Developing a steering approach to constrain high-resolution water resources modeling over West Africa

Sebastian Friedemann

Master’s Thesis
Developing a steering approach to constrain high-resolution water resources modeling over West Africa

Sebastian Friedemann
Master’s Thesis

Aufgabensteller: Prof. Dr. U. Rüde
Betreuer: D. Thönnes, Dr. T. Pellarin, Dr. habil. B. Raffin
Bearbeitungszeitraum: 1.10.2017 – 31.3.2018
Erklärung:

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

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

Erlangen, den 22. März 2018
Abstract

To get an understanding of the hydrologic processes in West Africa’s critical zone, high-resolution large scale simulations shall be performed. Unfortunately there is not enough knowledge on the hydrodynamic properties of this region. Thus one may rely on an approach using inverse modeling to retrieve simulation input parameters by comparing simulation outputs with observations. Therefore the execution of computational costly analysis on large amounts of data poses a challenging problem.

In Situ and In Transit workflows as well as computational steering try to overcome this by avoiding the file system bottleneck to increase scalability on recent highly parallelized high-performance platforms. For this they enable to perform model analysis and write in the background and online, while the simulation data is still in the memory.

The contribution of this work is to allow In Situ, In Transit and computational steering for the state-of-the-art, physically based, fully coupled critical zone model ParFlow-CLM. To achieve this the FlowVR middleware is used. FlowVR allows to define flexible data flows of distributed applications representing for instance In Situ and In Transit workflows as well as computational steering. These data flows become instantiated on local machines up to high-performance platforms. This increases the simulation’s performance for file system access and analysis on its output. For instance when writing out every timestep to the file system through In Situ on a domain of about 20,000 km$^2$, we experience simulation wall times only slightly longer (<1%, 4 nodes) than the wall times of the same simulation without any file output using the same computational resources.

Also In Situ and In Transit together with computational steering allow online analysis influencing the further model iterations to achieve e.g. nudging and inverse modeling by regulation. We show that this leads to an execution time increase of no more than 10% for exemplary cases.

Finally we facilitate online user interaction with running simulations, simplifying the exploration of hydrological domains and their feedbacks on parameter changes. This diminishes the amount of output data, produced while working with the model.

1i.e. the thin outer layer of the earth where water transfers occur, defined from the impermeable bedrock to the top of vegetation canopy
# Contents

1 Introduction ................................................. 1
   1.1 Motivation .............................................. 1
   1.2 Task ................................................... 2
   1.3 The West Africa Domain ................................. 2
   1.4 Related Work ........................................... 2
   1.5 Supplement ............................................. 4
   1.6 Acknowledgments ........................................ 4

2 Material and Methods ....................................... 5
   2.1 ParFlow ................................................ 5
      2.1.1 The ParFlow hydrological model .................. 5
      2.1.2 Technical Details .................................. 6
      2.1.3 Use Cases and Integration with other Models .... 6
   2.2 In Situ, In Transit, Helper Cores and Staging Nodes ... 7
      2.2.1 In Situ and In Transit Data Processing .......... 7
      2.2.2 Computational Steering ............................. 9
      2.2.3 Helper Cores and Staging Nodes .................. 9
   2.3 Data Assimilation ...................................... 10
      2.3.1 Variational and statistical Data Assimilation ... 10
      2.3.2 Twin Experiments .................................. 12
   2.4 Computation Time Measures ............................. 12
   2.5 FlowVR ............................................... 12
      2.5.1 Introduction ....................................... 12
      2.5.2 Components ....................................... 13
      2.5.3 Rules ............................................. 14
      2.5.4 Performance ...................................... 14

3 Implementation .............................................. 15
   3.1 The parFlowVR Framework .............................. 15
      3.1.1 Framework ......................................... 15
      3.1.2 Messaging Protocol ................................ 15
      3.1.3 Components ....................................... 17
      3.1.4 Synchronization ................................... 19
      3.1.5 Kick off Messages from PreSignals ............... 19
   3.2 The parflow Module .................................... 21
      3.2.1 A Contract System for Data Filtering .......... 21
      3.2.2 ParFlow-ParFlow Communication .................. 22
      3.2.3 Parallel Data Dumps ............................... 22
      3.2.4 Control flow options ............................. 23
   3.3 Output Modules ....................................... 23
      3.3.1 netcdf-writer .................................... 23
      3.3.2 visit-connector .................................. 23
3.4 The analyzer API ................................................................. 24
   3.4.1 Analyzer ................................................................. 24
   3.4.2 C-analyzer API .......................................................... 24
   3.4.3 Python-analyzer API .................................................. 24

4 Experiments ................................................................. 27
   4.1 Experimental Environment ............................................ 27
   4.2 Performance of Massive File I/O ..................................... 27
   4.3 Performance of Variable Dump and Steering Interaction ...... 29
   4.4 Proof of Concept: Constraint Optimization with a Feedback Loop .................................................. 31
       4.4.1 Twin Experiment: Optimize after Convergence of Pressure .................................................. 31
       4.4.2 Calibration of the Proportional Gain $K_p$ .................................................. 32
       4.4.3 Experiment Parameters ............................................ 33
       4.4.4 Regulate a Possibly Non Converging Pressure .................................................. 33

5 Results ................................................................. 34
   5.1 Repetition of tests and warmup times ................................ 34
   5.2 Performance of Massive File I/O ..................................... 34
   5.3 Performance of Variable Dump and Steering Interaction ...... 36
   5.4 Proof of Concept: Constraint Optimization with a Feedback Loop .................................................. 38

6 Discussion ................................................................. 39
   6.1 Performance of Massive File I/O ..................................... 39
   6.2 Performance of Variable Dump and Steering Interaction ...... 40
   6.3 Proof of Concept: Constraint Optimization with a Feedback Loop .................................................. 41
   6.4 Data assimilation in parFlowVR ........................................ 41
       6.4.1 Twin Experiments .................................................. 41
       6.4.2 Variational Data Assimilation and Kalman Filters .................................................. 43
       6.4.3 Data Assimilation Methods Propagating Multiple System States .................................................. 43

7 Conclusion and Future Work ........................................... 45
   7.1 Conclusion ................................................................. 45
   7.2 Future Work .............................................................. 46
List of Figures

1.1 The West Africa computational domain ........................................... 3
2.1 Schematic of the simulated domain and processes modelled by ParFlow-CLM .... 5
2.2 ParFlow domain decomposition model .............................................. 7
2.3 Traditional workflow of numerical simulation .................................. 8
2.4 Example process pinning to cores for an In Situ workflow ................. 9
2.5 Process pinning with helper cores and staging nodes ...................... 10
2.6 FlowVR run time environment .................................................. 13
3.1 An action message containing a steer message .............................. 16
3.2 Function of the MergeGridMessages filter .................................. 18
3.3 visit-connector ........................................................................ 20
3.4 A simple data flow which will be parallelized .............................. 20
3.5 Parallel execution flow due to multiple kick off messages ............ 20
4.1 Data flow used to test the performance of In Situ file I/O ............... 28
4.2 Simple loop data flow to measure ParFlow wall time if steered ....... 29
4.3 Data flow for algorithmic steering ............................................. 31
4.4 Sensibility of pressure to manning for controller calibration ........... 33
5.1 ParFlow wall times for different output methods on 2 nodes .......... 34
5.2 Wall time deviations when using the classic ParFlow output method ... 35
5.3 ParFlow wall times for different output methods .......................... 36
5.4 ParFlow wall times for output methods with node IO .................... 36
5.5 Steering performance test on a numerically easy domain ............... 36
5.6 Steering performance test on the West Africa domain .................. 37
5.7 Proof of concept: pressure regulation ........................................ 38
6.1 Data flow for a simple twin experiment ..................................... 42
6.2 Data assimilation in FlowVR .................................................. 44
1 Introduction

1.1 Motivation

Properly understanding the impact of global changes (i.e. the climate change, land-use and land-cover changes) on the water cycle of the critical zone is a major objective of modern hydrological research. The West Africa region is of special interest as having a population growth among the highest in the world [1, 43]. Further, West Africa is one of the most vulnerable regions to global change visible for instance by the increase of the intensity of extreme rain events over the Sahel [30, 35].

Simulations are a central tool to examine the impact of global changes. They could help predicting hydrological processes possibly leading to hydrological hazards and open perspectives in water management or even hydroelectricity. In earth sciences like oceanography and meteorology numerical models of water and air flow already give deep insights, not only because simulating the continental scale and larger is possible. In those fields large scale models are even used in operational systems e.g. for weather forecasts.

In contrast, large scale simulation is new in the field of hydrological surface and subsurface flows, but, nevertheless necessary to understand hydrological processes and how they are influenced by both: anthropic and natural forces. One of the reasons is that the heterogeneity of the hydrodynamic properties of soil controlling water transfer in the critical zone is much larger compared to e.g. the ocean’s or the atmospheric heterogeneity and much harder to measure intrusively due to the high density of soil compared to water and air. A lack of data and the high dimensionality of the system state hindered hydrological large scale simulations in the past.

In the era of big data, petascale computing [4] and satellite measurements, the lack of soil data can be tackled using data assimilation [44]. Thus the parameters of the soil are deduced from the observed phenomena. Accordingly calculation needs to be performed on unconventional large data sets, requiring new software infrastructure adapted for recent high-performance platforms to handle these tasks. In the following we will develop such an infrastructure especially facing the data challenge part.

Traditional numerical simulation on high-performance platforms relies strongly on file I/O as the classic workflow (Figure 2.3) uses the file system as interchange medium between the simulation and the analysis. Simulations are reading input data from the file system on initiation and then are producing an often even bigger amount of output data that is written back to the file system. After that during analysis the simulation’s output is read from the file system. Finally the analysis produces only a few output information which, again, is saved to the file system. Unfortunately file I/O is slow compared to the rate that processor cores are reading and producing data. Furthermore the file system is a shared resource, often between many cores and applications. This gives an additional slowdown when many cores try to

---

1 Defined on a vertical column from the deep impermeable bedrock to the top of the canopy, where water circulates. It is the critical zone for the development of humanity and the protection of the planet earth.

2 Leading high-performance computing platforms reach up to multiple PetaFLOPS (1 PetaFLOPS = $10^{15}$ floating point operations per second).
access it at the same time as they need to wait for each other [45].

Thus file I/O should be avoided as much as possible being especially important to scale for next generation’s exascale machines, which consist of many thousands of cores hypothetically blocking each other on file system access. Also inter node communications on such infrastructures should be minimized as more costly than intra node communication in both: energy and time.

