An Overview of the Common Component Architecture

Rob Armstrong and Teresa H. Ko
Sandia National Laboratory
7011 East Avenue
Livermore, CA 94551
(925) 294–2470, (925) 294–4829
{rob,thko}@sandia.gov

David E. Bernholdt
Oak Ridge National Laboratory
P. O. Box 2008, MS 6367
Oak Ridge, TN 37831-6367
(865) 574–3147
bernholdtde@ornl.gov

As the commercial software industry burgeoned, it was clear that increasingly complex software would require a mechanism scaling across people, geography and time. The answer came in the concept of software components. Software components are stand-alone modules that have a prescribed means for composition into an application. Component concepts enable, for example, MS Word documents to appear in MS Powerpoint slides, and has led to the point-and-click user interfaces that inhabit most desktop computers today. The idea of a component in software comes from its root word: "composeable". The process of connecting components together into an application can be likened to their electrical component analogue: hook transistors, diodes and resistors together one way, and you have a radio, another way and you have an MP3 player. The Common Component Architecture is a component model created by computational scientists from all of the DOE laboratories to establish a "plug and play" standard for high-performance computing. Recently the Common Component Architecture has been named on the Top 10 DOE Science Achievements in 2002 list (http://www.sc.doe.gov/sub/accomplishments/top_10.htm).

Though computing has been synonymous with the DOE labs long before anyone dreamed of having a computer on their desktop, scientific computing – high-performance scientific computing in particular – has not benefitted from these advancements. This is because parallel computing, the mainstay of high-performance computing, is not amenable to the component software existing in the commercial world. Parallel software requires a model that enables cooperation among thousands of individual processors, a situation not familiar to commercial software vendors. The Common Component Architecture was conceived to fill this gap. Beginning in 1998, the CCA Working Group grew from a grass roots effort to develop an architecture for high-performance component computing.

Currently, almost all parallel programming efforts involve a closely knit group of scientists working together to produce a single code. This means that code and algorithms common to most parallel programs has to be reinvented for each application. Also, in the last 5 years, various DOE programs have desired to sew together parallel simulations of subsystems, into larger simulations. The CCA component specification enables code-sharing among disparate groups by defining a few simple rules that, when followed, allow plug and play compatibility, with minimum impact on performance. In addition, the structure of the CCA is such that each application composed of CCA compliant components will be itself a component. This means that every application that conforms to the specification is automatically compatible with any other.

Computational scientists all are certain what the "right" programming language is; unfortunately it differs depending on who you talk to. Scientists will use Python, C, Fortran and are no less certain that their language is the best and are unwilling to give up their personal favorite. In the commercial software industry, the software component model usually dictates the language in which the programmer must write (MS COM requires the Visual C++ virtual table layout, Java Beans requires Java). Early on, the CCA working group realized that trying to dictate language preference to scientists was futile. This is why the CCA separates the abstract mechanism of connecting components from the underlying language binding. This accomplishes two goals: first the scientist can use whatever language she/he wishes, and second the performance of the resulting application will be as fast as if it were written without components. This concept however creates another problem to be solved, however. Components written in one language cannot work with components written in another. For this reason the CCA language specification is written in the Scientific Interface Def-

inition Language (SIDL) and, for only a slight extra overhead, components written in C, C++, Fortran, and Python can all interoperate.

The CCA model for high-performance computing provides a mechanism that sews together modules (i.e. components) into an application and then gets out of the way to allow the components to do their work unimpeded. The architecture only specifies a way of connecting one component to another, so that function calls are made directly without mediation. The user of a component pays only the performance price of the language in which (s)he is writing and nothing *per se* for using components. After the composition phase assembling components into a running application, no penalty is incurred for using CCA. For Grid-style, loosely coupled computing, CCA is flexible enough to allow proxies to be automatically generated to mediate connections for distributed object components using the same design pattern and architecture. Even though it is unlikely that a distributed object component designed for the Grid would be used in a high-performance setting the user does not have to learn a new set of use patterns to move over to the Grid arena.

The advantage of CCA is that it does not introduce anything that will potentially slow a high-performance application. The disadvantage is that it does not provide anything beyond a way to link together an application with components. Components must still use MPI or some other message-passing layer to accomplish their parallel algorithms. CCA relies almost entirely on components to provide value-added to applications. No parallel data model is assumed or provided by CCA for users and must be provided by components. There are a number of components already developed and in various stages of completeness including MxN data transfer component, a structured data component, an unstructured data component, equation solvers, and others. These components provide a basic toolkit to application developers.

There are a multitude of efforts developing tools for the CCA specification and writing applications and components that use it. The Dept. of Energy SciDAC Center for Component Technology for Terascale Simulation is devoted solely to the development and promulgation of CCA tools and technology. There are numerous educational materials located at the main CCA web site: `http://www.cca-forum.org`.

One application of particular interest for the embedded domain that is being explored in the CCA context is the rapid parallelization of computer vision applications. Effective use of computer vision research in embedded environments such as robotics and wireless sensor networks has been hindered by the trade off between robust accurate results versus real-time information extraction. Yet, sight is our most intuitive and natural sense and essential to achieving the goals of these systems. Providing high performance embedding computing in these constrained environments can increase the robustness, accuracy, and timeliness of the current state of the art computer vision algorithms. A detailed look at one specific algorithm developed for scene reconstruction illustrates the many avenues for parallelization: dedicated frame grabbers from multiple cameras, independent feature extraction on different images, pairwise feature correlation between images, etc.