Automated Provenance Collection for CCA Component Assemblies

Kostadin Damevski, Hui Chen

Virginia State University
Petersburg, VA, 23806, USA
{kdamevski,hchen}@vsu.edu

Abstract. The problem of capturing provenance for computational tasks has recently received significant attention, due to the new set of beneficial uses (for optimization, debugging, etc.) of the recorded data. We develop a provenance collection system aimed at scientific applications that are based on the Common Component Architecture (CCA) that alleviates scientists from the responsibility to manually instrument code in order to collect provenance data. Our system collects provenance data at the granularity of component instances, by automatically recording all method invocations between them, including all input and output parameters. By relying on asynchronous communication and using optimizations to handle large data arrays, the overhead of our system is low-enough to allow continuous provenance collection.

1 Introduction

Provenance is a collection of intermediate data that explains in some level of detail the transition from input data to output data in a scientific application. This type of data collection also exists in many other computing domains, under different constrains and application requirements; provenance collection is similar to logging in operating systems and to preservation of data lineage in databases. The form and granularity of data provenance depends on the type of application that is intended to consume this stored data. The collected data can be used for purposes ranging from error detection and debugging, to optimization by removing redundant computations, and even to accountability of individual parts in a multi-user system [1, 2]. In the scientific computing domain, provenance also serves the role of a "paper-trail" , in concert with the source code, to document the computational methods used to a particular scientific discovery.

In recent years, component technology has been a successful methodology for large-scale commercial software development. Component technology encapsulates a set of frequently used functions into a component and makes the implementation transparent to the users. Application developers typically use a group of components, connecting them to create an executable application. Component technology is becoming increasingly popular for large-scale scientific computing in helping to tame the software complexity required in coupling multiple disciplines, multiple scales, and/or multiple physical phenomena. The Common

Component Architecture (CCA) [3] is a component model that was designed to fit the needs of the scientific computing community by imposing low overhead and supporting parallel components. CCA has already been used in several scientific domains, creating components for large simulations involving accelerator design, climate modeling, combustion, and accidental fires and explosions [4]. These types of applications usually contain a large number of connected components, each of them at the granularity of one numerical computation. As individual components are often contributed by separate teams and reused for several applications, many of them are treated as black boxes. Applications based on the CCA model can leverage data provenance to establish whether a particular component behaves properly and to localize bugs and inconsistencies in application development. In addition, we can use the collected data to prune computations that do not have side-effects when there is a match in the input data, by supplying the already recorded output data. This strategy greatly improves performance and it has been employed, outside of the component space, by computational studies, which repeatedly execute the same application to explore its the parameter space [5]. We design a system to collect provenance data in CCA applications with these uses of the data in mind.

In order to collect provenance data, one needs to instrument the application by inserting invocations to a serialization routine. This can be a time-consuming and repetitive task, and a perfect candidate for automation. Automatic instrumentation makes it easy to start collecting data for any CCA application, even if the application consists of many individual component instances. We choose to collect data at the boundary between components, capturing and recording all communication flow between a pair of components, including method invocations and associated input parameters, output parameters and return values. The component boundary is an appropriate place to collect data for the majority of CCA applications we have encountered because the collected data is usually of acceptable granularity to be used together with the component instances in order to enable provenance applications such as accountability, debugging, and the removal of redundant computation (where the computation was previously completed and its input and output data were recorded). It is also the only place where automatic instrumentation is easily attainable. In this paper, we describe our implementation of a method to automatically instrument CCA components in order to collect provenance data. Our goals in designing our system for collecting provenance are to: 1)capture all the provenance data that passes between component instances and 2)impose low overhead so that provenance will be collected continuously, even in deployment scenarios.

We organize the discussion of our provenance gathering system as follows. Section 2 contains background and discussion of the problem. In Section 3 we show a detailed view of our design and implementation of the CCA provenance collection framework, while in Section 4 we discuss some of the preliminary benchmarks we have taken of our system. Finally, we finish with conclusions and future work of the project in Section 5.

2 Background and Problem Specification

Provenance is a technique in wide use in both core computer science and computer application domains. It can be most generally explained as the preservation of metadata regarding a specific data product. Simmhan et al.'s summary of provenance in e-Science [6] establishes a taxonomy of provenance techniques based on the following criteria: 1) application of provenance, 2) subject of provenance, 3) representation of provenance, 4) provenance storage and 5) provenance dissemination. These criteria provide a vehicle for examining a provenance system, considering both provenance collection and provenance querying and use. In our work we only consider the collection of provenance data, while dissemination and use of this data is outside of the scope of this paper.

