AN EXTENSIBLE PARAMETER VIEWER FOR CUMULVS

Dan Bennett, Paul Farrell, and Arden Ruttan
Edinboro University of Pennsylvania, Kent State University
dbennett@edinboro.edu, {farrell, ruttan}@cs.kent.edu

ABSTRACT
In this paper we present a novel viewer for use with the CUMULVS steering and visualization (STV) middleware for parallel computation. Designed to permit a computational scientist to observe or ”view” the data from a running computation, viewers form an integral part of any STV system. The extensible viewer provides a simple event based interface upon which a scientist can, with minimal effort, build customized viewers to meet the needs of an individual application. We describe the design, implementation and present several example viewers produced using this framework.

1. Overview

Our overriding research is to attempt to steer a distributed computation simulating a defect in a liquid crystal display through a bifurcation point. In this effort, we have employed CUMULVS, a steering and visualization middleware package which provides easy access to both observe and change data within the distributed computation. While CUMULVS provides a powerful array of tools for viewing distributed data, we have noticed that it lacks a complimentary set of tools for dealing with scalar data. Out of necessity, we have begun to build such a set of tools.

While we feel that our tools are comprehensive, we have come to realize that no such set of tools can be complete. There will always be a desire on the part of the computational scientist to create a custom tool for their particular application. So, to simplify the creation of such customized tools, we have created an extensible viewer. This viewer is built upon a utility library which supplies python bindings to CUMULVS. The viewer is event based and can be easily employed to produce both text and GUI based applications.

2. Background

2.1 Parallel Computing

The world of parallel computation is diverse. It ranges from instruction level parallelism within a processor to large clusters of multicore machines connected with proprietary networks. Our work focuses on cluster-based computing which is the current dominate architecture in computational sciences [1]. Furthermore, our interests are in in the SPMD or single program multiple data programming model. In this model all nodes in a computation run the same program but have access to different data.

Most large computer clusters are operated in a batch environment to promote optimal use of computing resources. Applications developed to run in these environments must therefore run without user interaction. All I/O must be directed tofiles, and data analysis is limited to pre-and postprocessing. In such computations the scientist has been “removed from the loop”. Steering and visualization middle-ware attempts to address this issue.

2.2 CUMULVS

Steering and visualization middleware, such as CUMULVS[2] provide a mechanism for interacting with batch style parallel computations. A computational scientist can instrument existing parallel application by inserting a small number of library calls into the simulation. This instrumentation allows CUMULVS to extract data from the running simulation and make it available for display by CUMULVS enabled clients called viewers. In addition, CUMULVS provides a mechanism by which viewers can change a value within the parallel application. To accomplish this task, CUMULVS relies on the SPMD parallel programming paradigm.

CUMULVS is able to extract two different types of data, the first being a scalar value, This scalar value is constant across all nodes in a computation, is called a parameter. Examples of a parameter include a loop control variable or a predefined tolerance value. CUMULVS is able to transfer parameter data both to and from the computation. The second type of data that cumulvs can handle is called field data. This is data which has been distributed over the nodes of the computation. This could be a section of a parallel matrix, or a portion of the molecules in a

molecular dynamics simulation. Each node in the computation contains a unique portion of the entire data set. CUMULVS is responsible for extracting, synchronizing, assembling and delivering any field data requested to an end viewer.

Viewers are programs which connect to an application via CUMULVS and request parameter values and field data. Viewers can provide visualization of data, or return new parameter values to the computation via CUMULVS. Returning such parameters is known as steering, and provides the computational scientist the ability to interact with, or steer the running computation. This has the effect of putting the scientist "back in the loop" from which they have been removed by the batch environment. Chin [3] provides a thorough discussion of the advantages of employing a steering and visualization system when performing research with a distributed scientific computation.

3. Viewers

3.1 Classes of Viewers

Viewers form one of the three major components needed to perform interactive steering on distributed computations. The other two components are the middleware system, which is a generic transport mechanism and the application, which by definition, is highly application dependent. Many tasks involved with steering an application are common to most applications, and as such can employ generic viewers. For example it is beneficial in many applications, to view a plot of a dependent and independent variable. Such a graph is illustrated in Figure 1. We have proposed [4], and are in the process of building a set of such universally applicable viewers.

