%] |     nan
     memory_footprint | 92228kB
cache_usage_intensity |    0.65
                IPC   |    1.53
       time_no_vec[s] |       5
              vec_eff |    1.25
       time_no_fma[s] |       4
              fma_eff |    1.00
```

Figure 3. Example output from a performance metric extraction workflow. See Appendix B for a complete explanation of these performance metrics and why they might be useful.

In order to track the health of ParFlow with each new release, we developed an automated performance metric extraction workflow and integrated this into our RCF to obtain key performance indicators such as MPI wait time, memory footprint, cache intensity, etc. in order to quickly assess whether new developments or additions to the code improve or degrade the overall performance. An example of such a workflow output is given in Fig. 3. Metrics shown in Fig. 3 not only describe the application in general but also assess potential bottlenecks, i.e., I/O, communication, node and core performance, and memory usage (see Appendix B for more details).

2.5 RCF: provenance tracking

JUBE has many provenance tracking features and tools. JUBE automatically stores the benchmark suite data for each workflow execution, which can be parsed by JUBE's analysis tools. Workflow metadata are automatically parsed by JUBE and then compiled into a report detailing the run and which settings were used for each suite. Subsequent analysis procedures can be predefined, added, or altered by the

user after the experiment to automate data processing. These features and tools are designed to facilitate documentation and archiving. Additionally, JUBE's workflow execution directory structure allows for runtime provenance tracking. JUBE's workflow management system automatically creates a suite of the parameter sets and steps for each workflow. JUBE then creates a unique execution unit or work package for a specific step and parameter combination. Each workflow execution has its own directory named by a unique numeric identifier, which is incremented for subsequent runs. Inside this directory, JUBE handles the workflow execution's metadata and creates a directory for each separate work package. This avoids interference among different work package runs and creates a reproducible structure. For dependent work packages, symbolic links are created to the parent work package, for user access.

We added extra provenance tracking features to ParFlow simulation runs such as configuration management (e.g. logging of time spent in important routines, performance tracking) and postprocessing of output to a standardized format enriched with metadata. The postprocessing of output entails converting unannotated ParFlow binary file model simulation output to a more portable NetCDF output containing standardized metadata enrichment using CMOR and CF standards (Eaton et al., 2009; Nadeau et al., 2017) and incorporation of all ParFlow model settings. The CF conventions for climate and forecast metadata are designed to promote the processing and sharing of files created in NetCDF format. The conventions define metadata that provide a definitive description of what the data in each variable represent, and of the spatial and temporal properties of the data. Use of the convention ensures that users of data from different sources can properly compare quantities, along with facilitating interoperability and portability. Interoperability in this case means that CMORized NetCDF files can be used on different architectures (big or little endian) and for different software (for use in various terrestrial systems software and visualization software). The data conversion postprocessing step was developed in accordance with state-of-the-art data life cycle management and to maintain interoperability (Stodden et al., 2016).

In addition, the postprocessing and analysis step we developed contains an archive process at the end of the modeling chain, which documents and collates the environmental variables, model input, model simulation scripts, model submission scripts, log files, postprocessed output, and application code in such a fashion that the archived output can be downloaded and rerun following the instructions in the simulation documentation, without need for any additional input, a practice recommended by Hutton et al. (2016). That is, if a user were to untar an output directory they would be able to compile and rerun the simulation, using the XML configuration file with JUBE, on the same machine, without having to obtain information elsewhere. The information contained in the tarred directory would also almost cover points 1–4 from Irving (2016) – an option for the user to reproduce published figures from the postprocessed NetCDF output files produced is not yet built-in. However ensuring the science remains the same on different HPC architectures needs to be considered when porting models. The developers of the software need to employ strategies such as continuous integration on multiple machines to ensure consistency of science across architectures and compilers. Our RCF could also be adapted to facilitate these strategies.

3 Case study: RCF profiling workflow

In this section, we present the experimental design of the test case, to be used in the profiling study, steps we took in porting the test case, and results of the profiling study, which demonstrate the usefulness of our RCF. For this study we use the highly scalable IBM Blue Gene/Q system JUQUEEN. JUQUEEN features a total of 458 752 cores from 1024 PowerPC A2 16-core four-way simultaneous multithreading CPUs in each of the 28 racks and a total of 448 TB main memory with a LINPACK performance of 5.0 petaflops. ParFlow runs under Linux microkernels on compute nodes using IBM XL compilers and a proprietary MPI library and IBM's General Parallel File System (GPFS). All profiling results shown in this paper are the result of running the benchmark described in Sect. 3.1 10 times to obtain an average and make sure the benchmark is running as it should, taking notice of the variance among results. See the tar ball supplied in the Supplement for this paper for the complete RCF used for this case study.

3.1 Case study: experimental design

In order to demonstrate the applicability of the RCF, a weak scaling demonstration study with an idealized overland flow test case was set up for ParFlow (Maxwell et al., 2015). ParFlow (v3.2, https://github.com/parflow) is a massively parallel, physics-based integrated watershed model, which simulates fully coupled, dynamic 2-D and 3-D hydrological, groundwater, and land-surface processes suitable for large-scale problems. ParFlow is used extensively in research on the water cycle in idealized and real data setups as part of process studies, forecasts, data assimilation experiments, hindcasts, and regional climate change studies from the plot scale to the continent, ranging from days to years. Saturated and variably saturated subsurface flow in heterogeneous porous media are simulated in three spatial dimensions using a Newton–Krylov nonlinear solver (Ashby and Falgout, 1996; Jones and Woodward, 2001; Maxwell, 2013) and multigrid preconditioners, for which the three-dimensional Richards equation is discretized based on cell-centered finite differences. ParFlow also features coupled surface–subsurface flow, which allows for hillslope runoff and channel routing (Kollet and Maxwell, 2006). Because

it is fully coupled to the Common Land Model (CLM), a land surface model, ParFlow can incorporate exchange processes at the land surface including the effects of vegetation (Maxwell and Miller, 2005; Kollet and Maxwell, 2008). Other features include a parallel data assimilation scheme using the Parallel Data Assimilation Framework (PDAF) from Nerger and Hiller (2013), with an ensemble Kalman filter, allowing observations to be ingested into the model to improve forecasts (Kurtz et al., 2016). An octree space partitioning algorithm is used to depict complex structures in three-dimensional space, such as topography, different hydrologic facies, and watershed boundaries. ParFlow parallel I/O is via task-local and shared files in a binary format for each time step. ParFlow is also part of fully coupled model systems such as the Terrestrial Systems Modeling Platform (TerrSysMP) (Shrestha et al., 2014) or PF.WRF (Maxwell et al., 2011), which can reproduce the water cycle from deep aquifers into the atmosphere.

