Capturing Provenance for Runtime Data Analysis in Computational Science and Engineering Applications

Vítor Silva, Renan Souza, Jose Camata, Daniel de Oliveira, Patrick Valduriez, Alvaro Coutinho, Marta Mattoso

Capturing Provenance for Runtime Data Analysis in Computational Science and Engineering Applications

Vítor Silva[1], Renan Souza[1,2], Jose Camata[1,3], Daniel de Oliveira[4], Patrick Valduriez[5], Alvaro L.G.A. Coutinho[1], Marta Mattoso[1]

Abstract. Capturing provenance data for runtime analysis has several challenges in high performance computational science engineering applications. The main issues are avoiding significant overhead in data capture, loading and runtime query support; and coupling provenance capture mechanisms with applications built with highly efficient numerical libraries, and visualization frameworks targeted to high performance environments. This work presents DfA-prov, an approach to capture provenance data and domain data aiming at high performance applications.

Keywords: Provenance; User Steering; Computational Science and Engineering; HPC.

1 Introduction

Computational Science and Engineering (CSE) applications are based on computational models that solve problems typically requiring High Performance Computing (HPC) [1]. CSE applications are not tied to a particular domain. They can be found in biology, chemistry, geology, several engineering areas, etc. They have the exploratory nature of scientific applications but have to deal with large-scale executions, which last for a long time even when using HPC. The software ecosystem for developing these applications involves much more than writing scripts or invoking a chain of legacy scientific codes. Computational scientists develop their simulation codes based on complex mathematical modeling that results in invoking components of CSE frameworks and libraries. For example, components are invoked to provide for: (i) support for PDE discretization methods like libMesh, FEniCS, MOOSE, deal.II, GREENS, OpenFOAM; (ii) algorithms for solving numerical problems with parallel computations, like PETSc, LAPACK, SLEPc; (iii) runtime visualizations, like ParaView Catalyst, VisIt, SENSEI; (iv) parallel graph partitioning, like ParMetis, Scotch; and (v) I/O data management like ADIOS.

As a result, a typical CSE software code works like a script, in the sense that to code the underlying mathematical modeling it requires invoking functions, components, or APIs from these libraries or frameworks. Fig 1 shows a fragment of the FEniCS Python code for solving the Cahn-Hilliard equation, a mathematical model from material science. The Cahn-Hilliard equation leads to a prototype of a transient nonlinear multi-physics code. Several parameters have to be set to invoke these highly efficient components, which are very difficult to preset and need monitoring for runtime fine-tuning. The Interoperable Design of Extreme-scale Application Software

[1] COPPE / Federal University of Rio de Janeiro, Brazil
[2] IBM Research
[3] Federal University of Juiz de Fora, Brazil
[4] Fluminense Federal University, Brazil
[5] Inria and LIRMM, Montpellier, France

(IDEAS) [2] is a family of projects, involving several institutions in the US, concerned with the complexity of developing software for CSE applications. IDEAS aims at "enabling a fundamentally different attitude to creating and supporting CSE applications" with desirable features like provenance and reproducibility [3]. In fact, provenance data can help in registering parameter choices. Associating them to results can improve both fine-tuning and data analyses at runtime.

```
dataflow_tag = "fenics-df"
t1 = Task(1, dataflow_tag, "MeshCreation")
t1.add_dataset(DataSet("iMeshCreation", [Element([96, 96])]))
# Create mesh
mesh = UnitSquareMesh(96, 96)
t1.add_dataset(DataSet("oMeshCreation",
                [Element([mesh.num_vertices(), mesh.num_cells()])]))
t1.end()

t2 = Task(2, dataflow_tag, "FunctionSpace", dependency=t1)
t2.add_dataset(DataSet("iFunctionSpace", [Element(["Lagrange", 1])]))
# Define function spaces
V = FiniteElement("Lagrange", mesh.ufl_cell(), 1)
ME = FunctionSpace(mesh, V*V)
t2.add_dataset(DataSet("oFunctionSpace", [Element([ME.dim()])]))
t2.end()

# parts of code were omitted
# (...)

t3 = Task(3, dataflow_tag, "NewtonSolver", dependency=t2)
t3.add_dataset(DataSet("iNewtonSolver",
                [Element(["lu", "incremental", 1e-6])]))
# Define Newton solver
solver = NewtonSolver()
solver.parameters["linear_solver"]       = "gmres"
solver.parameters["convergence_criterion"] = "incremental"
solver.parameters["relative_tolerance"]    = 1e-6
t3.add_dataset(DataSet("oNewtonSolver",
                [Element(["gmres", "incremental", 1e-6])]))
t3.end()
# continue in next frame
```

```
# Output file
file = File("output.pvd", "compressed")

# Step in time
t = 0.0; T = 50*dt; i =0
prev = t3
while (t < T):
   t += dt; i += 1
   current = Task(int(t3._id)+i ,dataflow_tag,"TimeStep", dependency=prev)
   current.add_dataset(DataSet("iTimeStep", [Element([t,dt])]))
   # Solver execution
   u0.vector()[:] = u.vector()
   iter_count, converged_flag = solver.solve(problem, u.vector())
   current.add_dataset(DataSet("oTimeStep",
                   [Element([converged_flag,iter_count,solver.residual()])]))
   current.end()

   twrite = Task(int(current._id)+1, dataflow_tag, "Visualization"+iter_count,
                   dependency=current)
   twrite.add_dataset(DataSet("iVisualization", [Element(["output.pvd"])]))
   # Visualization
   file << (u.split()[0], t)
   # Raw data extraction
   extracted_data = Extractor(ExtractorCartridge.PROGRAM, "output.pvd")
   twrite.add_dataset(DataSet("oVisualization", [Element(extracted_data[i-1])]))
   twrite.end()
```