The CCA model consists of a framework and an expandable set of components. The framework is a workbench for building, connecting and running components. A component is the basic unit of an application. A CCA component consists of one or more ports, and a port is a group of method-call based interfaces. There are two types of ports: **uses** and **provides**. A provides port (or callee) implements its interfaces and waits for other ports to call them. A uses port (or caller) issues method calls that can be fulfilled by a type-compatible provides port on a different component. A CCA port is represented by an interface, which is specified through the Scientific Interface Definition Language (SIDL). A SIDL specification is compiled into glue code that is later compiled into an executable together with the user-provided implementation code. The prevalent way of compiling SIDL is by using the Babel compiler [7]. Babel has the capability of compiling SIDL to bindings for several popular programming languages (C++, Java, Python, Fortran77 and Fortran95), which allows creating applications by combining components written in any of these languages. Babel has a large and growing user community and has become a cornerstone technology of the CCA component model.

Component technology is different in its approach compared to scientific workflows and grids. Scientific workflows and grid services are intended to guide an application from the highest level combining few complex tasks, while component applications are decomposed into many finer-grain tasks. This makes the demands of a provenance system applied to CCA unique, compared to existing approaches in other scientific software domains, in terms of quantities of collected data and common usage scenarios.

Collecting provenance data imposes some overhead on the execution of an application, and if this overhead is high, it may be tempting to turn off provenance collection in order to "squeeze out" more performance from of the machine. One of the principal goals of our system is to keep the overhead low in order to enable continuous provenance collection. In order to accomplish this goal, we need to have the provenance collecting system work in the background throughout the application runtime (as a daemon), and send data to it asynchronously. In addition, scientific applications that are based on CCA often handle large pieces of data; it is not uncommon for an application to have multidimensional arrays that are hundreds of megabytes or even gigabytes in size. When designing prove-

nance collection infrastructure, we need to be careful to avoid in-memory copies of large data, and eliminate needless performance overhead.

3 System Design

We designed and implemented a provenance collection system for CCA applications. Our design provides automatic code instrumentation with provenance calls that incur low application execution overhead. To automate the instrumentation process we need to add provenance collection code to the stub code of each CCA component, and to accomplish this we add functionality to Babel. The inserted provenance collection code will execute before and after every SIDL-defined method, gathering all the input parameters (*in* and *inout*) before the method executes and all the output parameters (*out*, *inout* and method return parameters) post execution. For instance, a SIDL definition of a CCA port containing only one method is given below. To properly collect the provenance data of the component which provides the *IntegratorPort*, we need to capture the two *in* parameters passed to *integrate* before the method executes, and the return value (of type *double*) after the execution completes.

```
interface IntegratorPort extends gov.cca.Port {
  double integrate(in double lowBound, in double upBound);
}
```

The SIDL language is relatively restrictive; it does not allow class variables and forces everything to be expressed in terms of interfaces, classes and methods. Therefore, the data types we are concerned in recording consist of: objects, SIDL-defined base types and arrays of base types. We designed our system to record each of these types, with differing levels of ease.

3.1 Automating Provenance Collection

Once properly modified, the Babel compiler creates special stubs for each method of an instrumented CCA port. These stubs copy the input and output parameters of a method, write them into a message which is later saved to disk as provenance. Since SIDL base types can only be defined as parameters of methods in the language, it is relatively straightforward to generate provenance collection code for each allowable base type (e.g. int, long, float, double, fcomplex, dcomplex, string, bool). On the other hand, this makes the serialization of objects passed as method parameters difficult to automate. Our approach is to force objects that can be serialized to implement a *Serializable* interface. Through this interface, each object must define how it can be written to disk and also reconstructed from its disk image. Our instrumented stubs detect whether an object implements *Serializable*, and will only then invoke the proper serialization methods on the object. Objects that do not know how to serialize themselves are ignored for provenance collection.