A three-dimensional sinusoidal topography as shown in Fig. 4 was used as the computational domain with a lateral spatial discretization of $\Delta x = \Delta y = 1\,\mathrm{m}$ and a vertical grid spacing of $\Delta z = 0.5\,\mathrm{m}$; the grid size, n, was set to $nx = ny = 50$ and $nz = 40$, resulting in 100 000 unknowns per CPU core, with one MPI task per core. In order to simulate surface runoff from the high to the low topographic regions with subsequent water pooling and infiltration, a constant precipitation flux of $10\,\mathrm{mm\,h^{-1}}$ was applied. This results in realistic nonlinear physical processes and thus compute times. The water table was implemented as a constant head boundary condition at the bottom of the domain with an unsaturated zone above, 10 m below the land surface. The heterogeneous subsurface was simulated as a spatially uncorrelated, log-transformed Gaussian random field of the saturated hydraulic conductivity with a variance ranging over 1 order of magnitude. The soil porosity and permeability were set to $0.25\,\mathrm{m\,day^{-1}}$. This idealized setup was used for the profiling case study as opposed to a real-world setup due to the symmetry inherent in the setup. In contrast, a real-world experiment has asymmetry in both the meteorological forcing and also the model topography, which naturally lead to load imbalances. These asymmetries could therefore obscure whether there are actually load imbalances due to poor software design.

The weak scaling experiment is defined as how the solution time varies with the number of processors for a fixed problem size of 100 000 degrees of freedom per processor. The horizontal (nx, ny) grid size is increased but the number of cells in the vertical direction, nz, remain constant. All model configurations were run for 10 h with a time step size of $\Delta t = 0.5\,\mathrm{h}$.

3.2 Porting ParFlow to JUQUEEN

When porting ParFlow onto JUQUEEN, we used the IBM XL C compiler available on the platform (v12.1), which provides several compiler options that can help control the optimization and performance of C programs. We focused on two aspects of optimization, namely, loop optimization (`-qhot`) and general optimization levels: `-O1` to `-O3`, where these optimizations range from local basic block to whole-program analysis. The higher the optimization level, the more sophisticated optimization techniques are that are applied. For example, using optimization level `-O1` performs only quick local optimizations such as constant folding and elimination of local common subexpressions, whereas optimization level `-O3` performs rewrites of floating point expressions, aggressive code motion, scheduling on computations, and loop optimizations, and additionally the compiler replaces any calls in the source code to standard math library functions with calls to the equivalent MASS library functions. The focus on these two aspects was a result of following the user guidelines set out by IBM in using the XL C compiler (IBM, 2012).

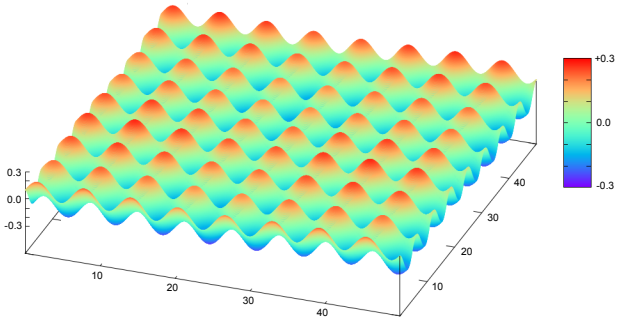

Figure 4. Model setup, showing cross-sectional domain and sinusoidal topography variation from the top of the model ($z = 20$) for each processor.

We set up the accuracy test described in Sect. 2.3 with the gold standard output to test against obtaining results previously generated with the model described in Sect. 3.1. The gold standard output was verified via inbuilt water balance and energy balance tests using ParFlow pftools (a package of utilities and a Tcl library that is used to set up and post-process ParFlow binary files). Accuracy was determined to be met when the output generated did not vary with the gold standard to six significant figures.

We found the optimal commonly used compilation flag, which did not compromise accuracy with the IBM XL C compiler for ParFlow to be `-O3 -qhot -qarch=qp -qtune=qp`, where the architecture and tuning flag was set to be `qp`, which indicates the specific architecture of JUQUEEN (see Table 1). Using the `-O3` compilation flag resulted in a speedup of close to a factor of 2 when running with 16 MPI tasks on 16 CPU cores (one compute node on Blue Gene/Q, no multithreading). The timing results were compiled using JUBE's result parser functionality, which was run as a postprocessing step. This is in agreement with the results in running the fully coupled TerrSysMP model system on JUQUEEN in Gasper et al. (2014).

Table 1. Time taken to run the ParFlow test case on JUQUEEN (IBM Blue Gene/Q) with 16 MPI processes using three different commonly used compiler flag optimizations.

IBM XL compiler flags	Time (s)
`-O1 -qhot -qarch=qp -qtune=qp`	203
`-O2 -qhot -qarch=qp -qtune=qp`	203
`-O3 -qhot -qarch=qp -qtune=qp`	110

3.3 Profiling results and analysis

The following section describes the results from the demonstration scaling study, following the code performance health check protocol given in Sect. 2.4.

3.3.1 Analysis of time spent in ParFlow functions: health check step no. 1

As a first step, a breakdown of the time spent in each annotated region of ParFlow was obtained via internal timings in ParFlow and a Score-P profile measurement, visualized as a bar chart in Fig. 5. From the breakdown it is clear that the core component of ParFlow is the computation of the solution to a system of nonlinear equations, reflected in Fig. 5, where most of the wall clock time is spent in the blue regions which make up the time spent obtaining a solution via a nonlinear solving step. A large part of the nonlinear solver's workflow can be summarized in two steps, which are as follows: the initialization of the problem for the specific input and the actual solver loop. The last two steps reside in the `KINSol` routine, which is a component of the SUNDIALS solver library (Hindmarsh et al., 2005). Therefore, the nonlinear solving routine and its components are the focus of interest for reducing ParFlow's runtime. The nonlinear solving loop performs the computational process of computing an approximate linear solution (`KINSpgmrSolve`), where the intermediate solution is updated every iteration until the desired convergence or tolerance is reached. Those two aforementioned steps within the nonlinear solve loop are manually annotated in the source code and we will focus on these for our results. For simplicity, we have shortened these two steps to `setup_solver` and `solver_loop` respectively and will refer to them by this nomenclature henceforth.

3.3.2 Scalability: health check step no. 2

Our scalability analysis of ParFlow is again based on the Score-P profile measurement. Figure 6 shows a plot of the execution time versus the number of MPI processes when running the weak scaling experiment as outlined in Sect. 3.1, broken down into the two regions of interest: `setup_solver` and `solver_loop`. The behavior of both regions shows an increase in execution time with an increase in the number of processes, though the `setup_solver` region shows better performance in comparison to the `solver_loop`. However, the strong scaling efficiency profile is comparable to similar codes (Mills et al., 2007).

Table 2. Weak scaling efficiency, load balance efficiency, communication efficiency, and parallel efficiency, running the weak scaling experiment up to 32 768 MPI processes.

No. of MPI processes	Weak scaling efficiency	Load balance efficiency	Communication efficiency	Parallel efficiency (%)
1024	1.0	0.96	0.97	0.93
2048	0.84	0.97	0.97	0.94
4096	0.84	0.97	0.82	0.80
8192	0.55	0.98	0.75	0.73
16 384	0.50	0.98	0.69	0.67
32 768	0.21	0.98	0.64	0.63

Upon examination of the results shown in Table 2, at 32 768 MPI processes, the weak scaling efficiency E_{ws} (see Eq. 2) drops to approximately 21 %. To try to ascertain the routines which hinder scalability, further inspection of the breakdown between computation and communication (Fig. 7) shows that total communication considerably increases at scale. This indicates that communication could be a scalability breaker and should be investigated further (see health check step no. 4).