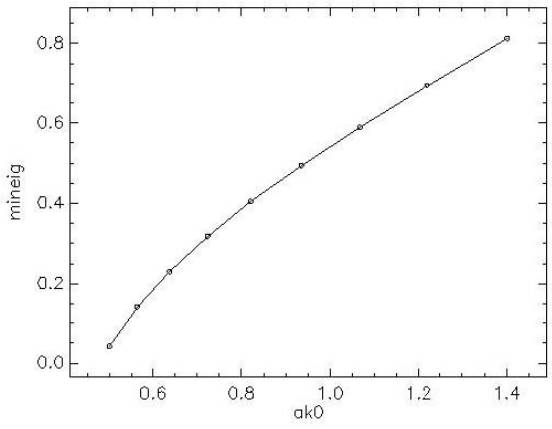


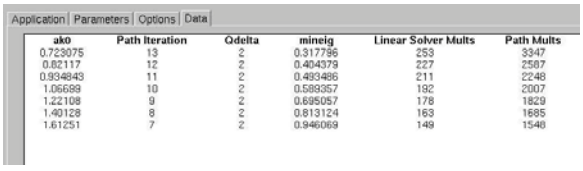
Figure 1: An x-y Plot Viewer

Viewers can be divided into two major classifications. Field viewers are designed to display field data, and as such tend to be highly graphical in nature and computationally intensive. Most STV packages, including CUMULVS tend to focus on such viewers and include access to a number of generic viewers of this nature. Pickels states that while such a system "... makes use of existing, powerful visualization tools, it does tie the user to them and also to a machine capable of running them." [5] CUMULVS, for example provides several custom field viewers as well as an interface to both the Visualization Tool Kit (VTK) and the commercial visualization package AVS Express.

A second class of viewers, called parameter viewers, is the set of viewers designed to display and manipulate parameters. This class of viewer is much more lightweight, as they are designed to deal with far less data. The simple x-y plot pictured in Figure 1 for example, will receive two data points per update, which will be displayed as part of a polyline containing perhaps 100 points. CUMULVS provides only simple text-based parameter viewers. While these viewers are illustrative examples of how to program such a viewer, they are somewhat cumbersome to use. In a previous work [6] we have shown that parameter viewers have minimal impact on the performance of a parallel application. Given the insight such viewers provide into the operation of an application, we believe that a robust set of parameter viewers is critical in the steering and visualization process, and is therefore a critical portion of any STV package.

3.2 Parameter Viewers

During our experimentation with CUMULVS, we have produced a number of simple parameter viewers. We have employed an x-y plot viewer to track the minimum eigenvalue of a representative matrix as the temperature (ak0) of the system changes. The viewer provides a quick graphical representation of the relationship between these values and indicates that the computation is approaching a bifurcation point. Figure 2 is a history viewer, which displays the most recent values of a selected set of parameters. Other viewers have been produced which perform tasks such as pause, single step and halt a computation, observe and change single parameter values, and provide an interface to user-level checkpointing.



ak0	Path Iteration	Qdelta	mineig	Linear Solver Mults	Path Mults
0.723075	13	2	0.317796	253	3347
0.82117	12	2	0.404379	227	2587
0.934843	11	2	0.493486	211	2248
1.06699	10	2	0.589357	192	2007
1.22108	9	2	0.695057	178	1829
1.40128	8	2	0.813124	163	1685
1.61251	7	2	0.946069	149	1546

Figure 2: A History Viewer

While producing these viewers, it was noted that the CUMULVS interface provided many lower level calls, which were always grouped together into higher level activities.

For example, connecting a viewer to an application requires a set of calls, which collectively can be grouped into a single logical action, connect. Abstracting these activities into an utility library, much like GLUT for OpenGL or Xt for the X window system, provides a cleaner interface for producing viewers which in turn simplifies this task. A prototype of this utility library has been implemented in C++ and bindings for python have been produced. The utility library has proved highly valuable in providing the ability to rapidly produce additional viewers.

Python was selected as a second development language for a number of reasons. The principle reason is python's ability to deal with arbitrary typed data. CUMULVS provides data as a memory reference and an associated type field, which is easily converted to a python object, that also carries an associated type value. Other features motivating the selection of python include the large number of libraries and utilities the language provides, the ability to perform rapid prototyping and application development, and a level of acceptance of python by the scientific computing community[7].

