Porting of VisIt Parallel Visualization Tool to the Cray XT3

Kevin Thomas, *Cray Inc.*

ABSTRACT: *VisIt, the popular visualization tool developed by the U.S. Department of Energy, was recently ported to run on the Cray XT3. While many applications are easily ported to the Cray XT3, VisIt, with its extensible and distributed architecture, presented a unique challenge. The Catamount OS lacks support for critical features used by VisIt, but the visualization of large datasets requires parallel computation. Through a variety of strategies, the port was completed without requiring source code changes to the application.*

KEYWORDS: Cray XT3, Catamount, Porting, Visualization

Introduction

VisIt, a visualization tool developed by the U.S. Department of Energy, is required by the U.K. Atomic Weapons Establishment (AWE) for use on its Cray XT3 system. While visualization is not a well-established application area for the Cray XT3, it is understandable that AWE should want to use VisIt, as the ability to perform simulation processing and visualization on a single computer system simplifies the research workflow. Otherwise, two systems are needed, and large datasets need to be migrated between them.

The primary goal for the VisIt porting project was to meet AWE's performance goals, measured by a set of benchmarks developed for the system procurement process. Full support for all standard VisIt capabilities, plus minimal source code changes to the application, were to be accommodated. Likewise, it was important that VisIt could be easily upgraded to new versions as they were released by its development team at Lawrence Livermore National Laboratory – changes needed to be small and localized so that they could be integrated into the standard source release.

VisIt version 1.4.1 was used for this port.

Application Architecture

VisIt is composed of several components, each of which is built as a separate program. These include:

- the graphical user interface (**gui**)
- the command-line interface (**cli**) with Python scripting
- the **metadata server** for managing datasets
- the **viewer** for 3D visualization
- the **compute engine** for manipulating datasets

These components intercommunicate via TCP/IP sockets that allow them to be distributed across a network. For example, the gui and viewer might run on a workstation, while the metadata server and compute engine run on a remote server. The cli is useful for batch processing, where a large dataset might be processed to generate a set of images, or for scripting repetitive operations.

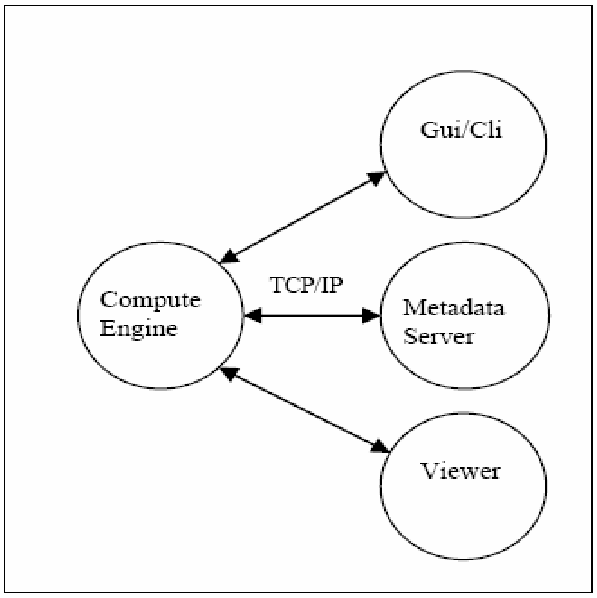

Figure 1: VisIt Components

While these components are available as single process programs, the compute engine also has a parallel version, implemented with MPI. It is functionally equivalent to the serial version; but it can process much larger datasets, and it uses parallel algorithms to reduce processing time.

End users can extend VisIt in several different ways. Three classes of extensions are implemented through plugins, which are shared libraries loaded at run time:

- **database plugins** read grid structure and property value data.
- **operator plugins** act on this data to create new results (for example, slice, clip, project).
- **plot plugins** generate visualization objects (for example, contour, pseudo-color, scatter).

VisIt comes with a wide range of standard plugins. In addition to these, AWE uses custom database plugins to read local data file formats.

Porting Strategies Considered

The VisIt architecture and operating system requirements were evaluated to determine the best approach to port it to the Cray XT3 system. While Cray XT3 login nodes run a standard Linux distribution, the Catamount OS that runs on the compute nodes does not support sockets or shared libraries. Because of this, it was desirable to use login nodes to run components that did not require large-scale compute resources. As VisIt had already been ported to Linux, no additional effort was required for these components.

Porting the Parallel Compute Engine

The parallel compute engine was the only component that did not fit into the initial porting strategy. Three approaches, described in the following sections, were considered to port it.

Use Linux-based login nodes

One approach to port the parallel compute engine was to build it for Linux, then run it on login nodes using an MPI library built to use TCP/IP sockets. This approach was rejected early on because it did not meet goals for performance and scalability: Cray XT3 systems contain only a few login nodes, and it had been determined that dozens of processors would be required to meet the AWE performance goals.