3.3.3 Load balance: health check step no. 3

Table 2 shows that the load balance efficiency at the different scales is between 0.96 and 0.98. This means that the workload is equally distributed among the processes for our idealized test case; that is, there is no inherent load imbalance issue in the algorithm.

3.3.4 Communication: health check step no. 4

Values of communication efficiency shown in Table 2 reduce from 0.97 (1024 processes) to 0.64 (32 768 processes). This means that time spent in communication grows at scale. Therefore, it is worthwhile taking a closer look at what is happening (Figs. 7 and 8).

Time spent in communication grows with increasing number of processes. For example, the communication time constitutes 37 % of the total time when running the test case with 32 768 MPI processes. The main contributors to communication time within the regions of interest are `MPI_Allreduce` in `setup_solver` and `MPI_Waitall` in `solver_loop`. However, the main communication problem is outside of `setup_solver` and `solver_loop`, e.g., in the initialization phase or the preconditioner as `setup_solver` and `solver_loop` communication time do not contribute much. The slight increase in communication time in those two routines could

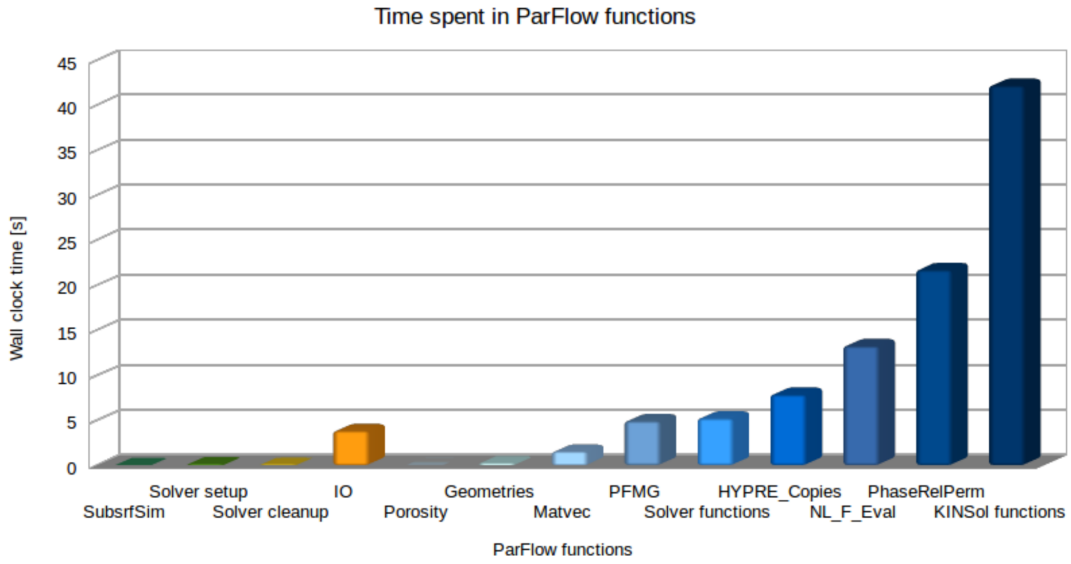

Figure 5. Time spent in ParFlow functions or routines, where the functions or routines can be divided into four categories: setup, cleanup, I/O, and solve. The functions in the category "setup" are depicted in green: SubsrfSim – setting up the domain; Solver setup – initializing the solver. The functions in the category "cleanup" are depicted in yellow: Solver cleanup – finalizing the solver. The functions in the category "I/O" are depicted in orange: PFB I/O – ParFlow binary I/O. The functions in the category "solve" are depicted in blue: Porosity – calculation of the porosity matrix; Geometries – calculation of the simulation domain; MatVec – matrix and vector operations; PFMG – geometric multigrid preconditioner from HYPRE, Solver functions – miscellaneous functions; HYPRE_Copies – copying data within HYPRE; NL_F_Eval – setting up the physics and field variables for the next iteration; PhaseRelPerm – setting up the permeability matrix; KINSol functions – nonlinear solver functions from SUNDIALS.

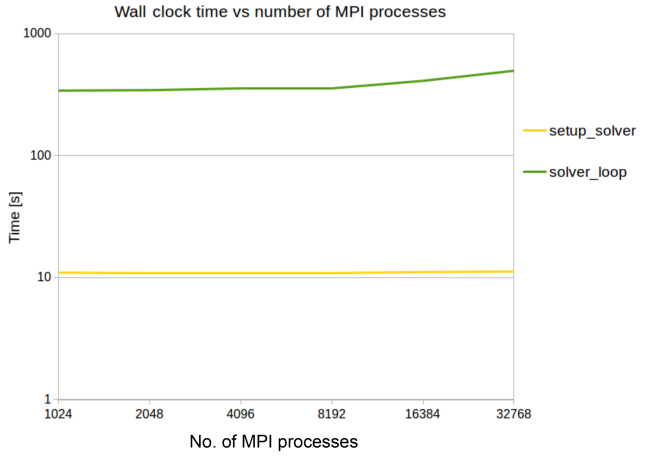

Figure 6. Execution time versus the number of MPI processes for the regions of interest, running the weak scaling experiment up to 32 768 MPI processes.

be attributed to `MPI_Allreduce` in `setup_solver` and `MPI_Waitall` in `solver_loop` by further breaking down the communication routines (see Fig. 8). A trace analysis using Scalasca identified a costly wait-state pattern constituting 23 % of total time in the initialization phase occurring in `MPI_Allreduce` in the preconditioner of the HYPRE v2.10.1 library. This is an example in which more in-depth analysis is needed after the initial health examination to clarify which part of the code must be improved.

3.3.5 Assessing the parallelization strategy: health check step no. 5

To assess the parallelization strategy of ParFlow as a whole it is necessary to perform step nos. 3 and 4. Now we are ready to compute the parallel efficiency, which is shown in Table 2. We can see that values decrease from 0.93 (1024 processes) to 0.63 (32 768 processes), where the loss in communication efficiency is the main cause of the reduction in parallel efficiency.

3.3.6 Resource bounds: health check step no. 6

Using the idealized weak scaling test case described in Sect. 3.1, Score-P was used to track memory usage of the test case on JUQUEEN. Due to the idealized behavior (symmetry) of the test case, all MPI ranks needed roughly the same amount of memory. For example, at 1024 processes, each rank needed roughly 95 MB and at 32 678 processes roughly 325 MB. Memory usage per MPI rank increases with scale as the mesh manager in ParFlow is implemented in such a way that the entire grid information is redundantly stored on each MPI rank. This becomes a scalability breaker for ParFlow as we can see from Fig. 9; there is a point at which the memory required will eclipse the mem-

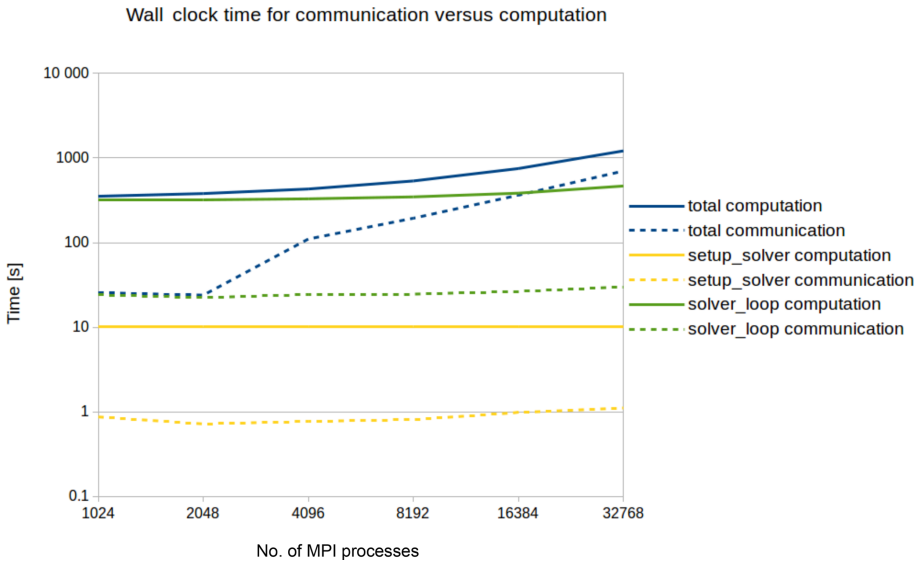

Figure 7. Wall clock time for communication versus computation for the regions of interest for the weak scaling experiment using the test case described in Sect. 3.1.