4. An Extensible Viewer

As our work steering a parallel computation progressed, we discovered that no matter what viewer is available, there is always a desire for something different. The xy-plot viewer is useful for viewing data with a direct relationship, but what if the relationship is not direct, or involves multiple parameters. For example, when employing a simulation of a large lake, the researcher may wish to plot the population of fish per square kilometer per unit time, as opposed to the total population per unit time. Furthermore we discovered that isolated tools are difficult to manage, and that better interfaces result from the ability to combine simple tools. To address these issues, and other concerns, an extensible viewer was developed.

4.1 Goals and Design Objectives

The main goal of the extensible viewer is to simplify the development of future viewers. In an ideal world a computational scientist will have a scientific visualization expert at their disposal, along with sufficient time to devote to the development of not only their scientific simulation but to the tools that accompany such a simulation. In the absence of this ideal condition, tools should be constructed which are, by default, powerful and straightforward to use, and easy to modify into a more desirable form.

A second goal is to maintain flexibility, allowing the tool to be used for as many applications as possible. This is accomplished by maintaining data flexibility. A statical typed language, such as C requires that either data types be defined at compile time, or the user must constantly deal with void pointers and type casts throughout the program. Python, with its dynamic typing allows data type decisions to be handled when they become important. Thus intermediate routines can pass data through, without concern for type, while final routines can convert data to the format required for display or computation.

4.2 Architecture and Implementation

Our initial designs included embedding an interpreter into an existing viewer. This would allow the user to modify and combine parameters used by the viewer by means of a simple program. The idea was eventually discarded as it was not sufficiently simple. This paradigm involved learning the language supported by the interpreter as well as the interface between C++ and the interpreter.

The nature of CUMULVS' interaction dictates that most viewers will be event-driven. CUMULVS collects all parameters and fields from an application and delivers this information to a viewer as a frame. The viewer must periodically query CUMULVS to see if new data has arrived. When such data does arrive, the viewer must extract pertinent information and inform CUMULVS that it is free do assemble and deliver the next frame.

After several prototypes were produced, an architecture based upon the reactor design pattern was selected. This design pattern consists of a reactor, which receives and dispatches events, and event handlers, which are called by the dispatcher when events occur[8]. The design pattern is a formalized method for implementing event-driven programming. The extensible viewer is the reactor, which provides the interface to the application through CUMULVS. The user is responsible for providing callbacks, which perform the desired operation on the parameters.

The reactor has been implemented and is responsible for all aspects of connecting with, receiving parameters from, and returning values to CUMULVS. Supplied with the application name the reactor will perform all tasks associated with connecting to the application. The reactor also supplies a "main loop" which checks for the arrival of a new frame of data. When a new frame is received, the reactor calls the appropriate callback functions and transfers data to the user-supplied callbacks. Currently the reactor supports two types of callbacks. Idle callbacks are called every iteration of the main reactor loop. An idle callback allows for rutine maintence tasks, such as GUI updates to occur within the user portion of the application. A frame update callback is called upon the arrival of a

new data frame. These callbacks are the main mechinism for user processing of parameter values. Each of these callbacks will be illustrated in the next section.

The user must declare and register callbacks. A callback will receive the name of the application which generated the call, a list of parameters and values requested from the call and a list of application data which the user has registered. An example of these callbacks can be seen in Figure 1. Note that the application name, as well as the parameter names requested are strings and match the corresponding CUMULVS strings for the application. These are the only application specific part of the program. This viewer will be fully discussed in the next section.

4.3 Example Viewers

In this section we will examine, in detail, two viewers. A simple text based viewer presented in Figure 3 with output in Figure 4, and a GUI viewer based on the Fast Light Tool Kit (FLTK) presented in Figures 5 and 6. Both of these viewers are built on top of the extensible viewer. The simple viewer is presented in its entirety, while only code fragments from the GUI viewer are presented.