As solution In Situ and In Transit workflows are introduced (subsection 2.2.1) and implemented backed by the FlowVR middleware (section 2.5). This enables human or algorithmic computational steering (subsection 2.2.2) and much faster file I/O (section 5.2). Accordingly we enable efficient and flexible coupling of the hydrological model ParFlow-CLM (section 2.1) with other numerical models and using it for data assimilation, and scientific data analysis on the continental domain scale.

1.2 Task

To let hydrological simulations benefit from recent machines and developments in high-performance computing, helping to allow large scale simulation in this field of studies, this master’s thesis focuses on developing the infrastructure

- to enable In Situ and In Transit workflows with the given hydrological model ParFlow-CLM tackling the data challenge in the current hydrological simulation domain,
- to enable computational steering and thus efficient assimilation of ParFlow-CLM in order to create a powerful tool that allows to constrain high-resolution water resources modeling over West Africa with both: measured data (data assimilation) and coupled 3rd party models.

1.3 The West Africa Domain

To simulate the West Africa domain (Figure 1.1), the ParFlow-CLM model will be executed on a domain of 2880 km * 1600 km reaching 10 grid cells (100m) into the ground with a resolution of 1 km on the x- and y-axis. Dumping every time step (30 min) the uncompressed pressure state in double precision of every grid cell leads to ≈350 MB of data as output per time step. The infrastructure developed in the following is capable to highly parallelized handle this amount of data avoiding the file I/O bottleneck. Further it will allow to produce higher-level outputs as e.g. dumping hydrological data of particular domain regions just after rain events in a high temporal resolution without changing the model code of ParFlow-CLM. This evades the fact that it is not possible to store the full simulation output in high temporal and spacial resolution due to the overwhelming amount of data it would produce.

1.4 Related Work

High-resolution simulation (≈1 km² resolution in the x-y-layer) on the continental scale (multiple million grid cells) with ParFlow were already performed by Laura Condon and Reed Maxwell on the continental
Figure 1.1: The computational domain that shall be handled by the software infrastructure developed in this work. The graphic shows the pressure distribution after a parking lot run: The top soil layer is set impermeable and after some induced rain rivers and lakes appear.

U.S. and by Keune et al. for modeling Europe [12, 18]. In particular the latter used ParFlow in a coupled soil-vegetation-atmosphere model with OASIS3-MCT as coupling framework [38] resulting in the Terrestrial System Modelling Platform (TerrSysMP).

Furthermore data assimilation for the land surface–subsurface part to integrate according measurements in terrestrial system modeling using ParFlow was already performed by Kurtz et al. using the PDAF framework, again with TerrSysMP [21, 28].

The FlowVR middleware as In Situ and In Transit framework was already used in environments of numerical simulation before, notably the GROMACS molecular dynamics simulator [7], especially regarding performant file output [14].

In this work we combine the ParFlow model with the FlowVR middleware as an alternative to the OASIS3-MCT coupler for simulation systems containing the ParFlow model creating the parFlowVR framework. This allows advanced control on coupling exchanges, In Situ and In Transit analysis and output, flexible adjustment of the couplings and application layout as well as rapid prototyping of steering forcings, other models and assimilation algorithms due to a Python API. An other advantage is that FlowVR abstracts couplings in a much more appropriate and easier readable manner: data flow graphs, where the nodes represent tasks being processed on the data and the links describe the actual data flow in between. This allows the description of distributed applications consisting of multiple data tasks such as running different models, analysis, data compression and data output to the file system even for large scale computing on multiple thousands of computing cores [15].
1.5 Supplement

The code developed for this work and the used version of the FlowVR middleware and ParFlow (including the parFlowVR framework) are accessible online:

- https://gitlab.cs.fau.de/friese/MasterSupplement

The scripts to run the experiments, the resulting raw data, and the scripts to generate charts from the raw data can be found in the supplement too.

1.6 Acknowledgments

Some of the computations presented in this paper were performed using the Froggy platform of the CIMENT infrastructure (https://ciment.ujf-grenoble.fr), which is supported by the Rhône-Alpes region (GRANT CPER07_13 CIRA), the OSUG@2020 labex (reference ANR10 LABX56) and the Equip@Meso project (reference ANR-10-EQPX-29-01) of the programme Investissements d’Avenir supervised by the Agence Nationale pour la Recherche.

I would like to thank Jean-Martial Cohard and Thierry Pellarin from the IGE Grenoble as well as Ulrich Rüde and Dominik Thönnes from the LSS of the Friedrich-Alexander University Erlangen-Nürnberg for making this master’s thesis possible and supervising me. Furthermore I would like to offer my special thanks to Bruno Raffin from the DATAMOVE team in Grenoble as well as to Basile Hector, Alban Depeyre and François Gibon from the IGE Grenoble for their valuable feedback and their time answering my questions on hydrology and high-performance computing.
2 Material and Methods

In this section we will present the ParFlow hydrological model which is used in this work. Further we introduce In Situ and In Transit workflows and how one can benefit from these techniques to steer numerical models. This will show the need of a middleware abstracting such work and data flows. We will propose FlowVR as solution. Finally we explain the concepts of data assimilation and measuring computation time as both are needed to understand the following chapters.

2.1 ParFlow

2.1.1 The ParFlow hydrological model

The ParFlow-CLM\(^1\) hydrological model \([9, 17, 19, 20, 23, 24]\) is a commonly used software in the hydrology community. It is a state-of-the-art, physically based, fully coupled critical zone model performing surface and subsurface flow simulation on high-performance computers. A hydrological domain as typically

\(^1\)CLM is a model integrated into ParFlow that can be activated to simulate the land surface interface. We describe CLM in greater detail in subsection 2.1.3.
simulated with ParFlow is shown in Figure 2.1. ParFlow is freely available as open source on GitHub [6] licensed under the GNU General Public License 2.1.

Physically the subsurface flow of water in the porous soil is described by the Richards’ equation. ParFlow uses the following formulation [25]:

\[
S(p)S_s \frac{\partial p}{\partial t} - \frac{\partial (S(p)\rho(p)\phi)}{\partial t} - \nabla \cdot (K(p)\rho(p)(\nabla p - \rho(p)\vec{g})) = Q, \text{ in } \Omega
\]  

(2.1)

The variable descriptions of Equation 2.1 can be found in Table 2.1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Omega)</td>
<td>the flow domain</td>
<td></td>
</tr>
<tr>
<td>(p)</td>
<td>the pressure-head of water</td>
<td>(L)</td>
</tr>
<tr>
<td>(S)</td>
<td>the water saturation</td>
<td></td>
</tr>
<tr>
<td>(S_s)</td>
<td>the specific storage coefficient</td>
<td>(L^{-1})</td>
</tr>
<tr>
<td>(\phi)</td>
<td>the porosity of the medium</td>
<td></td>
</tr>
<tr>
<td>(K(p))</td>
<td>the hydraulic conductivity tensor</td>
<td>(LT^{-1})</td>
</tr>
<tr>
<td>(Q)</td>
<td>the water source/sink term (includes wells and surface fluxes)</td>
<td>(L^3T^{-1})</td>
</tr>
</tbody>
</table>

Table 2.1: Variable descriptions of the Richards’ equation (Equation 2.1). \(L\) denotes length units, \(T\) denotes time units.

For the overland flow the shallow water equations are used as boundary condition for the subsurface flows [25].

2.1.2 Technical Details

For solving the partial differential equations (e.g. Equation 2.1) a multigrid-preconditioned conjugate gradient solver [9] and for non-linearities a Newton-Krylov nonlinear solver [3, 17] are used. Furthermore the Hypre package [2] is included for improved solver flexibility and greater preconditioning.

ParFlow works on Cartesian spatial coordinates. To parallelize execution the domain is decomposed into uniform cuboid parts as seen in Figure 2.2. As there is much communication needed in \(z\) direction, normally no domain decomposition along the \(z\) axis is performed. The simulated domain is spatially separated into cuboid grid cells. The \(x\)- and \(y\) spacings are constant. In \(z\) direction a variable grid spacing can be defined. To model topographic slopes the gradients \(\frac{\Delta z}{\Delta x}\) and \(\frac{\Delta z}{\Delta y}\) can be given for each column \((x, y)\) [25]. This is illustrated in Figure 2.1.

2.1.3 Use Cases and Integration with other Models

Equation 2.1 and the shallow water equations are holding true for very regional problems as well as for continental scale up to global scale problems. Thus ParFlow was already successfully used for problems of the hill slope scale up to problems of the continental scale for whole Europe or the continental United States of America [12, 21].
ParFlow uses a simple domain decomposition method: every ParFlow process holds a cuboid part of the simulated domain. As much more data exchange in $z$- than in $x$ or $y$ direction is needed for the calculations the grid is typically divided in the $x$-$y$-plane.

There are also use cases where ParFlow was coupled with other large scale environmental models already. In the TerrSysMP project for instance, ParFlow is coupled with the Community Land Model and the atmospheric model COSMO-DE [21]. The Community Land Model (CLM) [29] describes the land surface interface. It is a part of the official ParFlow distribution [?].

COSMO-DE is the atmospheric model mainly used by the German weather forecast.

TerrSysMP uses OASIS-MCT [38] to couple those models and execute them side by side on the same high-performance computing platform.

PDAF, the Parallel Data Assimilation Framework [28] is used to efficiently assimilate the resulting numerical model.

The approach proposed in the following will simplify the coupling to other models. The coupled models must be ported as parFlowVR modules (subsection 2.5.2). After this, one can vary couplings without the need to change the code of the coupled models or the transmission layer between the different models.

2.2 In Situ, In Transit, Helper Cores and Staging Nodes

2.2.1 In Situ and In Transit Data Processing

Traditional numerical simulation workflows depend strongly on file I/O as seen in Figure 2.3. Tasks are executed offline, one after another, each of them writing and reading intermediate files on the hard drive. There the file I/O often is a bottleneck since modern (high-performance) computers typically consist of many processors, which in contrast are sharing one (parallel) file system. In Situ tackles this problem. With In Situ we denote workflows avoiding intermediate file writes between simulation, analysis, visualization and data compression tasks [10]. In Situ performs those tasks online, while the data is still in the simulation’s memory (see Figure 2.4). Depending on the application even memory copies can be avoided with the help of shared memory (zero copy).

To achieve even better performances In Situ performs tasks asynchronously: while for instance In Situ analysis is executed on only a part of the computational resources, the simulation does not wait for it to
finish but calculates already the next cycle. For that reason so called helper cores are introduced. Those are cores dedicated to merging data from the simulation cores, compression, analysis or visualization. To minimize the data transfer through the network between helper cores and the simulation cores, the helper cores are on the same nodes as the actual model. Depending on the work that is laid off to the helper cores, one or more helper cores per participating node are used, processing the data of the simulation cores on the same node. This has advantages regarding nowadays high-performance computing environments: In Situ avoids the limiting file I/O bottleneck and makes efficient use of the multi processor structure of those systems as the simulation and analysis, visualization, file output or data compression run in parallel. A demonstration of In Situ using helper cores and its performance gain can be found in [13].