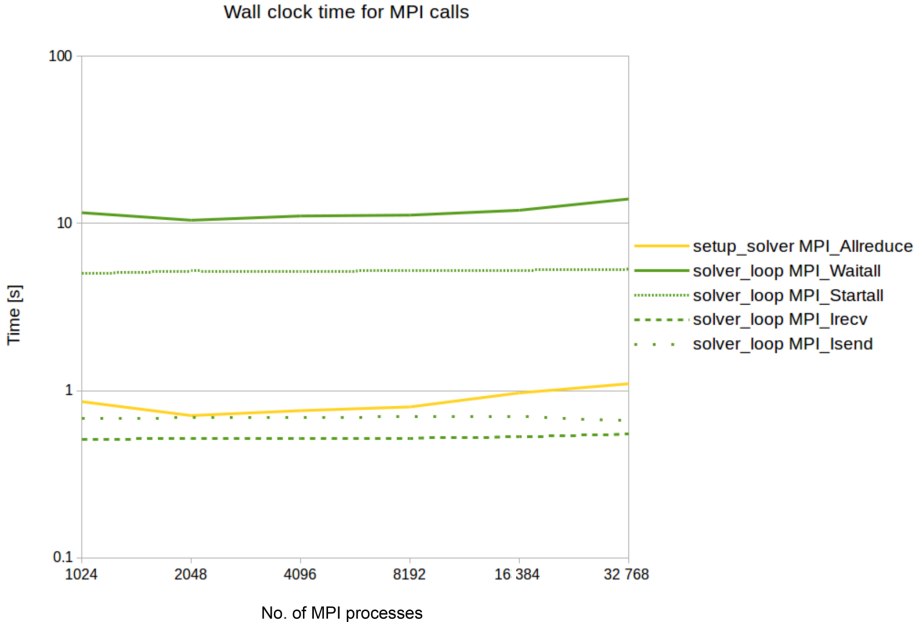

Figure 8. Wall clock time for MPI calls for the weak scaling experiment using the test case described in Sect. 3.1.

ory available (at around 64 000 cores for this test case) and this is due to storage of the grid information for the mesh manager. In ParFlow, the most memory consuming routines are `GetGridNeighbors`, `PFMGInitInstanceXtra`, `KinSolPC`, and `AllocateVectorData`.

3.3.7 Serial performance: health check step no. 7

A Score-P profile measurement with hardware performance counters was used to inspect serial performance (IPC). The serial performance for the test case (Sect. 3.1) with 1024 MPI processes shows lower-than-ideal values of IPC. For example, `solver_loop` has an IPC value of 0.31 out of 2 (the theoretical limit on the Blue Gene/Q platform).

Therefore, additional measurements with hardware counters were collected, which show that a significant number of cache misses in L1, stalls, and mis-predicted branches occur in the following routines: `RichardsJacobianEval`, `PhaseRelPerm`, `Saturation`, and `NlFunctionEval`. Since JUQUEEN is based on an in-order instruction execution model, mean-

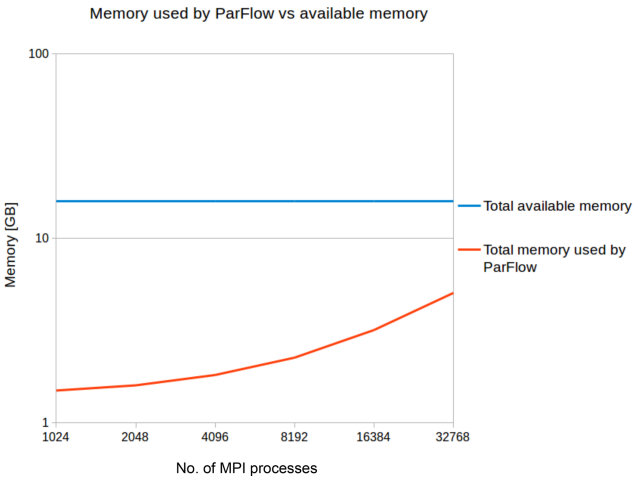

Figure 9. Memory usage of the weak scaling experiment described in Sect. 3.1 vs. the total amount of memory available.

ing instructions are fetched, executed, and committed in compiler-generated order, in the case of an instruction stall, all ensuing instructions will stall as well. Branching on JUQUEEN is therefore very expensive and can cause pipeline stalls. Thus, the aforementioned routines may account for the low IPC values.

3.4 Reproducibility

All simulation runs in the scaling study are separated into different subdirectories for each simulation run. Each subdirectory includes the environment description, the XML scripts used by JUBE, the compilation scripts, the job submission scripts, the job logs, the model scripts, the postprocessing analysis, and a description of the version of ParFlow used along with the ParFlow binary itself. Each directory is self-contained such that the model can be rerun on JUQUEEN without using any other external tools or files. After the simulation is run and the postprocessing step has been executed, the directory is automatically archived for long-term storage.

3.5 Outcomes of profiling case study and future developments using RCF

The detailed profiling work illuminated the main bottlenecks to scalability. To ensure that ParFlow can scale to machines that are in the same bracket as JUQUEEN, or higher, memory use, time spent in communication, and time spent in acquiring the solution for each time step need to be addressed.

To reduce the memory usage and to reduce the time spent in communication, an adaptive mesh refinement (AMR) library, p4est, is currently being implemented into ParFlow to function as the parallel mesh manager. The approach was minimally invasive and preserves most of ParFlow's data structures, the configuration system, and the setup and solver pipeline. The current mesh manager is a barrier to scalability as it requires that all cells store information about every other cell. This is reduced to neighboring cells under p4est, which results in a decrease in memory use (storage reduction) and a decrease in time spent in communication (communication reduced to neighboring cells only), allowing ParFlow to scale over all 458,752 cores on JUQUEEN (Burstedde et al., 2018). Using p4est as the parallel mesh manager has the additional potential benefit of integrating the adaptive mesh refinement functionality into ParFlow in order to address inactive regions (due to heterogeneous forcing, permeability, etc.), causing load imbalances in the real-world models.