The simple viewer demonstrates the ease with which a viewer can be built. This viewer requests a list of parameters from a single application. Each time a frame is received, the name of the application as well as the values of the individual parameters are displayed. In Figure 4

```
1 # ! /usr/bin/python
2 import exServer
3
4 def  call1 ( app , paramList , myList = ( ) ) :
5    print " ************************** "
6    print app , " returned to call1 "
7    for i in paramList:
8      print " Param: " , i [0] , "=" , i [1] [0]
9    print " ************************** "
10
11 app = "LIFE"
12 plst =( "Average Age " , " Generation " )
13 exServer.addCall ( app , plst , call1 , ( ) )
14
15 exServer.mainLoop ( )
```

Figure 3: A Simple Viewer

```
**************************
LIFE returned values to call1
Parameter : Average Age = 34.4264
Parameter : Generation = 297
**************************
**************************
LIFE returned values to call1
Parameter : Average Age = 34.592
Parameter : Generation = 298
**************************
```

Figure 4: Output of the Simple Viewer

this viewer is attached to a parallel simulation of Conway's Game of Life and the display shows the average cell age and the generation number for two iterations of the simulation.

Any instance of the extensible viewer must import the reactor (line 2). Callbacks are added to the reactor in line 13. These parameters include the application name (line 11), the list of parameters the callback wishes to receive (line 12), the name of the callback, and a list of user data the programmer wishes to pass to the callback routine. In this case, the user data is an empty list as no user data transfer is requested. The application name as well as the parameter names are determined during the instrumentation process, and are mapped directly to variables within the application. These values can be interactively obtained from the user of the viewer, but are hardcoded here for simplicity and brevity.

The callback is declared to receive the application name, parameter list and the user data list. The parameter list contains, for each parameter, the parameter name and a list of parameter values. CUMULVS supports a vector of parameters hence the parameter value consists of an additional parameter name and the actual parameter value. This is illustrated in line 8, which will print the name and value of a simple parameter, or the first parameter in a parameter vector.

```
 1 def pauseCall(ptr ):
 2         exServer . SetSteps (0)
 3         pauseButton . deactivate ()
 4         continueButton . activate ()
 5         oneStepButton.activate()
 6
 7 def stopCall(ptr ):
 8         exServer.SteerParam("LIFE", "Quit",1)
 9
10 def idleCall(myList =() ):
11         Fl . check ()
12
13 def call1(app, paramList , myList=()):
14         myList[0].label(paramList[0][1][0])
15
16 ...
17 # main routine
18 #       note text has been wrapped to
19 #       fit format requirements .
20 app = "LIFE "
21 srv.addCall(app,("Cells Alive") ,
22         call1 ,( cellsAliveData ,))
23 srv.addCall(app,("Average Age") ,
24         call1 , (AverageAgeData ,))
25
26 srv.addIdleCall(idleCall , ())
27
28 #  tell FLTK to display the main window
29 window.show(len(sys.argv), sys.argv)
30
31 #  call the reactor main loop
32 srv.mainLoop ()
```

Figure 5: Partial Code for a GUI Viewer

The GUI viewer presented in Figure 5 is a fragment of the code required to produce the interface in Figure 6. Nearly all of the routines to build and maintain the GUI have been removed from this listing, as have some of the code which performs interaction with the reactor. The pauseCall in lines 1-5 is illustrative of both the interaction with the reactor and maintenance of the GUI. Line 2 illustrates how an application can request that the reactor stop receiving updates from CUMULVS. CUMULVS maintains synchronization with an attached viewer, so this has the effect of pausing the computation.

The stopCall, lines 7-9 illustrate the ability of the extensible viewer to provide simple steering. In this routine, a parameter change is requested. Specifically, the program is requesting that the parameter Quit be set to the value 1. In this particular application, this will have the effect of causing the application to exit. It should be noted that this portion of the code is scheduled to be updated and will be replaced by a list based interface, more consistent with the parameter callbacks in the next version.

Lines 10 and 11 contain an example of the use of an idle callback. The idle callback, implemented as idleCall, is responsible for allowing the FLTK reactor to execute a single iteration of the event loop associated with the GUI. This routine is registered with the extensible viewer reactor in line 26. The code illustrates the extensible viewer's ability to interact with other event based systems.

The callback in lines 13 and 14 demonstrate user data passed to the callback. In this case, different FLTK box widgets are passed to the routine for different parameters (see lines 21 through 24). In line 14, the value in the box, thus the value displayed in the GUI is set. Note that call1 is used multiple times in this application.