We will refer to In Situ if In Situ tasks as e.g. analysis or visualization are executed on the same node as the simulation. Even transferring data to other nodes instead of writing it to the hard drive brings performance improvements. Accordingly we will refer to In Transit if the In Situ task is performed on another (dedicated) node without intermediate file I/O. This is illustrated in Figure 2.5: helper cores operate In Situ on each node. Further data processing then is done In Transit on a dedicated node.

To move from traditional workflows to In Situ and In Transit workflows, additional infrastructure replacing the file system as exchange layer and abstracting asynchronism is needed. The FlowVR middleware (section 2.5) is used to provide such infrastructure in the following.

Once such a middleware is successfully implemented the challenge is to already know before running the simulation which kind of analysis shall be performed on the simulation output. Only with this knowledge the desired analysis can be put into data flow components being executed In Situ, parallel to the simulation itself, working on memory segments shared with the simulation to avoid file I/O and possibly memory copies.

A major task when working with simulations is exploring which analysis leads to which results using the raw simulation output. In this case analysis cannot be executed In Situ. Still one can benefit from In Situ workflows as they allow file output and compression in the background effectively leading to shorter run times of the simulation [13].

Furthermore numerical models allow to play with their parameters manually: Traditionally the simulation is executed, and result files are inspected. Curiosity and the will to get a rough feeling for the systems parameter response lead to input changes followed by further simulation runs. In Situ allows to change the simulation state and parameters of a running simulation and also inspecting simulation output online, making file system interactions and model reinitialization obsolete allowing much smoother workflows.
2.2.2 Computational Steering

In Situ and In Transit analysis and visualization of simulation data allows to draw conclusions of the simulation during run time. Conclusions of simulating one time step provide parameter changes for the next simulation time step.

Thus changing simulation parameters during run time with the knowledge retrieved by In Situ or In Transit analysis and visualization is the logical next step. Introducing changes in a running simulation without the need of restarting it as illustrated in Figure 2.4 is called (computational) steering [40, 41]. Computational steering introduces a steering forcing into the simulation. This forcing can either be human generated (human in the loop) or calculated by an algorithm (algorithmic steering). Furthermore the steering forcing can be introduced by a coupling with another model.

Computational steering avoids the generation of multiple input files and the reinitialization of the simulation for different parameter sets. To guarantee the traceability of the steered simulation every steering forcing introduced must be appropriately logged. Computational steering can be used to do data assimilation.

2.2.3 Helper Cores and Staging Nodes

Sometimes output to the file system is nevertheless necessary. For instance backing up the current state of a simulation can be such a case. The In Situ or In Transit way to do so is via helper cores and staging nodes dedicated to merging and routing data to the file system. The simulation’s computing units are
not dumping the data directly to the disk. They dump the data to the dedicated helper cores or staging nodes, which then perform the actual file write. Figure 2.5 illustrates this.

Thus the computing units are not slowed down when waiting for the file system. Often the staging cores or nodes even perform some preliminary analysis or compression on the data before writing to minimize file output.

Depending on the ratio between computation and file I/O this can lead to a higher overall performance and shorter run times on the same resources [13].

### 2.3 Data Assimilation

#### 2.3.1 Variational and statistical Data Assimilation

Data assimilation tries to improve computational models by comparing the model’s results with observations [8]. Models need to be assimilated periodically to allow the most precise analysis or prediction. Thus data assimilation became a typical problem in computational modeling.

A data assimilation problem is described by the following formalism:

$$x_{k+1} = M x_k$$  \hspace{1cm} (2.2)

$$y_k = H x_k$$  \hspace{1cm} (2.3)

$$x_{opt} = \arg\min_{x_k \in \mathbb{R}^n} error(y_k)$$  \hspace{1cm} (2.4)

where $x_k$ is the system’s state vector in the $k$th iteration as calculated by the model operator $M$ applied on the preceding state vector $x_{k-1}$. $y_k$ is the according observation vector, containing the information used to interpret the model results. It is calculated by applying the $H$ operator on the state vector $x$. The error between the observation and the measured value, $error(y_k)$, is calculated and minimized, leading to an optimal input state $x_{opt}$ of the model. Thus assimilation problems are inverse optimization problems [8].
Model parameters (e.g., manning, permeability... in ParFlow) can be assimilated too. For this one must include them in the state vector. Parameters thus can be seen as entries of state vectors which the model operator $M$ will keep unchanged but which still impact how $M$ affects the other state vector’s entries. To find optimal parameter values, the assimilation should use measurements from multiple different times getting assimilated by different model time steps. Thus the techniques for state assimilation shown in the following are usable to assimilate parameters too.

Finding such a single optimal state $x_{\text{opt}}$ is called variational data assimilation. Unfortunately measurements as those used to compare the simulation with real measurements in $\text{error}(\cdot)$ are always error prone themselves. Thus the assimilated state $x_{\text{opt}}$ contains a deviation from the real state. With variational data assimilation we lose the information on the correctness of the retrieved optimal state $x_{\text{opt}}$.

To keep track of the uncertainty and thus of the probability density function of state $x_{\text{opt}}$, one needs to propagate the measurement error through the model $M$. When assimilating to measurements (calculating $x_{\text{opt}}$) the probability density function is maximized taking into account the model’s prediction on the state and the discrepancy between observed measurement and predicted observation. This is called statistical data assimilation. The most common methods to do so are Kalman filters, ensemble Kalman filters and particle filters.

Kalman filters are used when the operators ($M$, $H$) are linear\(^2\) and the measurement error can be described by a normal distribution. In this case we can describe the system state $x$ and its probability respecting the measurement errors with a multidimensional normal distribution’s covariance and its mean $x_{\text{opt}}$.

If measurement errors are normal distributed but the model has nonlinearities, ensemble Kalman filters are used. Here an ensemble of states describes the probability density function.

A more general way to propagate probability information through model iterations are particle filters. They can handle nonlinear models with arbitrary probability density functions. The drawback is that a big ensemble of weighted states (particles) needs to be handled, representing the state’s probability density function.

Concerning the data flow in these assimilation methods we can distinguish two groups:

1. Variational data assimilation and Kalman filters work on a single system state. (We include the parameters defining the probability density function in the Kalman filter in this state.) Thus a single simulation is used to propagate a single system state only.

2. Ensemble Kalman filters and particle filters propagate multiple states through the numerical model, technically different simulations on different system states are executed.

We will show in section 6.4 how to integrate these data assimilation methods with ParFlow benefiting from In Situ/In Transit data flows. The ParFlow simulation software poses the model operator $M$. Since the entries of the ParFlow state vector are the same values that are typically measured (e.g., pressure, manning...), the observation operator $H$ is the identity:

$$H = I \quad (2.5)$$

\(^2\)Roughly speaking: Small changes in the input lead to small changes in the model output.
2.3.2 Twin Experiments

Twin experiments are a tool to evaluate and to improve data assimilation. The model itself is used to define a reference trajectory \( y_{\text{ref}} \) of the background data or (synthetic) measurements by fixing the models initial conditions to the parameter set \( \theta \). To become more realistic some noise can be added to \( y_{\text{ref}} \). Now data assimilation using the same model is started but with an initial guess \( \theta_0 \) as start parameter set different from the reference: \( \theta_0 \neq \theta \). The goal is to assimilate on \( y_{\text{ref}} \). This is a basic and nevertheless important way to evaluate the performance of the used data assimilation system.

2.4 Computation Time Measures

There are two ways to measure the execution time of a program \( x \):

1. The wall time of program \( x \) is the time that one needs to wait until the execution of program \( x \) has finished. Thus it is also denoted as elapsed real time.

2. In contrast the CPU time of program \( x \) denotes the time the core(s) have spent on the given program \( x \).

Thus for programs running on multiple cores the CPU time gets easily larger than the wall time whereas for programs on single core systems the CPU time is typically smaller than the wall time as other problems are scheduled in between program start and end.

In the following we are inspecting wall times only, as those give a direct estimate of how long computations take. More precisely the wall time of ParFlow per time step is measured. To get an overall estimate on the wall time of a simulation this is the main part but some highly varying wall time parts for initialization and finalization have to be added. Still, when simulating over a long time span the impact of initialization and finalization to the wall time becomes more and more negligible and thus is ignored in our measures.

2.5 FlowVR

2.5.1 Introduction

FlowVR [14] is a middleware allowing flexible design of In Situ and In Transit workflows in high-performance computing.

In FlowVR the user defines a data flow graph as shown later in for example Figure 3.3 and Figure 4.2 for a distributed FlowVR application by a Python script. This graph assembles different FlowVR components, which do data processing such as simulation, file I/O, analysis, steering and routing. On run time the specified data flow graph is then instantiated on the given infrastructure. Thus new pinning of components to cores and different data flows can be explored without recompiling the FlowVR components. All this made FlowVR a good choice with many use cases for this work.

FlowVR components communicate by exchanging messages with each other. FlowVR takes care of message transmission between the different components no matter on which computing node or core
they were launched. To achieve this the middleware runs a daemon process on each node of the high-performance computing infrastructure. The intra node communication is performed through shared memory. Communication to components on remote nodes is performed through the daemons: First the component transmits the message through the shared memory to the local daemon process, which then routes it by means of MPI communications [26] or TCP/IP (depending on the daemon configuration) to the daemon process on the receiving node. The daemon process on the receiving nodes passes the message through the shared memory to the receiving component. Figure 2.6 illustrates this process.

![FlowVR process diagram](image)

**Figure 2.6:** FlowVR launches a daemon process on every participating node. Each daemon process then launches FlowVR components on its node, taking care of the intra node communication (shared memory) and the inter node communication (TCP or MPI network capabilities). Taken from [16].

### 2.5.2 Components

Components in FlowVR are performing program logic by processing messages. They have input ports to receive messages from other components and output ports to send messages to other components. There are two types of FlowVR components: Filters and Modules.

*Filters* are used for message routing purposes. They are instantiated as FlowVR daemon plugins.

*Modules* are performing tasks on data. They are started on their own using their usual launching command like codempirun. Modules communicate through shared memory with the local FlowVR daemon. Modules follow a simple messaging paradigm: They `wait()` for incoming messages, read them (`get()`) and react by `put()-ing` messages to their output ports. One such cycle is denoted *(module-)iteration*. Every module has two standard ports: The input port `beginIt`, which is used to connect the iteration start with a condition and the `endIt` output port, which sends a message at the end of each module iteration just before the next `wait()` is performed.