To further improve simulation runtimes using ParFlow, the RCF is used to benchmark different accelerator-enabled numerical libraries, for a simplified version of ParFlow, across different HPC architectures. To reduce time spent in preprocessing model input and postprocessing model output, a NetCDF reader and writer is under development, with testing of this new feature integrated into the RCF. There is still room for improvement with regards to serial performance. However, to tackle this problem effectively, more in-depth profiling is needed with the aid of performance analysis engineers. For example, we are currently working in conjunction with performance analysis specialists to identify and refactor individual loops in specific functions for vectorization in order to speed up serial performance. Naturally, we will use our RCF to then validate the effectiveness of these new developments and tuning efforts.

4 Conclusions

Adapting to new developments in HPC architectures, software libraries, and infrastructures, while ensuring reproducibility of simulation and analysis results has become challenging in the field of geoscience. Next-generation massively parallel HPC systems require new coding paradigms, and next-generation geoscientific models are based on complex model implementations and profiling, modeling, and data processing workflows. Thus there is a strong need for a streamlined approach to model simulation runs, including profiling, porting, and provenance tracking.

In this article, we presented our run control framework as a best practice approach to porting, profiling, and provenance tracking. Implementing an RCF using a workflow engine for the complete modeling chain consisting of preprocessing, simulation run, and postprocessing leads to code that can be ported easily and tuned to any platform and combination of compilers, for which dependencies are available in module format. Each simulation is self-contained and automatically documented, accounting for provenance tracking, which leads to better supplementary code sharing and ultimately reproducibility. The relevant profiling toolset can be applied on any platform where the toolset is available, leading to identification of bottlenecks, code tuning, refactoring,

and ultimately more efficient use of HPC resources. For example, the detailed profiling study of ParFlow led to the identification of bottlenecks and scalability breakers.

The proposed approach helps the novice user as well as the developer and can be embedded into regression testing and a continuous integration approach. Using our RCF, testing, benchmarking, profiling, and running models is less time consuming and more robust than running models in an ad hoc fashion, resulting in more efficient use of HPC resources, more strategic code development, and enhanced data integrity and reproducibility.

Code and data availability. The run control framework, data, and version of ParFlow used in this paper are available for download via https://gitlab.maisondelasimulation.fr/EOCOE/Parflow upon request for access. A tar ball containing the RCF, data, and version of ParFlow relevant to this study has been included in the Supplement.

Appendix A: Profiling tools implemented into the RCF

Table A1 describes the profiling tools which are currently supported by our RCF. New profiling tools can easily be added into the framework by adding to the `profiler.xml` file (see Fig. 2).

Table A1. Description of of the performance analysis tools currently supported by our RCF.

Performance analysis tool	Description
Score-P	Score-P is a community-maintained scalable instrumentation and performance measurement infrastructure for parallel codes. It can collect both profiles and event traces. https://www.score-p.org
Scalasca	Scalasca Trace Tools is a collection of scalable trace-based tools for in-depth analyses of concurrent behavior, in particular regarding wait states in communication and synchronization operations as well as their root causes. Supports Score-P traces from v2.0 forward. https://www.scalasca.org
Cube	Cube is an interactive analysis report explorer for Score-P profiles and Scalasca trace analysis reports. http://www.scalasca.org/software/cube-4.x/
Allinea Performance Report	Allinea Performance Report is a performance tool which provides a high-level overview of the runtime using a single-page report. https://www.allinea.com/products/allinea-performance-reports
Extrae	Extrae is a measurement system which is able to collect traces for use with Paraver. https://tools.bsc.es/extrae
Paraver	Paraver is a flexible and configurable performance analysis tool based on traces collected by the Extrae measurement system. It supports time-line views as well as histogram and statistics views on the trace data. https://tools.bsc.es/paraver
Intel® Advisor	Advisor is a tool to analyze node-level performance issues, in particular regarding code vectorization and threading. https://software.intel.com/en-us/intel-advisor-xe
Darshan	Darshan is tool to capture and characterize the I/O behavior of an application. http://www.mcs.anl.gov/project/darshan-hpc-io-characterization-tool
PAPI	PAPI is a library providing a consistent interface for accessing hardware performance counters of CPUs and other components. While it can be called from application code directly, PAPI is more often used through other performance measurement systems such as Extrae and Score-P. http://icl.utk.edu/papi

Appendix B: Description of performance metrics gathered from the automated performance metric workflow

The list of performance metrics gathered in the automated performance metrics workflow (see Fig. 2), with an explanation of why these particular metrics are useful, is given in the tables below. Table B1 contains general performance metrics and Tables B2–B5 contain performance metrics pertaining to specific areas such as I/O, communication, memory use, node-level performance, and core-level (serial) performance.

Table B1. Description of general performance analysis metrics currently supported by our RCF.

General performance metric	Description
Time (s)	Total application wall time to use as a reference.
time_io (s)	Average time spent in input–output operations for each rank. When this value is large compared to the overall runtime, the application spends a significant amount of time in input–output operations. Further measurements can be taken to illuminate problem areas (see I/O section of Table B2).
time_mpi (s)	Average time spent in MPI for each rank. When this value is large compared to the overall runtime, the application spends a significant amount of time in MPI operations. Further measurements can be taken to illuminate problem areas (see MPI section of Table B3).
mem_vs_comp	Memory bound versus compute bound. Memory bound means that the application would be faster if the memory bandwidth was larger and compute bound means that the application would be faster if the CPU was faster. (Close to 1.0 means the application is strongly compute bound, close to 2.0 means the application is strongly memory bound.)
load_imbalance	Ratio of the load imbalance overhead to the critical path duration. This ratio signifies the potential for speedup if the load imbalance were nonexistent. For example, if a 20 % load imbalance is measured, fixing this load imbalance would improve the runtime of the code by 20 %.

Table B2. Description of performance analysis metrics pertaining to I/O, currently supported by our RCF.

I/O performance metric	Description
io_volume (MB)	Total number of data in I/O. Can indicate whether I/O is going to be a bottleneck for the application.
io_calls (nb)	Total number of I/O calls. Can indicate whether the I/O subroutines are inefficient.
io_throughput (MB s^{-1})	Speed of I/O. Can indicate whether the HPC architecture is suitable for the application.
avg_io_ac_size (kB)	Average number of data per I/O call. Can indicate whether performance could be improved by changing data size (coalescing reads/writes).

Table B3. Description of performance analysis metrics pertaining to MPI communication, currently supported by our RCF.

MPI performance metric	Description
num_p2p_calls (nb)	Average number of point-to-point MPI operations per MPI rank. Can indicate inefficiency of point-to-point communication pattern.
p2p_comm_time (s)	Average time spent in point-to-point MPI operations per MPI rank. Can indicate inefficiency of point-to-point communication pattern.
p2p_message_size (kB)	Average size of point-to-point MPI messages per MPI rank. Can indicate whether performance could be improved by changing message size.
num_coll_calls (nb)	Average number of collective MPI operations per MPI rank. Can indicate inefficiency of collective communication pattern.
coll_comm_time (s)	Average time spent in collective MPI operations per MPI rank. Can indicate inefficiency of collective communication pattern.
coll_message_size (kB)	Average message size in collective communications per MPI rank. Can indicate inefficiency of collective communication pattern.
delay_mpi (s)	Total amount of MPI time spent in waiting caused by inefficient communication patterns. If this value is large, it signifies that the application has a significant number of delays that cause wait states in MPI operations.
delay_mpi_ratio	Ratio of waiting time caused by MPI to total time spent in MPI. If this value is large, it signifies that the application has a significant number of delays that cause wait states in MPI operations.