Labels:
Black → Python native code
Red → FEniCS invocation
Green → DfAnalyzer invocation
Purple → VTK invocation

Fig 1 - FEniCS Python script for the Cahn-Hilliard equation adapted from [4].

Despite the several solutions available for making applications provenance-aware [5–7], capturing provenance data in CSE applications is still an open issue. The challenges are mainly related to performance and provenance granularity. Stamatogiannakis *et al.* [5] evaluated tradeoffs in provenance capture mechanisms. They consider that solutions that are easy to deploy collect provenance in a very fine grain and present a significant overhead, while solutions that are based on function calls present low overhead and granularity is controlled by the code instrumentation. The disadvantage of inserting function calls is the need to have access to the code. This is not an issue in CSE applications as very often the code to be instrumented (Fig1) is written by the computational scientist, who can assist in inserting the calls.

In CSE applications, the mechanism for provenance capture has to be deployed in an HPC environment and preferably manage provenance data, asynchronously, in computing nodes separate from the application. This separation avoids resource competition, particularly in the memory hierarchy data space. Since CSE data are very large, provenance capture cannot be in fine grain. Capturing provenance at the operating system or file level is not an option. CSE applications, like the one in Fig. 1, are written in languages, like Python and C/C++, which are mapped to the CSE software ecosystem, therefore solutions that are language specific are a limitation. HPC Scientific Workflow Management Systems (SWMS) would be a natural solution for CSE. However, conflicts among the parallel execution control of the workflow engine and the CSE libraries prevent using SWMS in CSE software.

This work presents DfA-prov, an approach that follows the PrIMe methodology [8] to make CSE applications provenance-aware and to provide runtime data analysis. DfA-prov is language agnostic and does not present the limitations of capture mechanisms that compete with the computing nodes that execute the CSE application. DfA-prov adopts DfAnalyzer [9] as provenance-aware components to be invoked by the CSE applications. It works in the same way computational scientists invoke the CSE and visualization libraries. Provenance data is captured by directly accessing input data and parameters of the CSE function calls using *in-situ* and *in-transit* approaches. To address the limitation of having coarse-grain provenance, DfA-prov provides function calls that access raw data from files. In a previous work [10], we used DfAnalyzer tightly coupled to a CSE application observing negligible overhead (less than 1%) in its provenance capture, while providing rich data analytics at runtime. These results encouraged us to propose DfA-prov as a standalone library with a corresponding methodology to help on the adoption of provenance capture in CSE applications.

2 DfA-prov making CSE applications provenance-aware

DfA-prov follows the PrIMe methodology [8] to address CSE challenges for provenance capture. After applying the methodology, DfA-prov generates a provenance database, W3C PROV-compliant, enriched with domain data to be queried at runtime or after the CSE application execution. DfA-prov is based on two main components from DfAnalyzer, the provenance data capture and the raw data extractor.

PrIMe defines three phases. The first phase is an analysis step that identifies questions related to provenance for data analysis. More specifically, Phase 1 identifies data items and data transformations (or processing steps), all to be modeled using a data representation. Phase 2 iteratively analyzes the application structure to identify actors and interactions that provide the data items and data transformations to be registered as provenance data. Phase 3 aims at adapting the application to capture provenance data. We adapted these phases to match CSE application requirements.