To use the FlowVR middleware for a scientific workflow its components must be represented by FlowVR components. Thus a coding effort has to be done. FlowVR has module APIs in C++, C and Python to create FlowVR modules from e.g. existing simulation codes.
2.5.3 Rules

FlowVR defines some rules for the transmitted messages that need to be known to develop scientific data flows:

1. A module starts its next cycle if a message was received on every connected input port.  
2. Each output port can send to more than one input ports.  
3. Each input port can only receive messages from one output port.

2.5.4 Performance

Communication

As for optimal communication between FlowVR components on one node shared memory is used, one can even let components work directly in the message's shared memory segment so that zero copy communication is possible. For inter node messages TCP/IP and MPI communications between the daemons are implemented by the middleware. Choosing MPI communications enables to benefit from high-performance network standards like InfiniBand when available on the used machine.

It could be shown for In Situ frameworks that the daemon process, even if taking away some computing power, often still has a positive impact on performance. This is the case as the daemon process allows optimized data flows reducing the I/O bottleneck. [13][14]

In this work two ways of doing so are considered:

- In Situ and In Transit analysis to reduce file I/O
- In Situ and In Transit file output to reduce race conditions on the file system

Trees

To parallelize data broadcasts and gatherings tree generation is supported by the FlowVR middleware. For instance instead of broadcasting (gathering) one message to (from) \( x \) components, the middleware provides functionality to generate trees of variable arity \( n \).

This leads to \( \log_n x \) chained executions of broadcasts (gathers) to (from) \( n \) components. Normally those can be parallelized resulting in run times

\[
t_{\text{broadcast/gather}}(x) = O\left( \log_n x \right)
\]

(2.6)

Such a gathering tree of arity \( n = 2 \) merging data from multiple ParFlow processes is for instance shown in Figure 4.2.

---

\( ^3 \)This is not holding true for filters and modules with asynchronous input ports. The latter is not used in this work as it breaks the conceptual difference between filters and modules.
3 Implementation

This chapter presents how the parFlowVR framework was implemented to combine ParFlow with FlowVR. We will present the semantic of the implemented FlowVR components, justify design decisions and show how algorithmic steering can be achieved using the created Analyzer API. This knowledge will allow the reader to understand and modify the data flows of the distributed applications shown in the later chapters.

The developed parFlowVR framework and its dependency FlowVR can be found in the supplement to this work (section 1.5).

3.1 The parFlowVR Framework

3.1.1 Framework

In the following the transformation of the ParFlow workflow into a workflow using FlowVR is discussed. Thus a framework was implemented that we will refer to as parFlowVR. parFlowVR consists of FlowVR components, which are assembled to express the desired data flow solving a particular hydrological problem. The components communicate by transmitting messages between each other. Components representing analysis tasks can be easily created with parFlowVR.

Modules of the parFlowVR framework are written in C as ParFlow is too. This ensures that code (e.g. for message type declarations and message parsing) can be used from all modules and in the ParFlow code.

3.1.2 Messaging Protocol

The components in the parFlowVR data flows, representing distributed applications, communicate by exchanging messages through the FlowVR layer. Such FlowVR messages have an attached shared memory segment that is used to transport data. Depending on the message type this shared memory segment contains different data structures except for empty messages (which actually have no data segment attached). These memory segments used by parFlowVR start with a header data structure (as declared in Listing 3.1). It can be followed by the header data structure of the next message in a concatenation of multiple messages (as produced by some filters) or by an array representing raw grid data. This creates messages of minimal size, which can still be easily parsed as the header states how many bytes of grid data must be skipped to reach the next header (when parsing a concatenated message) or the end of the message. Further this facilitates filtering inspecting the header only.

\[1\] In fact, while implementing parFlowVR, ParFlow becomes packed into such a module too.
typedef struct {
    Variable variable;
    Action action;
} ActionMessageMetadata;

typedef struct {
    int nx, ny, nz, ix, iy, iz;
} SteerMessageMetadata;

typedef struct {
    int nX, nY, nZ;
} GridDefinition;

typedef struct {
    double time;
    Variable variable;
    GridDefinition grid;
    int nx, ny, nz, ix, iy, iz;
    char run_name[128];
} GridMessageMetadata;

Listing 3.1: Message metadata declarations used for message headers. nx, ny, nz define how many grid cells are transmitted per dimension. ix, iy, iz define where to put the transmitted cells in the simulation’s grid. nX, nY, nZ define the grid size for the given variable in the simulation. run_name defines from which ParFlow simulation the grid message is sent.

In more detail the following message types are used by parFlowVR:

**Action Messages**

Action messages are messages either requesting a steering (Figure 3.1) or snapshots to be sent on the snapshot port. Thus action messages contain a header declared as ActionMessageMetaData (Listing 3.1). The header defines which action to apply on which variable. If the action is a steer action like add, multiply or set a steer message is attached.

**Steer Messages**

Steer messages transmit a grid that will act as operand of a steer action. They are built up from a SteerMessageMetaData (Listing 3.1) object describing on which part of the variable’s grid to apply the operand. Behind the SteerMessageMetaData the raw operand values are appended to the message.

![Figure 3.1: An action message containing a steer message.](image-url)
Grid Messages

Grid messages are generated by the ParFlow simulation if a snapshot is requested or if a variable needs to be dumped to an output port. The header is a GridMessageMetadata object (Listing 3.1). It defines the current timestamp, the transmitted variable, the overall grid dimension for the transmitted variable and the transmitted grid fragment’s position. It is followed by the raw grid data of the transmitted grid fragment.

Empty Messages

Furthermore to kick of actions in some places, empty messages acting as tokens are sent. They are also used by the parflow module to signal that it finished all the time steps to be processed.

3.1.3 Components

Used Filters

For this work some predefined FlowVR filters are used. They are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Filter Name</th>
<th>Description</th>
<th>Used to</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreSignal</td>
<td>A filter sending one or more empty kick off messages if every component is ready.</td>
<td>• release one or more execution tokens in the data flow graph to allow parallelism</td>
</tr>
<tr>
<td>Merge</td>
<td>A filter that sends a concatenation of the first message per input port as soon as there is at least one message at each input port. The amount of input ports is variable.</td>
<td>• merge sub grids of one time step to prepare file write • merge distributed steer messages that need to be handled at the same time</td>
</tr>
</tbody>
</table>

Table 3.1: Overview of used FlowVR filters.

Furthermore the filters MergeItExt and MergeGridMessages are implemented to facilitate the abstraction of asynchronism and message conversion (concatenations of grid messages to one grid message) in the data flow layer. In the following, they are explained in more detail:

MergeItExt is an extension of the MergeIt filter. As input it has a variable amount of in ports and an order port. Always if there is a message on the order port, a message is sent out that is either a concatenation of all the messages waiting on the in ports or an empty message if no message is waiting on the in ports. It is used to synchronize asynchronously incoming steering and snapshot requests to the simulation’s cycle.

If ParFlow is executed on multiple cores, each core runs a parflow module as a process. Each of those modules are ParFlow processes sending grid messages containing only the part of a variable they hold in memory. Those messages then are typically merged with trees of Merge filters. Such a merge tree can be seen in Figure 3.2. To convert the resulting concatenated messages resulting from Merge filters into a single grid message holding the whole variable, the MergeGridMessages filter is used. It has one
input port accepting concatenated grid messages. Its outport responds with the converted overall grid message. An example use is shown in Figure 3.2 and Figure 4.2.

**Figure 3.2:** Each ParFlow module works on a part of the decomposed grid (top). To get the full grid in parFlowVR a tree of Merge filters is used to concatenate all grid messages. Each grid message contains a header and a grid part. A concatenation of multiple grid messages containing different segments of a variable’s grid is recombined to an overall grid message (bottom) by a MergeGridMessages filter.

**Used Modules**

The FlowVR modules listed in Table 3.2 were implemented to represent the ParFlow simulation framework by a FlowVR data flow.
Table 3.2: Overview of used FlowVR modules.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Description</th>
<th>Consumes</th>
<th>Produces</th>
</tr>
</thead>
<tbody>
<tr>
<td>parflow (section 3.2)</td>
<td>This is the module performing ParFlow simulations. At the beginning of each ParFlow cycle all waiting action messages (section 3.1.2) are processed. This module sends out grid messages as specified in the ParFlow input database. Therefore a simple contract system (subsection 3.2.1) is used.</td>
<td>action messages</td>
<td>grid messages</td>
</tr>
<tr>
<td>netcdf-writer (subsection 3.3.1)</td>
<td>A module dumping incoming data to netCDF files. If more than one variable shall be written into the same file a merge must be defined in the FlowVR data flow. Multiple processes of this can be executed connected via MPI.</td>
<td>grid messages</td>
<td></td>
</tr>
<tr>
<td>C-analyzer, Python-analyzer (section 3.4)</td>
<td>Example modules showing how to write performant analyzer modules in C respectively Python that react on grid messages from ParFlow by introducing steering forcings in the ParFlow simulation.</td>
<td>grid messages</td>
<td>steer action messages</td>
</tr>
<tr>
<td>visit-connector (subsection 3.3.2)</td>
<td>This module is used to inspect the state of ongoing ParFlow simulations with VisIt at any given time. Thus no output files need to be created for this purpose.</td>
<td>grid messages</td>
<td>action messages</td>
</tr>
</tbody>
</table>

### 3.1.4 Synchronization

FlowVR was originally designed completely synchronous, meaning that every FlowVR module needs a message on each connected input port to start its work. As this is not fitting for especially user interaction, asynchronous ports were added by the FlowVR developers and are now part of the standard FlowVR distribution [16]. In the case of parFlowVR all input ports of all modules are synchronous. The parFlowVR modules start message processing only if there is a message on each connected input port. Nevertheless the user interaction of the VisIt connector must be introduced asynchronously. Sometimes it is useful to asynchronously introduce Steering forcings too. In these cases the MergItExt filter (section 3.1.3) is used to abstract this from the parFlowVR modules. An example setup can be seen in Figure 3.3.

### 3.1.5 Kick off Messages from PreSignals

Since FlowVR modules start their work only if a message is available on each connected input port (see subsection 2.5.3), a closed loop data flow would be never started. For this PreSignal filters are used as they insert a fixed number $n$ of empty messages when the FlowVR application is ready to start. Later they just forward incoming messages. The number $n$ also defines the "grade of asynchronism": We assume a FlowVR application with 3 components, assembled as in Figure 3.4 with the PreSignal releasing $n$ kick off messages at application start. Then module A will be executed $n$ times before taking into account results from module B for the first time.

In the most cases we set $n \geq 2$ since this leads to A and B being executed in parallel as illustrated in Figure 3.5. The drawback is that A will get messages from B that are outdated by $n - 1$ iterations.
Figure 3.3: VisIt requests data from the parflow module on user interaction. This is asynchronous. To synchronize it, the MergeItExt filter is used. MergeItExt always sends a message when getting polled by the parflow module. Sometimes the message sent is empty if there was no user interaction.