Table B4. Description of performance analysis metrics pertaining to memory use, currently supported by our RCF.

Memory performance metric	Description
memory_footprint (kB)	Average memory footprint per MPI rank. This metric helps to estimate the total amount of main memory that the program uses while running.
cache_usage_intensity	Ratio of total number of cache hits to the total number of cache accesses. If this value is small, the application uses the cache inefficiently.

Table B5. Description of performance analysis metrics pertaining to node-level performance currently supported by our RCF.

Node performance metric	Description
time_omp (s)	Total time spent in OpenMP parallel regions. Can indicate load imbalances with regards to inter- and intra-node operations.
omp_ratio (%)	Ratio of the time spent in OpenMP parallel region to the total computation time. Can indicate load imbalances with regards to inter- and intra-node operations.
delay_omp (s)	Total amount of OpenMP synchronization overhead. If this value is large, it signifies that the application has a significant number of delays that cause wait states in OpenMP constructs.
delay_omp_ratio	Ratio of synchronization overhead time in OpenMP to total time spent in OpenMP. If this value is large, it signifies that the application has a significant number of delays that cause wait states in OpenMP constructs.

Table B6. Description of performance analysis metrics pertaining core-level performance currently supported by our RCF.

Core performance metric	Description
IPC	Ratio of total instructions executed to the total number of CPU cycles. This metric shows the workload of the CPU. Low values usually indicate the presence of pipeline bubbles and/or cache misses and/or mis-predicted branches.
time_no_vec (s)	Wall clock time without compiler vectorization.
vec_eff	Ratio of total wall time of the reference run to the total wall time without vectorization. If this value is low, the application is not vectorized or poorly vectorized.
time_no_fma	Total wall time with disabled "fused–multiply–add" (FMA) instructions.
fma_eff	Ratio of total wall time of the reference run to the total wall time without fused–multiply–add (FMA) operations. If this value is low, the application does not use FMA or poorly uses FMA operations.

The Supplement related to this article is available online at https://doi.org/10.5194/gmd-11-2875-2018-supplement.

Competing interests. The authors declare that there are no conflicts of interest.

Acknowledgements. The authors gratefully acknowledge the reviewers thoughtful suggestions and comments, which led to a greatly improved paper. We also acknowledge computing time granted by the JARA-HPC partition for the project: "Fractal Scaling of Hydrodynamics at the Catchment Scale" on the supercomputer JUQUEEN at Forschungszentrum Jülich. This work was supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 76629, funded within the Horizon2020 framework of the European Union and POP (676553), the Centre of Excellence Performance Optimisation and Productivity. In addition we acknowledge the supercomputing support provided to us by the Simulation Laboratory Terrestrial Systems (www.hpsc-terrsys.de/simlab) of the Centre for High Performance Scientific Computing in Terrestrial Systems (Geoverbund ABC/J) and the Jülich Supercomputing Centre (JSC).

The article processing charges for this open-access publication were covered by a Research Centre of the Helmholtz Association.

Edited by: Steve Easterbrook
Reviewed by: three anonymous referees

References

Alonso, P., Badia, R. M., Labarta, J., Barreda, M., Dolz, M. F., Mayo, R., Quintana-Orti, E. S., and Reyes, R.: Tools for Power-Energy Modelling and Analysis of Parallel Scientific Applications, in: 2012 41st International Conference on Parallel Processing, 420–429, https://doi.org/10.1109/ICPP.2012.57, 2012.

Ashby, S. F. and Falgout, R. D.: A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations, Nucl. Sci. Eng., 124, 145–159, 1996.

Attig, N., Gibbon, P., and Lippert, T.: Trends in supercomputing: The European path to exascale, Comput. Phys. Commun., 182, 2041–2046, https://doi.org/10.1016/j.cpc.2010.11.011, 2011.

Bahra, A.: Managing work flows with ecFlow, ECMWF Newsletter,. Tech. Rep., 129, 30–32, 2011.

Bierkens, M. F. P., Bell, V. A., Burek, P., Chaney, N., Condon, L. E., David, C. H., de Roo, A., Döll, P., Drost, N., Famiglietti, J. S., Flörke, M., Gochis, D. J., Houser, P., Hut, R., Keune, J., Kollet, S., Maxwell, R. M., Reager, J. T., Samaniego, L., Sudicky, E., Sutanudjaja, E. H., van de Giesen, N., Winsemius, H., and Wood, E. F.: Hyper-resolution global hydrological modelling: what is next?, Hydrol. Proc., 29, 310–320, https://doi.org/10.1002/hyp.10391, 2015.

Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., and Storaasli, O. O.: State-of-the-art in heterogeneous computing, Sci. Programming, 18, 1–33, https://doi.org/10.3233/SPR-2009-0296, 2010.

Burstedde, C., Fonseca, J. A., and Kollet, S.: Enhancing speed and scalability of the ParFlow simulation code, Comput. Geosci., 22, 347–361, https://doi.org/10.1007/s10596-017-9696-2, 2018.

Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., and Ross, R.: Understanding and Improving Computational Science Storage Access through Continuous Characterization, ACM T. Storage, 7, 1–26, https://doi.org/10.1145/2027066.2027068, 2011.

Davis, N. E., Robey, R. W., Ferenbaugh, C. R., Nicholaeff, D., and Trujillo, D. P.: Paradigmatic shifts for exascale supercomputing, J. Supercomput., 62, 1023–1044, https://doi.org/10.1007/s11227-012-0789-3, 2012.

Eaton, B., Gregory, J., Drach, R., Taylor, K., and Hankin, S.: NetCDF Climate and Forecast (CF) Metadata Conventions, available at: http://cfconventions.org/cf-conventions/v1.6.0/cf-conventions.html (last access: 2 July 2018), 2009.

Eyring, V., Bony, S., Meehl, G. A., Senior, C. A., Stevens, B., Stouffer, R. J., and Taylor, K. E.: Overview of the Coupled Model Intercomparison Project Phase 6 (CMIP6) experimental design and organization, Geosci. Model Dev., 9, 1937–1958, https://doi.org/10.5194/gmd-9-1937-2016, 2016.

Gasper, F., Goergen, K., Shrestha, P., Sulis, M., Rihani, J., Geimer, M., and Kollet, S.: Implementation and scaling of the fully coupled Terrestrial Systems Modeling Platform (TerrSysMP v1.0) in a massively parallel supercomputing environment – a case study on JUQUEEN (IBM Blue Gene/Q), Geosci. Model Dev., 7, 2531–2543, https://doi.org/10.5194/gmd-7-2531-2014, 2014.

Geimer, M., Wolf, F., Wylie, B. J. N., Ábrahám, E., Becker, D., and Mohr, B.: The Scalasca performance toolset architecture, Concurr. Comp.-Pract. E., 22, 702–719, https://doi.org/10.1002/cpe.1556, 2010.