SIDL also defines an *opaque* type, typically used to represent a pointer to memory. Babel defines *opaque* as a sequence of bits that are relayed exactly between function caller and callee, which maps to a *void** in C/C++. In order for our system to capture the data to which this pointer points to, we require an additional parameter defining the length of the data. Extra parameters can be defined with SIDL and passed into the Babel compiler using the *%attrib* keyword. We use this mechanism to specify the length of the data pointed to by an *opaque* type in SIDL, and enable our provenance system to record it. For instance, using our integrator example from before, we may have:

```
interface IntegratorPort extends gov.cca.Port {
  void integrate(in opaque %attrib{length=20} data);
}
```

In the scientific domain, arrays are the predominant data type used to represent various data (e.g meshes, fields, matrices, etc.) Unlike *opaque* types, the length of an array does not need to be externally specified because it is part of each array's "struct" in Babel, and is easily obtainable through method calls available in the Babel runtime. Since arrays consist of sequences of SIDL base types, they appear straightforward to handle by our provenance system. However, some arrays in this domain can grow to millions (or more) of elements, so we need to be careful in designing our system to handle them. Below, we propose some prescriptions and optimizations in provenance collecting large CCA arrays.

In order to gather all of the provenance data in one place, as well as provide a single location for querying of such data, which can happen even as it is being written to disk, we need a provenance collection component. Such a component would provide us with a well defined interface and encapsulation for both provenance storage and query. To communicate data asynchronously to the provenance collection component we use an event mechanism. This publish/subscribe asynchronous communication mode allows the provenance component to subscribe to one or more topics in which running components can deposit provenance information. Events decouple the provenance sending from the provenance receiving, writing to disk (or database or other medium) across space and time, enabling a low overhead design. To deliver data to the provenance component, our instrumentation code serializes all the parameters going in and out of a method into a message and publishes it. The provenance component contains a daemon thread that periodically grabs all published messages and writes them to disk. Fig. 1 illustrates an example of this type of communication in our provenance gathering system.

3.2 Optimizations for Large Data

Arrays are often encountered in CCA applications, and some of them can grow to contain a considerable amount of data. CCA components are most often at the granularity of one task, and often a single application passes large arrays to several components. In a shared address space, these arrays are passed by

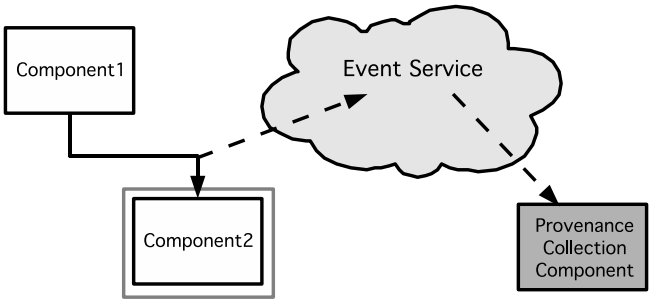

Fig. 1. Displays an example of two application components (Component1 and Component2) and our provenance collection component. When Component1 invokes a method on Component2, our system publishes the method's input parameters using an event message, which is later collected by the provenance collection component.

reference at very little performance cost, so a single method may be invoked many times with a large array as a parameter. Our system is designed to capture all data going in and out of a method invocation, serializing and pushing all this data to disk. In the case of these large arrays, this may be a task that can badly influence the performance and scalability of provenance collection.

One operation we need to avoid is in-memory copies of large arrays. If an array is sufficiently large, an in-memory copy may exceed physical memory size and may lead to OS thrashing. In order to record large arrays properly, we have to circumvent our system's default operation: copying of memory to construct an event message. We add a synchronous way to directly communicate to the provenance component in order to serialize large arrays directly to disk. We need a synchronous call for this operation as we need to ensure that the array is not modified by a component before our system finishes writing it for provenance. We only rely on this mechanism if we detect that an array is too large to copy, leaving unchanged the base system of asynchronously sending data via publish/subscribe for smaller arrays.

Another scenario we want to avoid in the context of large arrays is needlessly serializing the same array that may appear in multiple method invocations (to the same method or a different one). For instance, a visualization component may be invoked every n seconds passing to it the latest data, which may often be unchanged, or a component may simply forward a received array to another component which will do all the work. These are patterns in which our provenance collection system would make several copies of a large data array, costing us application performance and disk space. In order to avoid this scenario we use a very fast hash function to compute a checksum that will enable us to quickly and accurately compare two arrays and determine if they are the same. If two SIDL arrays are in-fact the same array, then their resulting checksums will match and we can avoid the space and time cost of storing one of them. Storing a checksum requires a very small amount of space (usually between 8 and 128 bits), however computing it requires that we perform an operation across the length

of the array. We dismiss the option of computing a checksum using a portion of the array, as it runs a risk of checksum collision. Obtaining the same checksum for different arrays would cause our system not to store an array, which would result in incorrect provenance data and is something we must avoid. Although the cost of computing a checksum may take on the order of seconds for a large array, it is still several fold less than the time needed to store such an array to disk, while also not considering the aforementioned storage space savings.