Figure 3.4: A simple loop data flow. The “grade of asynchronism” can be defined by the number \( n \) of kick off messages released by the PreSignal component.

Figure 3.5: When moving from one to two kick off messages we introduce parallel execution. This shortens the wall time assuming we have enough free cores. The data flow is as in Figure 3.4. Horizontal black arrows show messages exchanged between module executions. Dashed red arrows are kickoff messages.
3.2 The parflow Module

3.2.1 A Contract System for Data Filtering

To reduce and cleanly handle data output sent through the FlowVR layer, we wanted to use automatic data filtering by a contract system as proposed in [27]. This minimizes the data communicated through the network while still being portable.

Thus normally an extra data operator is implemented which performs routing of data sent from producers to consumers. For this purpose the data operator holds a list of contracts defining which consumer needs and which producer can deliver which kind of data at a particular frequency. In our case the parflow module acts as producer and the different analyzer, visualization and writer modules (e.g., C-analyzer, Python-analyzer, visit-connector, netcdf-writer) act as consumers. The parflow module consumes action messages produced by the analyzer modules or the visualization. We will not consider the latter in our contract system as these action messages can be produced asynchronous, making it hard to depict them in a contract defining a frequency of production / consumption. So the parflow module is the only producer with all the other modules seen as consumers in our contract system. Furthermore the parflow module produces all kind of simulation output data with the highest possible frequency (one simulation step). Thus the tasks of the data operator become trivial. Accordingly the data operator is coded within the parflow module. This removes the messaging layer between producer and data operator resulting in a performance boost.

The parflow module produces the variables listed in Listing 3.2 once each DumpInterval. The variables correspond to the parameters from the hydrological model (subsection 2.1.1). The DumpInterval is specified in the ParFlow input database file. ParFlow then ensures to have valid values for all the parameters every DumpInterval time units.

Contracts specify which variable to send with a specific periodicity and offset on which output port. Thus a contract is fulfilled and the according variable is dumped if the following two conditions hold true:

\[
\text{timestep} \geq \text{offset} \\
(timestep - \text{offset}) \mod \text{periodicity} = 0
\]

Every contract creates its own output port. If there is a consumer that needs more than one variable maybe even at different periodicities and offsets this can be realized with filters in the FlowVR layer.

The contracts themselves are specified in the ParFlow input database file generated by the configuration Tcl-script (see Listing 3.3).
pfset FlowVR.Outports.out1.Periodicity 10
pfset FlowVR.Outports.out1.Variable "permeability_y"
pfset FlowVR.Outports.out1.Offset 5

Listing 3.3: Contract configuration. Specifying output ports of the parflow module: on the port out1 the variable permeability_y will be sent every 10 dump intervals (Periodicity) starting after 5 dump intervals (Offset). (ParFlow guarantees to have a recent output at least every dump interval. Thus dump interval is used as unit.)

3.2.2 ParFlow-ParFlow Communication

As written in section 2.1 ParFlow already supports execution on supercomputers and in parallel. The Message Passing Interface (MPI, [26]) is used for communication between multiple ParFlow processes.

This communication was kept untouched when moving ParFlow into an FlowVR module: Launched ParFlow cores, even if launched as FlowVR modules, communicate in their own MPI layer with each other. There are two reasons for this:

1. ParFlow can still be executed without FlowVR
2. There is one communication layer less for the ParFlow-ParFlow communication compared to an approach where this communication would be routed through FlowVR. This naturally gives a better performance.

For communication between ParFlow and other FlowVR components the FlowVR layer is still traversed.

As a result ParFlow processes connected via MPI are synchronized: They read messages on the in input port at the same time and wait for each other when finishing their timestep. This is used when reacting to the end of a ParFlow timestep: in such cases listening to the messages sent from the endIt output port from one ParFlow process is sufficient.

3.2.3 Parallel Data Dumps

Every time the condition in Equation 3.1 holds true, the parflow module sends a grid message (section 3.1.2) containing the specified variable to an output port.

If more than one parflow modules were started, each of them sends such a grid message. Each ParFlow process holds a different part of the grid specifying the variables (see Figure 2.2).

Typically the different grid messages are merged with a Merge filter in the FlowVR layer. This means the grid messages of all parflow modules are concatenated and have to be read out sequentially later. Alternatively the concatenated message can be converted into a single overall grid message with the MergeGridMessages filter as seen in Figure 3.2.
3.2.4 Control flow options

As the parflow module is the main module in parFlowVR (every parFlowVR workflow will probably have at least one in it), it can be configured to control the overall parFlowVR application. Options are:

- Stop the parFlowVR application immediately when the problem given to a parflow module ended (pfset FlowVR.OnEnd Abort)
- Serve the final state for interaction with visualization even after the full problem was solved (pfset FlowVR.OnEnd ServeFinalState)
- Send an empty message to all by contracts defined output ports to broadcast that the ParFlow simulation has ended (pfset FlowVR.OnEnd SendEmpty)

The last option allows to ensure that all components of the parFlowVR workflow are shut down properly after all remaining messages were processed.

3.3 Output Modules

3.3.1 netcdf-writer

The netcdf-writer has an input port accepting grid messages with a stamp containing the filename. It works comparable to the netCDF write module in the standard ParFlow distribution [?] meaning it produces the exact same output files when properly configured. It is backed by the same netCDF library [5] as ParFlow.

It is used to realize In Situ and In Transit file writing on helper cores or staging nodes as well as for testing purposes. For instance comparing files written out from the ParFlow integrated netCDF output module and the parFlowVR netcdf-writer shows the integrity of data transferred through the FlowVR layer.

3.3.2 visit-connector

The visit-connector uses libsimV2 provided by VisIt [11, 42] to cooperate with the VisIt visualizer software.

VisIt sends through the visit-connector module action messages to parflow modules. Those action messages request meta data or simulation snapshots from ParFlow. When the parflow module processes such a message, the requested data is sent out to the snap port, which must be connected with the in port of the visit-connector.

The user can chose which data of the ongoing ParFlow simulation shall be shown in VisIt. Figure 3.3 illustrates how to connect a visit-connector module with the parflow module. Since the user interaction with VisIt is asynchronous, the visit-connector is connected via the MergeItExt filter with ParFlow.
3.4 The analyzer API

3.4.1 Analyzer

We denote FlowVR modules capable of doing intelligent, automated online steering of ParFlow simulations as analyzer. To write such modules the analyzer APIs were implemented.

Analyzer modules typically access data dumps from ParFlow and calculate a steering forcing from this. The steering forcing is introduced in the on going simulation by transmission of steer messages (section 3.1.2). They are written in C or Python programming language.

A typical analyzer works as follows:

1. It registers as a FlowVR module.
2. It waits for data from the parflow module.
3. The received data is parsed and analyzed. In the same time a steering forcing is calculated.
4. A steer message (section 3.1.2) representing the calculated steering forcing is created and sent to the parflow module. Go to step 2.

3.4.2 C-analyzer API

The native language to write FlowVR modules is C++. Nevertheless there is no C++ analyzer API: ParFlow and thus all the messaging protocol as shown in subsection 3.1.2 are implemented in C. So implementing the analyzer API in C was straight forward too. The C-analyzer API consists of methods to interact with parflow modules and the appropriate data types organized in a library. A simple C example file implementing the algorithm shown above (subsection 3.4.1) is given as example. It provides the structure to follow when writing a new FlowVR module that communicates with the parflow module through the FlowVR layer.

The C-analyzer API enables the user to write fast parallel analysing code. The user is responsible for many implementation details, even of the FlowVR layer. On the one hand this allows many optimization possibilities. For example one C-analyzer can represent multiple FlowVR modules or can have additional ports to communicate with other FlowVR modules too. Unfortunately, on the other hand, writing analysis logic in C is error prone. Also handling data arrays in C is not very convenient. Thus the Python-analyzer API was developed.

An Example of an analyzer written with the C-analyzer API is given in Listing 3.4.

3.4.3 Python-analyzer API

The Python-analyzer API is used to specify analyzer logic in Python. It is provided as the pypfAnalyzer python library. The Python-analyzer API was created with the objective to allow users to straight forward implement their analysis code with a minimum of surrounding code. Thus the Python-analyzer API
hides the FlowVR layer completely from the user. Therefore it uses a simple callback approach: The user defines at least a \texttt{GridMessageParser} callback function. This function gets called whenever the analyzer receives a grid message. As argument the grid message is provided as shown in section 3.1.2. Typically in the body of the callback function analysis are performed. If the analysis shows that a new steering forcing is necessary, the API function \texttt{SendSteer} is called to introduce the steering in the ongoing ParFlow simulation. An example is given in Listing 3.5.

The grid data in the received grid message and also the grid data transferred to the \texttt{SendSteer} function are hereby handled as convenient to use NumPy arrays [39].

Often we want to use an \texttt{analyzer} module to set some initial conditions in the ParFlow module. Therefore it is possible to define an \texttt{onInit} callback function. It gets called after the initialization of the FlowVR module representing the analyzer, independently from waiting messages on the analyzer's input ports (subsection 2.5.3). So it can be used to let analyzer modules act like \texttt{PreSignal} filters (section 3.1.3) introducing kick off messages into the FlowVR data flow.

To achieve all the described functionality, the FlowVR logic containing the message sending and the wait loop is handled in C routines. Also the calls to set the described callback functions are C routines. All those routines are exposed into a python library with the help of the SWIG software development tool [33, 34].

The disadvantage of the Python-\texttt{analyzer} API is that the user is more limited in to a specific structure of his analysing code compared to the C-\texttt{analyzer} API. Furthermore analysis logic in Python will run slower as it is an interpreted language.

For a quick comparison of the two analyzer APIs see Listing 3.4, Listing 3.5.
Listing 3.4: An analyzer module in the C-analyzer API.

```c
#include <fca.h>
#include <messages.h>

#define MergeMessageParser(onGridMessage)
{
...
    SendSteerMessage(...);
}

int main(int argc, char *argv[])
{
    flowvr = fca_new_empty_module();
    // Add ports to the FlowVR module
    port_in = fca_new_port("in", fca_IN, 0, NULL);
    fca_append_port(flowvr, port_in);
    // Register FlowVR module
    fca_init_module(flowvr);
    // Start message loop
    while (fca_wait(flowvr))
    {
        ParseMergedMessage(port_in, onGridMessage, NULL);
    }
    fca_free(flowvr);
}
```

Listing 3.5: An analyzer module in the Python-analyzer API.

```python
import pypfAnalyzer as pfa
import numpy as np

def onGridMessage(arr, m):
    ...
    pfa.SendSteer(...)
    pfa.SetGridMessageParser(onGridMessage)

# Run as a FlowVR module
pfa.run([])
```
4 Experiments

In this section we describe the experiments that were done with the parFlowVR framework to examine its usability in order to constrain high-resolution water resources modeling on the continental scale. For that reason we will show and explain the data flow of each experiment’s setup.