Hammond, G. E., Lichtner, P. C., and Mills, R. T.: Evaluating the performance of parallel subsurface simulators: An illustrative example with PFLOTRAN, Water Resour. Res., 50, 208–228, https://doi.org/10.1002/2012WR013483, 2014.

Han, X., Hendricks Franssen, H.-J., Jiménez Bello, M. Á., Rosolem, R., Bogena, H., Alzamora, F. M., Chanzy, A., and Vereecken, H.: Simultaneous soil moisture and properties estimation for a drip irrigated field by assimilating cosmic-ray neutron intensity, J. Hydrol., 539, 611–624, https://doi.org/10.1016/j.jhydrol.2016.05.050, 2016.

Heinzeller, D., Duda, M. G., and Kunstmann, H.: Towards convection-resolving, global atmospheric simulations with the Model for Prediction Across Scales (MPAS) v3.1: an extreme scaling experiment, Geosci. Model Dev., 9, 77–110, https://doi.org/10.5194/gmd-9-77-2016, 2016.

Hethey, J. M.: GitLab repository management: delve into managing your projects with GitLab, while tailoring it to fit your environment, Packt Pub,Birmingham, 88 pp., ISBN: 9781783281794, 2013.

Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S.: SUNDIALS, ACM T. Math. Software, 31, 363–396, https://doi.org/10.1145/1089014.1089020, 2005.

Hutton, C., Wagener, T., Freer, J., Han, D., Duffy, C., and Arheimer, B.: Most computational hydrology is not reproducible,

so is it really science?, Water Resour. Res., 52, 7548–7555, https://doi.org/10.1002/2016WR019285, 2016.

Hwu, W.-M.: What is ahead for parallel computing, J Parallel Distr. Com., 74, 2574–2581, https://doi.org/10.1016/j.jpdc.2014.02.005, 2014.

IBM: IBM XL C/C++ for Blue Gene/Q: Compiler Reference, version 12.1, IBM Corporation ,available at: http://www-01.ibm.com/support/docview.wss?uid=swg27027065&aid=1 (last access: 1 July 2018), 2012.

Irving, D.: A Minimum Standard for Publishing Computational Results in the Weather and Climate Sciences, B. Am. Meteorol. Soc., 97, 1149–1158, https://doi.org/10.1175/BAMS-D-15-00010.1, 2016.

January, C., Byrd, J., Oró, X., and O'Connor, M.: Allinea MAP: Adding Energy and OpenMP Profiling Without Increasing Overhead, in: Tools for High Performance Computing 2014, Springer International Publishing, Cham, 25–35, https://doi.org/10.1007/978-3-319-16012-2_2, 2015.

John, E. and Rubio, J.: Unique chips and systems, CRC Press, 34–35, 2008.

Jones, J. E. and Woodward, C. S.: Newton–Krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems, Adv. Water Resour., 24, 763–774, https://doi.org/10.1016/S0309-1708(00)00075-0, 2001.

Kandalla, K., Mendygral, P., Radcliffe, N., Cernohous, B., Knaak, D., McMahon, K., and Pagel, M.: Optimizing Cray MPI and SHMEM Software Stacks for Cray-XC Supercomputers based on Intel KNL Processors, Proceedings of 2016 Cray User Group (CUG), 2016.

Keune, J., Kollet, S., Sulis, M., Shresta, P., Görgen, K., and Ohlwein, C.: Implementation of a coupled soil-vegetation-atmosphere system over the European CORDEX domain, Hans-Ertel Centre for Weather Research workshop 2013, Bonn, Germany, Climate Monitoring Branch, Centre for High-Performance Scientific Computing in Terrestrial Systems, 2013.

Keyes, D. E.: Exaflop/s: The why and the how, CR Mecanique, 339, 70–77, https://doi.org/10.1016/j.crme.2010.11.002, 2011.

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., and Wolf, F.: Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir, in: Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, edited by: Brunst, H., Müller, M. S., Nagel, W. E., and Resch, M. M., September 2011, ZIH, Dresden, Germany, Springer, 79–91, https://doi.org/10.1007/978-3-642-31476-6_7, 2012.

Kollet, S. J. and Maxwell, R. M.: Integrated surface–groundwater flow modeling: A free-surface overland flow boundary condition in a parallel groundwater flow model, Adv. Water Resour., 29, 945–958, https://doi.org/10.1016/j.advwatres.2005.08.006, 2006.

Kollet, S. J. and Maxwell, R. M.: Capturing the influence of groundwater dynamics on land surface processes using an integrated, distributed watershed model, Water Resour. Res., 44, W02402, https://doi.org/10.1029/2007WR006004, 2008.

Kurtz, W., He, G., Kollet, S. J., Maxwell, R. M., Vereecken, H., and Hendricks Franssen, H.-J.: TerrSysMP–PDAF (version 1.0): a modular high-performance data assimilation framework for an integrated land surface–subsurface model, Geosci. Model Dev., 9, 1341–1360, https://doi.org/10.5194/gmd-9-1341-2016, 2016.

Labarta, J., Gi Enez, J., Martínez, E., Gon, P., Servat, H., Llort, G., and Aguilar, X.: Scalability of Visualization and Tracing Tools, in: Parallel Computing: Current & Future Issues of High-End Computing, edited by: Joubert, G. R., Nagel, W. E., Peters, F. J., Plata, O., Tirado, P., and Zapata, E., NIC Series Vol. 33, 869–876, ISBN 3-00-017352-8, 2006.

Langdon, W. B., Vilella, A., Lam, B. Y. H., Petke, J., and Harman, M.: Benchmarking genetically improved BarraCUDA on epigenetic methylation NGS datasets and nVidia GPUs, in: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, ACM, Denver, CO, USA, 20–24 July 2016, 1131–1132, https://doi.org/10.1145/2908961.2931687, 2016.

Leutwyler, D., Fuhrer, O., Lapillonne, X., Lüthi, D., and Schär, C.: Towards European-scale convection-resolving climate simulations with GPUs: a study with COSMO 4.19, Geosci. Model Dev., 9, 3393–3412, https://doi.org/10.5194/gmd-9-3393-2016, 2016.

Liu, B., Zydek, D., Selvaraj, H., and Gewali, L.: Accelerating High Performance Computing Applications: Using CPUs, GPUs, Hybrid CPU/GPU, and FPGAs, in: 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies, IEEE, 337–342, https://doi.org/10.1109/PDCAT.2012.34, 2012.

Lührs, S., Rohe, D., Frings, W., Thust, K., and Schnurpfeil, A.: Flexible and Generic Workflow Management, Adv. Par. Com., 27, 431–438, https://doi.org/10.3233/978-1-61499-621-7-431, 2016.

Manubens-Gil, D., Vegas-Regidor, J., Prodhomme, C., Mula-Valls, O., and Doblas-Reyes, F. J.: Seamless management of ensemble climate prediction experiments on hpc platforms, in: 2016 International Conference on High Performance Computing & Simulation (HPCS), IEEE, 895–900, 2016.

Mavroidis, I., Papaefstathiou, I., Lavagno, L., Nikolopoulos, D. S., Koch, D., Goodacre, J., Sourdis, I., Papaefstathiou, V., Coppola, M., and Palomino, M.: ECOSCALE: Reconfigurable computing and runtime system for future exascale systems, in: Proceedings of the 2016 Conference on Design, Automation & Test in Europe, EDA Consortium, 696–701, 2016.