4 Results

To show the feasibility of our approach, and not as an end in themselves, we performed some experiments. These proof of concept scenarios explored the choices of hash functions for the redundant copy avoidance optimization in large arrays and explored the overhead of our provenance system for a simple application: solving a boundary condition problem. All experiments were performed on a single machine with an 2 GHz Intel Core 2 Duo processor and 1GB of RAM. We made modifications to the Babel compiler provided for automatic insertion of the instrumentation code, and used SCIJump [8], a research CCA component framework that provides low overhead and support for parallel and distributed computing, to conduct our experiment.

4.1 Hash Function Selection

In choosing a hash function to optimize the recording of large arrays we should consider the tradeoff between its performance and the likelihood of collisions in the resulting checksums. Since one of the purposes of this optimization is to reduce overall application overhead, we have to consider the performance cost of hash function computation for a large data array. We cannot afford commonly used hash functions (such as MD5 or SHA) because of the enormous computational load in computing a checksum based on the whole array. While performance is very important, we cannot risk raising the probability of hash function collisions. A strategy we adopt in order to reduce the likelihood of collisions is to make the computed checksum itself larger; a larger checksum reduces the likelihood of collisions by several-fold.

To locate a reasonable and fast hash function, we considered some of the performance benchmarks of hash functions in the Crypto library [9], but concluded that all of the functions in this commonly used library are too expensive. The work of Maxino [10] evaluates and compares the speed and collision potential of very fast hash function for embedded system design, and recommends one's complement addition (also know as the Internet Checksum) as the most reasonable choice in fast hash functions, compared to commonly used XOR, and two's complement addition. To further reduce the likelihood of collision in the checksum, while not greatly increasing computational time, we extended the one's complement addition algorithm's checksum size to 64-bits for our provenance system.

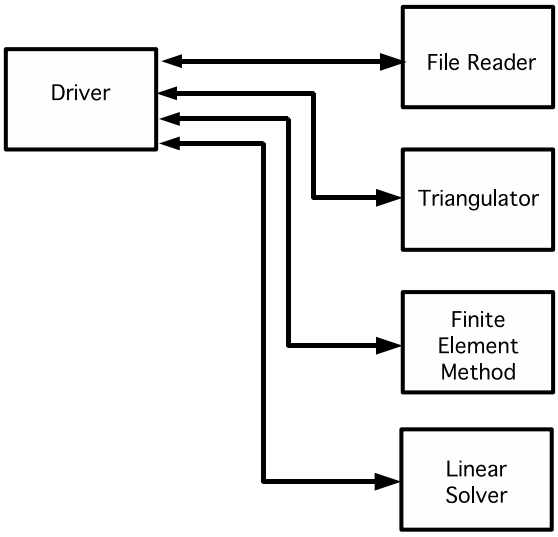

Fig. 2. The heat distribution component application for which we collected data provenance using our system. The Driver component communicates to and from each of the component instances implementing computational tasks.

In this way, the checksum computation produces a small additional time overhead in our system, while providing us with reasonable confidence that checksum collisions will not occur.

4.2 Provenance Overhead in Applications

To validate our design and determine the overall overhead, we implemented our provenance system and used it to collect the provenance of an application. The application we chose solves the heat distribution problem over a small L-shaped 2D domain. It consists of several components, each of which computes a part of the solution (see Fig. 2).

We measured that the provenance collection added less than 1% overhead to the execution time of our application, which is in line with the goals we set forth. We note that this application did not need any of the large data optimizations which we designed, due to the small size of its computational problem. The application did, however, record data containing most SIDL-defined types, resulting in approximately 1KB/method invocation of data logged.

It is possible to design applications where our kind of provenance collection would incur a larger overhead; if the problem is decomposed into a very fine grain resulting in frequent inter-component communication (method calls) that do not spend much time computing. Each method call and its parameters would be recorded by our system, raising the overhead percentage to a higher level than we have encountered here. Although this is possible, it is not the usual way that CCA (or components in general) is applied to a scientific problem.

5 Related Work

The need for data provenance has been widely acknowledged and is evident is many computer domains. Here we intend to only review systems similar to our provenance collection scheme for software components. The provenance survey by Simmhan et al. [6], provides a good overview of provenance applications across different domains. We direct the reader's attention again to this survey for an overview of provenance activities unrelated to scientific software architectures (such as components, workflows, and web services).