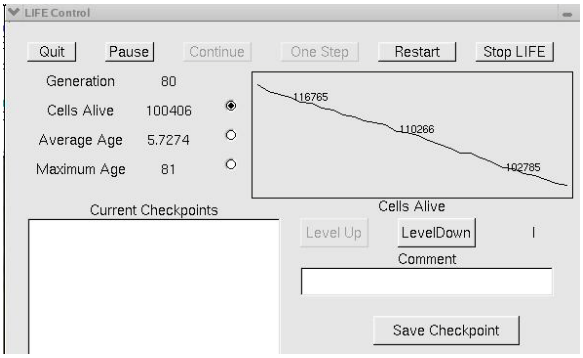


Figure 6: A GUI Based Viewer

These viewers are only two examples of the viewers produced based on the extensible viewer. We have produced a number of these viewers and are convinced of the usefulness of the extensible viewer. We are currently in the process of refactoring earlier viewers so that they can be incorporated into this framework. By removing CUMULVS interaction, especially for the reactor main loop, and providing routines to act as callbacks, these viewers can become building blocks for the construction of more complex tools employing the extensible viewer.

4.4 Limitations and Future Work

The extensible viewer presented is a working prototype and as such has a number of limitations. The viewer is currently limited to interaction with a single application. This limitation can be easily addressed within the reactor by maintaining a list of applications and associated callbacks.

The viewer also lacks the ability to deal with field data. As indicated previously, we believe that there are a sufficient number of field viewers, and as such placed a low priority on exploring this avenue. Recent requests, espe-

cially for the ability to combine afield visualization wi th parameter visualization and control, have prompted exploration of these possibilities. We are currently engaged in activities to addfield handling to the python interface to CUMULVS, which will quickly allow for inclusion in the extensible viewer. We envision this extension will be via afieldcallback registration function, which will cause the reactor to pass data to the appropriate callback. We plan to employ the python numerical array package, numpy, for this implementation.

Other limitations include a shortage of documentation and a lack of properly documented example programs. These shortfalls will be addressed as the package is prepared for distribution.

5. Conclusions

In this paper we have presented an extensible viewer for CUMULVS. Written in python the extensible viewer can be used in the creation of both text and GUI based viewers. Based on our python interface to a CUMULVS utility toolkit library, this viewer removes nearly all of the complexity in dealing with the CUMULVS interface to parallel computations. We believe that this simplfied interface will allow computational scientists to more rapidly develop custom parameter viewing and control systems for their individual computations.

Based on this work we have begun to develop a graphical tool, much like the interface to OpenDx, which will allow a computational scientist to create, on thefly, a front end for applications which have been instrumented with CUMULVS. Early prototypes of this tool have shown much promise.

While a full release of the python CUMULVS utility toolkit library and the extensible viewer has not yet been made, the authors will happily provide alpha versions of this work. A full release is expected within the next year.

6. Acknowledgments

The authors would like to thank the Faculty Professional Development Council of the Pennsylvania State System of Higher Education for its support through a faculty professional development grant, which has funded, in part, this work.

References:

[1] H. Meuer, E. Strhmaier, J. Dongarra, & H. Simon, Top 500 list. *World Wide Web electronic publication*, 2008, retreived Feb, 2009, http:www.top500.org.

[2] J. A. Kohl, T. Wilde, & D. E. Bernholdt, Cumulvs: Interacting with high-performance scientific simulations, for visualization, steering and fault tolerance, *Int. J. High Perform. Comput. Appl., 20(2)*, 2006, 255–285.

[3] J. Chin, J. Harting, S. Jha, P. V. Coveney, A. R. Porter, & S. M. Pickles, Steering in computational science: mesoscale modelling and simulation, *Contemporary Physics, 44(5)*, 2003, 417-434. 2003.

[4] C. Stein, D. Bennett, P. A. Farrell, & A. Ruttan, A steering and visualization toolkit for distributed applications, Proc. The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, 2006,451–457.

[5] S. M. Pickles, R. Haines, R. L. Pinning, & A. R. Porter, Practical tools for computational steering, *Proceedings of the UK e-Science All Hands Meeting 2004*, East Midlands, UK, 2004.

[6] D. Bennet & P. Farrell, Experiences instrumenting a distributed molecular dynamics program, *Proceedings of the 21st Annual Spring PACISE Conference*,Indiana, PA, 2006.

[7] T. E. Oliphant, Python for scientfic computing, *Computing in Science and Engineering*, *9(3)*, 2007, 10-20.

[8] D. C. Schmidt & J. O. Coplien, *Pattern languages of program design* (New York, NY: ACM Press, 1995)