Maxwell, R. M.: A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling, Adv. Water Resour., 53, 109–117, https://doi.org/10.1016/j.advwatres.2012.10.001, 2013.

Maxwell, R. M. and Miller, N. L.: Development of a Coupled Land Surface and Groundwater Model, J. Hydrometeorol., 6, 233–247, https://doi.org/10.1175/JHM422.1, 2005.

Maxwell, R. M., Lundquist, J. K., Mirocha, J. D., Smith, S. G., Woodward, C. S., and Tompson, A. F. B.: Development of a Coupled Groundwater–Atmosphere Model, Mon. Weather Rev., 139, 96–116, https://doi.org/10.1175/2010MWR3392.1, 2011.

Maxwell, R. M., Condon, L. E., and Kollet, S. J.: A high-resolution simulation of groundwater and surface water over most of the continental US with the integrated hydrologic model ParFlow v3, Geosci. Model Dev., 8, 923–937, https://doi.org/10.5194/gmd-8-923-2015, 2015.

Meadows, L.: Experiments with WRF on Intel many integrated core (Intel MIC) architecture, in: OpenMP in a Heterogeneous World, Springer, 130–139, 2012.

Mills, R. T., Lu, C., Lichtner, P. C., and Hammond, G. E.: Simulating subsurface flow and transport on ultrascale computers using PFLOTRAN, J. Phys. Conf. Ser., 78, 012051, https://doi.org/10.1088/1742-6596/78/1/012051, 2007.

Moore, S., Terpstra, D., London, K., Mucci, P., Teller, P., Salayandia, L., Bayona, A., and Nieto, M.: PAPI deployment, evaluation, and extensions, 2003 User Group Conference Proceedings, Bellevue, WA, USA, USA, 9–13 June 2003, IEEE, 349–353, https://doi.org/10.1109/DODUGC.2003.1253415, 2003.

Nadeau, D., Doutriaux, C., Bradshaw, T., Kettleborough, J., Weigel, T., Hogan, E., and Durack, P. J.: Pcmdi/Cmor: Cmor Version 3.2.2, https://doi.org/10.5281/zenodo.345171, 2017.

Nerger, L. and Hiller, W.: Software for ensemble-based data assimilation systems–Implementation strategies and scalability, Comput. Geosci., 55, 110–118, https://doi.org/10.1016/j.cageo.2012.03.026, 2013.

Oliver, H. J., Shin, M., Fitzpatrick, B., Clark, A., Sanders, O., M214089, Smout-Day, K., Matthews, D., Wales, S., Osprey, A., Reinecke, A., Williams, J., Kinoshita, B. P., Pulo, K., and Valters, D.: Cylc/Cylc: Cylc-7.3.0, https://doi.org/10.5281/ZENODO.545663, 2017.

Prein, A. F., Langhans, W., Fosser, G., Ferrone, A., Ban, N., Goergen, K., Keller, M., Tölle, M., Gutjahr, O., Feser, F., Brisson, E., Kollet, S., Schmidli, J., van Lipzig, N. P. M., and Leung, R.: A review on regional convection-permitting climate modeling: Demonstrations, prospects, and challenges, Rev. Geophys., 53, 323–361, https://doi.org/10.1002/2014RG000475, 2015.

Rane, A., Krishnaiyer, R., Newburn, C. J., Browne, J., Fialho, L., and Matveev, Z.: Unification of Static and Dynamic Analyses to Enable Vectorization, Springer, Cham, 367–381, https://doi.org/10.1007/978-3-319-17473-0_24, 2015.

Rigo, A., Pinto, C., Pouget, K., Raho, D., Dutoit, D., Martinez, P.-Y., Doran, C., Benini, L., Mavroidis, I., Marazakis, M., Bartsch, V., Lonsdale, G., Pop, A., Goodacre, J., Colliot, A., Carpenter, P., Radojkovic, P., Pleiter, D., Drouin, D., and de Dinechin, B.: Paving the way towards a highly energy-efficient and highly integrated compute node for the Exascale revolution: the ExaNoDe approach, in: 2017 Euromicro Conference on Digital System Design (DSD), IEEE, 486–493, 2017.

Rosas, C., Giménez, J., and Labarta, J.: Scalability prediction for fundamental performance factors, Supercomputing Frontiers and Innovations, 1, 4–19, https://doi.org/10.14529/jsfi140201, 2014.

Ruti, P. M., Somot, S., Giorgi, F., Dubois, C., Flaounas, E., Obermann, A., Dell'Aquila, A., Pisacane, G., Harzallah, A., Lombardi, E., Ahrens, B., Akhtar, N., Alias, A., Arsouze, T., Aznar, R., Bastin, S., Bartholy, J., Béranger, K., Beuvier, J., Bouffies-Cloché, S., Brauch, J., Cabos, W., Calmanti, S., Calvet, J.-C., Carillo, A., Conte, D., Coppola, E., Djurdjevic, V., Drobinski, P., Elizalde-Arellano, A., Gaertner, M., Galàn, P., Gallardo, C., Gualdi, S., Goncalves, M., Jorba, O., Jordà, G., L'Heveder, B., Lebeaupin-Brossier, C., Li, L., Liguori, G., Lionello, P., Maciàs, D., Nabat, P., Önol, B., Raikovic, B., Ramage, K., Sevault, F., Sannino, G., Struglia, M. V., Sanna, A., Torma, C., and Vervatis, V.: Med-CORDEX Initiative for Mediterranean Climate Studies, B. Am. Meteorol. Soc., 97, 1187–1208, https://doi.org/10.1175/BAMS-D-14-00176.1, 2016.

Saviankou, P., Knobloch, M., Visser, A., and Mohr, B.: Cube v4: From Performance Report Explorer to Performance Analysis Tool, Procedia Comput. Sci., 51, 1343–1352, https://doi.org/10.1016/j.procs.2015.05.320, 2015.

Schwitalla, T., Bauer, H.-S., Wulfmeyer, V., and Warrach-Sagi, K.: Continuous high-resolution midlatitude-belt simulations for July–August 2013 with WRF, Geosci. Model Dev., 10, 2031–2055, https://doi.org/10.5194/gmd-10-2031-2017, 2017.

Shrestha, P., Sulis, M., Masbou, M., Kollet, S., and Simmer, C.: A scale-consistent Terrestrial Systems Modeling Platform based on COSMO, CLM and ParFlow, Mon. Weather Rev., 142, 3466–3483, https://doi.org/10.1175/MWR-D-14-00029.1, 2014.

Stodden, V., McNutt, M., Bailey, D. H., Deelman, E., Gil, Y., Hanson, B., Heroux, M. A., Ioannidis, J. P. A., and Taufer, M.: Enhancing reproducibility for computational methods, Science, 354, 1240–1241, https://doi.org/10.1126/science.aah6168, 2016.

Zhukov, I., Feld, C., Geimer, M., Knobloch, M., Mohr, B., and Saviankou, P.: Scalasca v2: Back to the Future, in: Tools for High Performance Computing 2014, Springer International Publishing, Cham, 1–24, https://doi.org/10.1007/978-3-319-16012-2_1, 2015.