The Karma framework [11] collects provenance in the context of web service scientific applications. This provenance service is general and supports different web service systems by using event (publish-subscribe) standards for web services (WS-Eventing). This is the same communication mechanism, in the software component rather than web service domain, that we chose to use in our system. Our work extends Karma in providing an automated approach for provenance collection. Karma further addresses provenance query and visualization, which we do not attempt in our work so far.

Scientific workflows are a software architecture similar to components; workflows usually encompass a wider variety of tasks (such as database data collection, batch job system control etc.) and decompose a problem in a coarser grain than CCA. Altintas et al. present a provenance collection system [12] that requires minimal user effort and a system for "smart" re-runs which mine the provenance information for repetitive data. The provenance data in their system is collected by a special provenance recorder which listens and collects events that are produced by the workflow by default. However, this mechanism does not handle external data automatically, requiring special API calls. External data is probably the majority of the interesting data in workflow applications. Therefore, the provenance collection part of their work would likely require a fair amount of non automatic instrumentation of applications.

6 Conclusions and Future Work

This paper presents a design for automated provenance collection in CCA component applications that requires no user intervention and provides low overhead ($\tilde{1}$% in the application we tested) in order to be useful in deployment scenarios. We designed our provenance collecting system to record data being communicated through each component instance's ports and interfaces. Optimizations were necessary in order to handle large data arrays and the way they are often communicated between component instances. We posit that provenance data collected in this way is usable for debugging, accountability of untrusted component instances, as well as optimizations by removing computations for which we have previously gathered the output.

The future work of this project is to explore techniques of even further reducing the overhead that our system imposes on CCA applications in two ways: 1)by considering novel ways to overlap computation and collection of provenance

data, and 2)by extending our approach to encompass new applications that may have a different computation to communication ratios. Each of these advances in our system is important in achieving broad applicability and acceptance in the CCA and applications communities.

References

1. Guo, Z., Wang, X., Tang, J., Liu, X., Xu, Z., Wu, M., Kaashoek, M.F., Zhang, Z.: R2: An application-level kernel for record and replay. In: Proceedings of the 8th Symposium on Operating Systems Design and Implementation. (2008)
2. Scheidegger, C.E., Vo, H.T., Koop, D., Freire, J., Silva, C.T.: Querying and creating visualizations by analogy. IEEE Transactions on Visualization and Computer Graphics (2007)
3. Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinski, B.: Toward a common component architecture for high-performance scientific computing. In: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation (HPDC). (1999)
4. McInnes, L.C., Allan, B.A., Armstrong, R., Benson, S.J., Bernholdt, D.E., Dahlgren, T.L., Diachin, L.F., Krishnan, M., Kohl, J.A., Larson, J.W., Lefantzi, S., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Zhou, S.: Parallel PDE-based simulations using the Common Component Architecture. In Bruaset, A.M., Tveito, A., eds.: Numerical Solution of PDEs on Parallel Computers. Volume 51 of Lecture Notes in Computational Science and Engineering (LNCSE). Springer-Verlag (2006)
5. Yau, S.M., Damevski, K., Karamcheti, V., Parker, S.G., Zorin, D.: Result reuse in design space exploration: A study in system support for interactive parallel computing. In: Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing, (IPDPS). (2008)
6. Simmhan, Y.L., Plale, B., Gannon, D.: A survey of data provenance in e-science. SIGMOD Record **34**(3) (2005)
7. Kohn, S., Kumfert, G., Painter, J., Ribbens, C.: Divorcing language dependencies from a scientific software library. In: Proceedings of the 10th SIAM Conference on Parallel Processing, Portsmouth, VA (2001)
8. Zhang, K., Damevski, K., Venkatachalapathy, V., Parker, S.: SCIRun2: A CCA framework for high performance computing. In: Proceedings of The 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments. (2004)
9. Dai, W.: Crypto++ 5.5 Benchmarks – http://www.cryptopp.com/benchmarks.html (2008)
10. Maxino, T.C.: The effectiveness of checksums for embedded networks. Master's thesis, Carnegie Mellon University (2006)
11. Simmhan, Y.L., Plale, B., Gannon, D.: A framework for collecting provenance in data-centric scientific workflows. In: International Conference on Web Services. (2006)
12. Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance collection support in the kepler scientific workflow system. In: First Internation Provenance and Annotation Workshop. (2006)