The scripts to run the presented experiments can be found in the supplement to this work (section 1.5).

4.1 Experimental Environment

For the following experiments we use openMPI 1.8.4. All Processes (ParFlow processes, analyzer modules and the writer modules) are pinned on the core they run (using openMPI’s hwloc, [32]) except for the flowvr daemon processes which are in fact also running filters.

The experiments described in section 4.2 and section 4.3 are performed on the Froggy platform of the CIMENT infrastructure. Generally nodes with 64 GB RAM and Intel(R) Xeon(R) CPU E5-2670 cores are used. For these experiments we measure the wall time (section 2.4) of ParFlow on one ParFlow process\(^1\). As all ParFlow process synchronize each time step, this gives meaningful results. The wall times on this are retrieved by injecting time measuring code using the system clock with a precision of \(1\cdot10^{-5}\)s.

The experiments from section 4.4 are performed on a local desktop machine as they do not measure wall times.

4.2 Performance of Massive File I/O

Since In Situ file writes can improve the overall performance of simulations as mentioned in subsection 2.2.3, we want to test this for the parFlowVR framework.

The data flow shown in Figure 4.1 is used for this purpose. We measure the wall time per iteration as described in section 4.1.

\(^1\)MPI rank 0
Figure 4.1: Data flow used to test the performance of In Situ file I/O. The Merge filter (comNto1PressureMerge) pile up to a tree if more ParFlow modules are added while there will stay only one netcdf-writer module. If using FlowVR with node IO such a data flow is instantiated on every participating node so that every node will have its own netcdf-writer module.

We distinguish 5 cases:

1. The no output case: no output, ParFlow alone without FlowVR running on all the cores of all nodes. Useless, as one doesn’t get the simulation results, but gives the best performance ParFlow can achieve.

2. The classic case: file output the traditional way handled by ParFlow, ParFlow running on all cores of all nodes. All cores will access the file system at the same time dumping the part of data they work on. No FlowVR is running in that case.

3. The FlowVR case: file output performed by one single helper core in the background for the whole distributed application. ParFlow running on the remaining cores.

ParFlow supports “node IO” meaning there will be only one process per node accessing the file system for data dump. Thus we compare also the two following setups with the no output case:

4. The classic Node IO case: file output handled by ParFlow with activated NodeIO. Only one core per node will access the file system writing the data that is processed by the ParFlow processes on its node. No FlowVR is running in that case.

5. The FlowVR Node IO case: file output performed by a helper core per node in the background for the whole distributed application. ParFlow running on the remaining cores. The helper core dumps the data that is produced by the ParFlow processes on its node.

All experiments are executed on 1 node (16 cores), 2 nodes (32 cores) and 4 nodes (64 cores) simulating a typical hydrologic 2D domain on the Oueme basin in Benin of 144 km * 144 km size with 1 km² spatial resolution and time steps of 0.5 h. The pressure data (double precision) for every grid cell (144 * 144) is dumped every time step leading to 162 kB of raw data per time step.

We are using ParFlow’s netCDF output module comparing it to the netcdf-writer in parFlowVR.
4.3 Performance of Variable Dump and Steering Interaction

In this experiment the ParFlow wall time per iteration is measured when transferring data from ParFlow to an analyzer and back into ParFlow.

The data flow illustrated in Figure 4.2 is instantiated on one node with 16 cores of the Froggy platform of the CIMENT infrastructure. For asynchronism 2 kick off messages are introduced by the PreSignal (see subsection 3.1.5). The analyzer in this loop is more a router: It packs the received pressure grid message as operand into a SET_PRESSURE steer message and sends it back to ParFlow. For simplicity it is written with the Python-analyzer API.

We ran the shown data flow for a ParFlow domain that is not very computational intensive\(^2\) with a size of 1000 * 1000 * 8 grid cells. After each ParFlow iteration the pressure data (1000 * 1000 * 8 * 8 Bytes ≈ 60 MB) is transferred through the FlowVR layer to the analyzer and before each iteration it is read back from the parflow module.

\(^2\)Due to constant boundary conditions the system converges into a steady state.
To get more detailed information we also measured the solver wall time\(^3\) and the wall time used by ParFlow to interact with the FlowVR layer\(^4\).

We distinguish 2 cases:

1. **The normal case**: For reference we run just the ParFlow simulation alone without output or FlowVR interaction on all 16 cores.

2. **The flowvr case**: We run the simulation on 15 cores, it is interacting with the analyzer (running on the 16th core).

As a second example we measure the steering performance for a drainage test\(^5\) on the West Africa domain as described in section 1.3. The experiment uses 4 nodes (each with 16 cores, 64 cores in all). In the non steered (ideal) case ParFlow runs on all the 64 cores. For the flowvr case the analysis is executed on a dedicated helper core. ParFlow processes run on the remaining 60 cores (15 per node). Thus a data flow similar to the one shown in Figure 4.2 is instantiated per node. There is no communication in the FlowVR layer between components executed on different nodes. Still the ParFlow processes, even on different nodes, communicate via MPI with each other. Furthermore we steer only a part of the simulation’s pressure grid: Only the grid of one ParFlow process per node is affected resulting in a total steer of 1/15 of the pressure grid cells.

---

\(^3\)The time needed to calculate the next time step, assuming all necessary variables are already in the simulation’s memory. Thus this does not contain the time for memory copies between FlowVR messages and the ParFlow memory.

\(^4\)Contains waiting for messages, copying their content in ParFlow’s memory, and the inverse: sending messages.

\(^5\)Drainage tests simulate how water drains into the soil caused by gravity, building up a surface and subsurface water stream network.
4.4 Proof of Concept: Constraint Optimization with a Feedback Loop

![Diagram of data flow used to assimilate a simulation by algorithmic steering operating In Situ. On application start the PreSignal releases its start token. The MergeItExt filter induces asynchronism.]

**Figure 4.3**

4.4.1 Twin Experiment: Optimize after Convergence of Pressure

The first real world problem which was solved with the parFlowVR framework is about finding the global manning constraint $n$ for a ParFlow simulation which converges fast into steady state conditions. This in fact is a data assimilation problem (section 2.3). We will solve it by algorithmic steerings that are calculated by a proportional controller in a simple feedback loop [37]. In the field of data assimilation this is called nudging or dynamic initialization technique [8]. The data flow in Figure 4.3 is used.

To define the manning $n_{ref}$ that shall be found we use a twin experiment (subsection 2.3.2): Simulation 1 and Simulation 2 start from the exact same input. They differ only in the manning constraints: Simulation 1 (the reference) has a constant manning $n_{ref}$ for all time steps. After simulating some time steps, the so-called spin up sequence, the system’s pressure at point $\vec{x}$ stays equal (with an error $< \epsilon$). We denote this equal pressure at point $\vec{x}$ set point $p_{ref}$.

Simulation 2 (the simulation to be assimilated) starts from the manning $n(t = 0) \neq n_{ref}$. All other inputs of Simulation 2 are identical to Simulation 1. From Simulation 2’s output the process value $p(\vec{x}, t)$ can be extracted by online measuring the pressure. The analyzer module tries to set this value ($p(\vec{x}, t)$) equal to the set point $p_{ref}$ by changing the manning constraint $n$ over time.

The analyzer can be seen as a controller in a feedback control system. The output $P_{out}$ of the proportional controller is then added to the current manning value $n$ and introduced back in the still running Simulation 2.
Given the proportional gain $K_p$ and the maximum error $\epsilon$ this leads to the following algorithm for the proportional controller:

1. receive the last pressure dump $p(t)$ from Simulation 2
2. check for convergence: if $|p(\vec{x}, t) - p(\vec{x}, t - 1)| > \epsilon$ then goto step 1
3. calculate the error: $error(t) := p_{ref} - p(\vec{x}, t)$
4. if $|error(t)| < \epsilon$ and $|p(\vec{x}, t) - p(\vec{x}, t - 1)| \leq \epsilon$ then exit assimilation
5. calculate the controller output: $P_{out} := K_p \cdot error(t)$
6. change the old manning by the controller output: $n := n + P_{out}$
7. send the new manning $n$ as steering forcing to Simulation 2 and goto step 1

As one can see in step 2 we always wait for convergence\(^6\). This is done to limit the variety of received process values and also to simplify finding a fitting proportional gain $K_p$ (Also in the sensibility analysis we wait for convergence, see subsection 4.4.2. Thus we obtain results on $p(n)$, which are especially valid after converging).

As seen in step 4, the assimilation ends if the pressure has the desired value (within $\epsilon$ precision) and it converged as in step 2.

The used parFlowVR network ensures that Simulation 2 runs in parallel to the analyzer. Simulation 2 will never send the data of the same timestep $t$ more than once to the analyzer.

Since we do not know much about the analyzer runtime we cannot predict when the steering forcing produced by it will be taken into account from the ParFlow simulation.

### 4.4.2 Calibration of the Proportional Gain $K_p$

To find the proportional gain $K_p$ a sparse sensibility analysis for $p(\vec{x}) = f(n)$ is done. Therefore we ran multiple simulations of the problem (Simulation 2) that we later run the assimilation on, each time only changing the manning parameter $n$ in a meaningful interval:

$$n \in [0.00001, 0.0001]$$

until the convergence criteria from subsection 4.4.1 step 2 holds true. We set $\epsilon = 10^{-10}$. The retrieved curve can be roughly described by linear function

$$p(\vec{x}) \approx a \cdot n + b$$

shown as fit in Figure 4.4.

As

$$\Delta n = P_{out} = K_p \cdot error(t)$$

$$\Delta p(\vec{x}) = p_0 - p(\vec{x}, t) = error(t)$$

\(^6\)pressure changes $< \epsilon$
4.4.3 Experiment Parameters

For the experiment we set $K_p$ according to Figure 4.4:

\[ K_p := 0.00475 \text{ min} \cdot m^{-1/3} \cdot \text{mWg}^{-1} \]  

(4.6)

and

\[ n_{\text{ref}} := 0.0000111 \]  

(4.7)

Simulation 2 starts with a global manning constraint set to

\[ n := 0.0000222 \]  

(4.8)

4.4.4 Regulate a Possibly Non Converging Pressure

Of course there are cases where a steady state for the process value $p(x', t)$ will probably never be reached by the simulation. Thus we did the same experiment as in subsection 4.4.1 without the convergence condition in step 2. The used proportional gain $K_p$ was left unchanged.
5 Results

The results of the performed experiments are presented in the following. This will show how efficient parFlowVR is to solve typical scientific tasks in hydrological simulation.