DfA-prov requires a collaboration between the CSE application developer (named as user) and a PROV specialist, as expected in Phase 1. The user identifies data items to be tracked and how it relates to other data items along its lineage. The PROV specialist models the data transformation chain using W3C PROV-DM activities and entities with extensions for the domain data items, particularly data that need to be extracted from raw data files. The result of this phase is a UML class diagram. The UML classes are then mapped to a relational provenance database. The participation of the user in this data modeling helps on query formulations. In addition, it selectively chooses only application data of interest to be registered, providing a coarse-grain with relevant provenance data and selected raw data. In Fig. 1, examples are: solver convergence, number of iterations, and residual norms.

Provenance library calls are inserted in the CSE application as shown in Fig. 1 as *input*, *output*, *task* and *output* followed by an *extracted data* call. Similarly to PROV-Template [6], DfA-prov has a set of RESTful services (and libraries on C++, Python, and Java) to help plugging the calls into the CSE applications. The invoked provenance components capture data asynchronously during the CSE application execution. They get the data and send all insert/update requests to a columnar database system that runs in computing nodes different than the CSE application. As new phases with-

in DfA-prov, users configure CSE applications coupled to provenance-aware components to specify input parameter values and the HPC environment. Then, they submit provenance monitoring queries like *what is the average error estimate calculated in all iterations so far*. Users can submit provenance queries using graphical interfaces or SQL queries based on a dataflow abstraction. Finally, the monitoring helps parameter fine tunings on the CSE application as evidenced in [10]. Real life applications are much more complex than the script in Fig. 1, involving monitoring at runtime on an HPC machine quantities of interest over time, metadata to visualization snapshots, nonlinear systems solves, mesh adaptation parameters etc. These issues can be seen in [10] for a particular CSE application, with examples in [11].

3 Conclusions

DfA-prov is an approach for making CSE applications provenance-aware and providing runtime data analytics. DfA-prov is based on application analysis, provenance data modeling, and provenance-aware components to be invoked by the applications. In addition to well-known advantages of collecting provenance in CSE applications, such as reproducibility and reliability, runtime provenance augments online data analytical potential and is especially useful for CSE simulations in large-scale. Visualization tools (*e.g.*, ParaView Catalyst) have been coupled to DfA-prov calls to complement domain data analyses. Based on runtime data analyses, the user may dynamically adapt dataflow elements.

Acknowledgments

We thank Vinícius Campos for his help in DfA-prov development. The research has received funding from CAPES, CNPq, FAPERJ and Inria (SciDISC projects), the European Commission (HPC4E H2020 project), and the Brazilian Ministry of Science, Technology, 290 Innovation and Communications. It has been performed (for P. Valduriez) in the context of the Computational Biology Institute.

References

1. Rüde, U., Willcox, K., McInnes, L.C., Sterck, H.D., Biros, G., *et al.*: Research and Education in Computational Science and Engineering. CoRR. abs/1610.02608, (2016).
2. IDEAS productivity, https://ideas-productivity.org.
3. Bernholdt, D., Dubey, A., Heroux, M., Klinvex, A., McInnes, L.C.: Improving Reproducibility Through Better Software Practices. SIAM Conference on CSE, Atlanta, GA (2017).
4. Alnæs, M., Blechta, J., Hake, J., Johansson, A., Kehlet, B., *et al.*, Archive Of Numerical Software: The FEniCS Project Version 1.5. University Library Heidelberg (2015).
5. Stamatogiannakis, M., Kazmi, H., Sharif, H., Vermeulen, R., Gehani, A., *et al.*: Trade-Offs in Automatic Provenance Capture. In: IPAW, pp. 29–41. Springer International Publishing, Cham (2016).
6. Moreau, L., Batlajery, B.V., Huynh, T.D., Michaelides, D., Packer, H.: A Templating System to Generate Provenance. IEEE Trans. Softw. Eng. 44, 103–121 (2018).
7. Pimentel, J.F., Murta, L., Braganholo, V., Freire, J.: noWorkflow: a tool for collecting, analyzing, and managing provenance from python scripts. PVLDB 10, 1841–1844 (2017).
8. Miles, S., Groth, P., Munroe, S., Moreau, L.: PrIMe: A methodology for developing provenance-aware applications. ACM Trans. Softw. Eng. Methodol. 20, 1–42 (2011).
9. Silva, V., De Oliveira, D., Valduriez, P., Mattoso, M.: DfAnalyzer: Runtime Dataflow Analysis of Scientific Applications using Provenance. In: PVLDB. Rio de Janeiro, Brazil (2018).
10. Camata, J.J., Silva, V., Valduriez, P., Mattoso, M., Coutinho, A.L.G.A.: In situ visualization and data analysis for turbidity currents simulation. Comput. Geosci. 110, 23–31 (2018).
11. DfAnalyzer tool demonstration, https://github.com/vssousa/dfanalyzer-spark.