The result’s raw data can be found in the supplement to this work (section 1.5).

5.1 Repetition of tests and warmup times

All tests were executed multiple times. Since using different cluster nodes and variabilities in the cluster workload, especially the wall time measurements fluctuate. To obtain nevertheless meaningful results multiple runs were performed.

Due to run time optimizations as caching and execution path prediction, the measured wall times, especially for the first simulation cycles, are considered unstable. Thus calculations of averages, medians and standard deviations of wall times are ignoring the first twenty time steps of the test runs in the following.

5.2 Performance of Massive File I/O

![Figure 5.1](image)

**Figure 5.1:** Wall time of ParFlow per time step for a simulation with no output, output written by the classic implementation (no FlowVR) using each core for writing and the FlowVR In Situ case where only one helper core in the distributed application performs the writing. The measures are performed on 2 nodes (32 cores) of the Froggy platform.
The results of the tests on In Situ file write can be seen in Figure 5.1. Here we compare the classic netCDF output implemented in ParFlow with the case where we run ParFlow with no output or with output through FlowVR using parFlowVR. In the classic case every participating core (namely 32 cores for the case in Figure 5.1) accesses the file system for writing after each simulation time step. In the FlowVR case we run the ParFlow simulation on one core less (31 cores instead of 32 cores for the chart given in Figure 5.1). The 32nd core executes file writing.

One netCDF file with a size of \(\approx 992\, \text{kB}\) is written per time step.

It turns out that using the classic way of output in ParFlow (using the netCDF output module integrated in ParFlow) is much slower and the wall times are more variable than using the In Situ way with FlowVR. Especially when working on a single node this variability is high. Table 5.1 and Figure 5.2 illustrate this. Also classic node IO (using the netCDF output module integrated in ParFlow doing the file write only on one core per node) compared to node IO with FlowVR (one helper core performs the file write in the background) shows this characteristics if executing on multiple nodes (see Table 5.1).

<table>
<thead>
<tr>
<th>Method</th>
<th>1 node (16 cores)</th>
<th>2 nodes (32 cores)</th>
<th>4 nodes (64 cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParFlow simulation with no output</td>
<td>0.043</td>
<td>0.020</td>
<td>0.022</td>
</tr>
<tr>
<td>classic ParFlow output performed by every core</td>
<td>1.165</td>
<td>0.352</td>
<td>1.464</td>
</tr>
<tr>
<td>FlowVR output with one writer in the distributed application</td>
<td>0.010</td>
<td>0.021</td>
<td>0.036</td>
</tr>
<tr>
<td>classic node IO output performed by one core per node</td>
<td>0.081</td>
<td>0.254</td>
<td>0.156</td>
</tr>
<tr>
<td>FlowVR node IO output with one writer per node</td>
<td>0.014</td>
<td>0.015</td>
<td>0.021</td>
</tr>
</tbody>
</table>

Table 5.1: Mean of the standard deviation of ParFlow wall times in seconds per time step over the considered 76 time steps from 4 runs.

The overall wall times for ParFlow simulations on increasing cluster sizes using different output configurations are shown in Figure 5.3 and Figure 5.4. For the FlowVR case there cannot be seen a systematic difference in the wall times between the cases with one writer per node (node IO) or one writer per distributed application.
5.3 Performance of Variable Dump and Steering Interaction

Figure 5.3: The accumulated ParFlow wall time (76 time steps) for the cases no output, classic output (every core is writing) and FlowVR output (one core in the application performs output)

Figure 5.4: The accumulated ParFlow wall time (76 time steps) for the cases no output, classic node IO output (one core per node is writing) and FlowVR node IO output (one dedicated core per node performs output in the background)

Figure 5.5: ParFlow wall time per time step comparing an unsteered numerically easy simulation with the steered version of it. The steer affects ≈60 MB of grid values per iteration.
As seen in Figure 5.5 steering a simulation on a numerically easy to calculate domain (since with constant boundary conditions) of 1000 * 1000 * 8 grid cells increases the wall time per timestep in average by less than 10%. A closer measurement gave that 25% of the 10% additional wall time is from interaction with the FlowVR layer especially waiting for messages whereas 75% is time lost in other parts of the ParFlow code.

The computing power which is taken away due to freeing one core for the analysis is negligible: comparing the wall times of the unsteered simulation running on 15 respectively 16 cores gave an average difference in wall time smaller 1.5% of the wall time on 16 cores.

Further measurements (e.g. Figure 5.6) have shown that the increase in wall time caused by steering depends on many factors. The most important ones are:

- the domain size
- the numerical complexity of the problem
- the complexity of the analyzer code
- the amount of the steered grid cells
- the change that the steer introduces
- the amount of cores taking part in the distributed application

On the West Africa domain (Figure 5.6) we experienced that it needs in average ≈7% more wall time per iteration when steering. Further measurements show that ≈88% of this additional time is caused by an increase of the solver wall time. Thus the actual variable change caused by the steering and the interaction with the FlowVR layer are responsible for less than ≈12% of the wall time increase.

![Figure 5.6: Steering on the West Africa domain](image)

Figure 5.6: Steering on the West Africa domain (section 1.3). ParFlow wall time per time step comparing an unsteered numerically difficult simulation with the steered version of it. The steer affects ≈12 MB of grid values per iteration.

---

1 e.g. simulations with strongly varying boundary conditions are considered more complex than those leading to a steady state due to constant boundary conditions.

2 The 10 z-layer system takes very long to spin up making it difficult to average the wall time over multiple time steps. Thus we present measurements of the West Africa domain with only 5 z-layers resulting in a system of 2880 * 1600 * 5 grid cells.
5.4 Proof of Concept: Constraint Optimization with a Feedback Loop

Figure 5.7 shows that it takes much more time steps to regulate to the optimal manning in the case where we wait for pressure convergence before steering. The assimilation for this case ends after 583 time steps. The assimilation if skipping the convergence criteria finishes after 189 time steps. Still in both cases the correct manning with a deviation less than $10^{-4} \%$ is found.

![Figure 5.7: Regulation of the pressure by changing the manning parameter. Left column: wait for convergence before steering. Right column: do not wait for convergence but steer directly after receiving simulation output. Red dashed line: set point. Blue dots: pressure. Green dots: manning.](image)
6 Discussion

In this chapter we discuss the results of the performed experiments. We will give an explanation of the observations allowing to derive guidelines for further use of ParFlow, FlowVR and the parFlowVR framework. Further we will explain how parFlowVR could be used for data assimilation purposes in the future.

6.1 Performance of Massive File I/O

Using helper cores or nodes for file output decreases the number of cores performing numerical calculation. Still this has an overall positive effect on the wall time of ParFlow simulations as allowing file system access to be performed in background not blocking all ParFlow processes. Taking away some cores from the simulation is negligible on the tested domain as seen in Figure 5.1 Figure 5.3 and Figure 5.4 as the no output case and the FlowVR case have almost the same wall times.

Furthermore the variation between the single runs if using the classical output is much higher than in the FlowVR case. The reason is the increasing unpredictability of the order of file system accesses when every of the 16/32/64 ParFlow processes tries to access the file system. From the discrepancy between the average (mean) and the median in Figure 5.2 we learn further that the experienced variation is probably not normal distributed.

Running ParFlow on one node with activated node IO turned out to be almost as fast as if In Situ writing with one writer in the background. This shows that writing the test domain’s output to the disk has no big slowdown itself. The seen slowdown is caused by concurrent file system accesses that are blocking each other while the next time step can only be calculated if all the processes finished the write. For more nodes we cannot state for sure that classic output (no FlowVR) with node IO performs faster than classic output without node IO but as seen in Table 5.1, at least the variation between different runs is much smaller. The reason is that less file accesses from different processes are performed (from 16 processes per node accessing the file system going down to one) making the scheduling of file accesses more predictable and less influenced by race conditions.

The results of the classic case regarding the wall time development on one, two and four nodes is not as expected, even if taking the rather high standard deviation into account. To explain this comportment further investigation on the implementation of the ParFlow netCDF output module as well as wall time measurements of different parts of its code would be necessary.

In general ParFlow users can largely benefit from In Situ file write with FlowVR. For the FlowVR case and the measured problem size we could not see any differences between node IO and the case with only one writer in the whole distributed application. Still we recommend to use the FlowVR node IO variant even if it decreases more the ParFlow computation resources than having just one writer. The reason is that node IO will probably scale better when using more than 4 nodes as the communication in the FlowVR layer can be performed completely in the shared memory of the nodes avoiding network latency.
6.2 Performance of Variable Dump and Steering Interaction

The results from section 5.3 show that doing analysis In Situ can notably increase (10%) the wall time of a ParFlow simulation, even if executing the analyzer in parallel with the simulation. (In this experiment we allow the simulation to calculate the next time step without taking into account the analyzer results of the time step directly before. This is achieved by inserting 2 kick off messages as described in subsection 3.1.5.)

In the experiment the increased wall time is caused by 3 main reasons:

1. The solver normally benefits from the processor cache. Changing (steering) a large amount of data in the memory destroys the cache structure. Thus the next memory accesses from the solver take longer.

2. After each ParFlow process was steered, changing the grid part it holds, a cell update has to be broadcasted to other processes depending on those changed cells. This synchronization takes extra time.

3. Furthermore, if the model stepping takes less time than it takes for the data to traverse the different filters and modules connected by the FlowVR layer this increases the simulation’s wall time too as we used a synchronous flow where the simulation will wait for a steer message before every time step.

As written in section 5.3 we experienced nearly the same wall times on 15 and 16 cores for the normal simulation run (no FlowVR, no steering). Since the test case is not very computational intense, the wall time depends in large parts on the duration of communications. Thus when adding a 16th core the resulting more communication almost counters the positive effect of the computing power added by the 16th core.

The most hydrological workflows with steering will differ in 2 main points from our test case:

- The simulation problems are computationally more intensive.
- The size of the steering operand will not consider the whole grid.

Thus the simulation will wait less for the analyzer and the merge trees concatenating data. Further due to a smaller size the cache won’t be as affected as in our experiments. Both hypothetically lead to an even smaller impact of steering on the simulation’s wall time than the measured 10%. Still computational more intensive simulation could also benefit more from having the extra cores.

The steering performance test for the West Africa domain (Figure 5.6) gives an illustration for this. Since here the simulation is numerical intensive (about 23 seconds per time step on 64 cores without steering) running ParFlow on less cores (namely 60 for the steered case) has a higher impact on the wall time. Still we steer less data fields and thus also the cache structure is less influenced having a positive impact on the performance compared to the experiment shown in Figure 5.5. Finally the simulation is also not waiting for messages anymore as simulating takes much longer the simulation output messages have the time to traverse the FlowVR layer (including the analyzer on the helper core) before awaited by the simulation. Using 60 instead of 64 cores, freeing 4 helper cores, is a computational resource lost of ≈6% for the ParFlow simulation. Assuming ideal scalability this could explain already the largest part of the increased wall time of ≈7%. This and the fact that ≈88% of the wall time increase comes from the solver
time itself show that introducing In Situ analysis for simulations on the West Africa domain creates just a small foreseeable increase in wall time.

To sum up the extra wall time added due to steering is highly application dependent. The discussion in this section can still give important guidelines when optimizing the wall time of steering workflows.

The increase of ParFlow wall time when steered is smaller than the time which would be needed when performing analysis using intermediate files. To reach the same analysis resolution as in the experiment, time consuming restarting of the simulation software itself would take place after every time step. In addition, the file writes from the simulation and the according reads from the analyzer followed again by writing the analyzer results to disk, are skipped when using steering.

6.3 Proof of Concept: Constraint Optimization with a Feedback Loop

The experiment from section 4.4 shows that data assimilations with the implemented parFlowVR framework are possible. Furthermore it shows that the ParFlow model itself is robust: We did not specify exactly when parameter changes by steerings are introduced due to the asynchronous data flow in Figure 4.3. They could influence already the next simulation time step after the `parflow` module sent data to the analyzer or any later time step. Still we get precise values for the assimilated manning parameter. If introducing the steering in the system while it is in an transition state due to the last steering, the distributed application finds the good manning too (the case if we do not wait for convergence before steering). The latter case needs even less time steps.

Due to FlowVR’s flexibility the data flow used for this experiment (Figure 4.3) can easily be parallelized using e.g. more ParFlow processes or analyzer modules. Especially when adding more ParFlow processes we expect the same results produced in less wall time. In contrast possibly more time steps will be needed as steering forcings get introduced later due to more communication in the FlowVR layer and merge with more layers.

6.4 Data assimilation in parFlowVR

6.4.1 Twin Experiments

When developing data assimilation systems for large scale models it is reasonable to perform twin experiments (subsection 2.3.2) online, avoiding unnecessary file writes and reads of the reference trajectory $y_{ref}$. This can be achieved by launching two or more ParFlow models in parallel and synchronously where one runs on a fixed parameter set as reference. The others have fixed initial conditions of their parameter sets only, since they are steered by the assimilation process in the parFlowVR distributed application.

We used the data flow described in Figure 6.1. To add a perturbing noise to the reference trajectory $y_{ref}$ a perturbation module could be implemented and added in between `parflow_2` module and the `MergeTree` filter.
**Figure 6.1:** Data flow for a simple twin experiment. Here with two ParFlow simulations running synchronous. `parflow_1/0` is the reference. `parflow_2/0`’s state is assimilated. Note that the merge of the two simulation’s dumps assures synchronization as the merge filter blocks until having a message on every input port. The Python-Analyzer knows from the message header which part of the merged message belongs to which simulation.
To let the Python-Analyzer run in parallel, the PreSignal is instantiated with at least 2 initial messages. So there would be an empty message available that kicks off the ParFlow simulations when the Python-Analyzer is busy for the first time (see subsection 3.1.5).

### 6.4.2 Variational Data Assimilation and Kalman Filters

Variational data assimilation can be implemented in parFlowVR with a data flow similar to the one described in section 4.4, Figure 4.3. This workflow avoids unnecessary file I/O and model reinitialization at the same time parallelizing analysis and model execution. It can be flexibly scaled on more cores or even nodes. When enlarging the domain size the analyzer that is evaluating the whole domain’s grid can be a bottleneck. To tackle this a data flow consisting of more analyzers (e.g. one per node) each calculating just the error from the same or neighbouring nodes can be used. After error calculation the error is sent to the main analyzer which then calculates the system state change (steer) from it.

To implement Kalman filters (see section 2.3) in the parFlowVR framework, a similar data flow can be used. Only the analyzer module must be changed to perform the Kalman filter algorithm.

The data flow shown in Figure 4.3 does not ensure that the simulation runs exactly one iteration for each analysis step. As mentioned in section 6.3 we nevertheless retrieve the desired results (namely a state $x_{opt}$ according to the measurement). Depending on the objective of the data assimilation it might become necessary to run simulation and analyzer modules in sync. This can be achieved by an even simpler data flow consisting of a loop containing an analyzer module that sends (optionally through a 1 to N tree, see section 2.5.4) its output to the ParFlow simulation module that again sends its results (through a merge tree) to the analyzer module. Depending where the data flow should start a PreSignal filter must be introduced.

Further, data flows with different analyzers processing different simulation data and e.g. steering different domain areas in the ParFlow simulation are possible.

### 6.4.3 Data Assimilation Methods Propagating Multiple System States

When assimilating data using e.g. ensemble Kalman filters or particle filters, the model and the observation operators $M$ and $H$ are called on multiple states $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$. Applying the operators on the state vectors is one of the most computation resources consuming part of these data assimilation methods. Thus we want to run it In Transit. In parFlowVR we propose the data flow graph illustrated in Figure 6.2.

If the model state $x$ contains lots of dimensions adding helper cores per simulating node that do data compression before transferring it to the analyzer might become necessary.

Furthermore due to the flexibility in the FlowVR middleware backing parFlowVR, it is possible to move the steps of the assimilation algorithm in different modules running in parallel. For example for the ensemble Kalman filter the mean of all ensemble states needs to be calculated each iteration. This could be outsourced in modules running in parallel, each operating on a subset of state vector entries.

An other improvement e.g. for particle filters is to filter out the observation entries from the simulation’s observation $(Hx)$ that we have measurements for, as the particle’s model states $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ are not used in the particle filter iterations and thus can be kept in the simulation’s memory.
Figure 6.2: Data assimilation in FlowVR. The analyzer module does the calculation needed for the data assimilation. The cost intensive model iterations are outsourced. Dashed components are optional and can be replaced by a simple connection if not needed. Arrows: FlowVR port connections between components. Dashed arrows: additional MPI connections between components (not touched by parFlowVR).
7 Conclusion and Future Work

In this chapter we want to draw a conclusion about the work which has been done during this master’s thesis. Also the possibilities parFlowVR opens in the field of large scale high-resolution simulations are stated and an outlook on future work that we plan to do in this field is given.

7.1 Conclusion

With parFlowVR a powerful tool to do online steering and analysis of the ParFlow hydrological model is developed. This is backed by the FlowVR middleware which could once more show its performant and flexible integration on high-performance platforms.

parFlowVR introduces In Situ and In Transit workflows for ParFlow. This facilitates analysis and data assimilation to be performed online (while the ParFlow simulation is running and its data still rests in the simulation’s memory) avoiding file I/O and extra initialization of the simulation, resulting in higher performance. A downside is that often the analysis which shall be performed on the simulation output is not well defined before simulation run time yet. In these cases users can still benefit from parFlowVR as it enables In Situ and In Transit file output being much faster than comparable classic output methods of ParFlow (section 5.2).

Furthermore parFlowVR enables online inspection of running simulations. Thus users can react much easier and with less output file generation to the current simulation state.

Steering and online observation together will allow interactive workflows in the future. Users will be able to change the simulation parameters and state while it is running. Thus the user gets a direct response from the simulation omitting the need to store results to the disk and load them again for later inspection. This is an efficient way to reduce the data stored to disk, especially during exploration of simulation output response to input changes. Thus the time humans need to process the simulation output later is shortened too since less meaningless output is produced.

In particular the experiment of section 4.4 gives a proof of concept of the work the PHYREV team\(^1\) from the IGE plans to perform in the next years: The parameters of continental scale ParFlow simulations shall be assimilated using In Situ data assimilation and online calculated steerings.

Further it could be shown that even steering a domain of multiple million pixels as the West Africa domain (section 1.3) is possible with parFlowVR. (section 4.3 and Figure 5.6) the next step will be integrating the needed data assimilation methods with FlowVR.

Thus the In Situ middleware FlowVR turns out to be appropriate for numerical simulation tackling the upcoming data challenge due to to the increasing amount of measurements e.g. coming from the internet

\(^1\)Cohard et al., http://www.ige-grenoble.fr/recherche/equipes/phyrev/
of things (IoT) and satellites. The developed codes in the environment of the parFlowVR framework can act as template for integrating other numerical models with FlowVR. Some of the modules and filters implemented for parFlowVR can even be reused in different contexts then.

### 7.2 Future Work

In the current parFlowVR implementation it is not yet possible to steer the entries of all simulation parameters. Efforts in this direction will be done in the future as this allows many new scenarios as for instance In Situ and In Transit data assimilation regarding the whole set of model parameters and model state variables.

It is further planned to implement modules for intelligent output filtering, which produce less, but higher-level output. For instance a module which writes only regional variable dumps of the grid cells affected by a local rain event is planned.

The integration of studies on temporal parameter evolution into ParFlow simulations is an other field of interest. For instance it is planned to use the parFlowVR framework to couple a ParFlow simulation with a surface permeability model which returns the temporal permeability response due to regional soil work [22].

Further work will be done on implementing modules providing a strong toolkit to do online variational, statistical and hybrid data assimilation as shown in section 2.3 using In Situ and In Transit approaches. A first start might be porting the PDAF framework [28] into FlowVR modules. This enables distributed data assimilation applications as described in subsection 6.4.3. Evaluating the model response to parameter changes using an In Transit approach in the parFlowVR environment where new computational resources (cluster nodes) are allocated during run time (similar as done in [36]) seems promising too.

Finally FlowVR modules abstracting other models are a field of interest. This permits to define model couplings elegant and flexible in the FlowVR layer. In contrast to coupling frameworks like OASIS3-MCT, FlowVR allows more flexible couplings depending on the runtime environment. In addition, the data flow between the models can be defined much easier (due to data flow graphs) and in greater detail. Thus FlowVR poses a serious alternative to the standard approach achieved by coupling frameworks like OASIS3-MCT.
Bibliography


[12] Laura Condon and Reed Maxwell. EVALUATING GROUNDWATER SURFACE WATER INTERACTIONS ACROSS THE CONTINENTAL U.S. WITH AN INTEGRATED HYDROLOGIC MODEL. January 2016. doi: 10.1130/abs/2016AM-286862. 3, 6

47


[27] Clément Mommessin, Matthieu Dreher, Tom Peterka, and Bruno Raffin. Automatic Data Filtering for In Situ Workflows. September 2017. URL https://hal.inria.fr/hal-01581032/document. 21


[34] SWIG and Python. URL http://www.swig.org/Doc1.3/Python.html. 25