Use mixed node-type execution

The second approach was to use an untested feature of the software architecture, developed for the Sandia Red Storm project – the Cray XT3 system prototype – that allows a parallel program to run on a mix of Linux and Catamount nodes. This feature had the potential to remove the requirement to support socket communication on Catamount, because only MPI process rank 0 of the parallel compute engine communicates with the other VisIt components via sockets. A parallel compute engine runs with process rank 0 on the login node and the remaining ranks on compute nodes might provide a solution. Some changes to the engine source would be required to isolate the heavy computation within the parallel engine from the rank 0 process and therefore offload the login node.

This approach was eventually abandoned when further investigation revealed that the Cray XT3 system software did not allow parallel jobs with mixed node types. It was also incompatible with Catamount Virtual Node (CVN), the OS feature that provides support for dual-core Opteron processors.

Use Catamount-based compute nodes

The third strategy was to run the full parallel compute engine on compute nodes. This required resolving some missing capabilities within Catamount, discussed in the next section.

Porting Issues and Solutions

Since Cray XT3 compute nodes running the Catamount OS do not support sockets or shared libraries, the porting effort concentrated on developing solutions to compensate for these missing features. There were two additional, more subtle issues to be resolved, described in the following sections.

Building statically-linked executables

The default build procedure for VisIt and its support libraries generates a large number of shared libraries. Linking with these libraries creates dynamically-linked executables – but the Catamount OS requires that executables be statically linked. So, with the exception of the plugin shared libraries, the build procedure could be adjusted to generate all static libraries, primarily through changes to settings for the GNU `autoconf` configure scripts that are used to set up the Makefiles before the software is compiled. This change is not desirable for Linux-based components, so separate software builds are required for Linux and Catamount respectively, even though each component targets one OS or the other.

X Windows libraries not available on Catamount

The second issue arose because all VisIt components share a set of common libraries. This is partly required to avoid redundant implementation and partly a convenience of software organization. The result is that each component executable has a large static call tree of routines, many of which are never called. This is largely innocuous, especially when shared libraries are used in a demand-paged, virtual memory environment like Linux. In the case of the parallel compute engine for Catamount, this led to the requirement that X Windows libraries be available.

Because the X Windows libraries are standard on Linux, it is allowed to link these libraries into a Catamount program as long as the routines are not called during run time. This is possible because Catamount and Linux share the same base programming environment, so the libraries are passively compatible, and warnings printed by the linker for these libraries can be safely ignored. Of course any attempt to call these routines from a Catamount program would fail.

Plugin Support

The VisIt plugin infrastructure accesses plugins via the `dlopen`, `dlsym`, and `dlclose` system calls. With these calls, it is possible to access external symbols (routines or data structures) within a shared library that is not specified at program link time – a form of dynamic linking. The plugin can be developed after the program is developed so long as it adheres to the interface requirements of the program. In VisIt, plugins have a general interface plus component-specific interfaces. These interfaces are implemented as external data structures in the shared libraries, and have fixed, well-known names.

First Approach

The first approach to support plugins was to implement `dlopen`, `dlsym`, and `dlclose` as utility library calls. Here, `dlopen` allocates a region of memory on the heap and reads the shared library into this region. `dlsym` uses the ELF[1] library primitives to access the symbol table of the shared library, just as the Linux linker does. While this approach works fine for leaf routines, more extensive support for dynamic linking proved too ambitious for this project.

Second Approach

A second approach to support plugins was to statically link the required plugins into the parallel compute engine. Some flexibility was lost with this approach, since all plugins must be available when the parallel compute engine is linked. This limitation was not too burdensome because new plugins are not created frequently. Code size grew with this approach, but the overall application size only increased moderately.

[1] ELF (Executable and Linking Format) object files can be manipulated using a standard system library libelf.

Building the plugin libraries

The first step was to build static plugin libraries. This was accomplished in the same way as with other libraries, by using specially-crafted configure parameters. Some minor hand-editing of the generated Makefiles was required to avoid warning messages.

Renaming the plugin interface symbols

Since the plugin libraries have the same external symbol names (in conformance with the VisIt plugin requirements), the libraries cannot be directly linked into a single program. Also, the components do not contain explicit references to the libraries or their symbols, because these are determined at run time. Thus, directly linking plugin libraries into a component is not possible.

To work around this problem, two utilities were created. The first, `bldplug`, uses features of the system linker, GNU `ld`, to convert the static plugin libraries into object files, and at the same time adds a unique prefix onto each external symbol. With these plugin object files it is possible to link all plugins into a single program without conflict.

For example, there are database plugins for PLOT3D and Silo. Both shared libraries contain the `GetGeneralInfo` external symbol used to access the plugin data structure. `bldplug` renames these as follows to make them unique.

Plugin	**Original Symbol**	**New Symbol**
`PLOT3D`	`GetGeneralInfo`	`PLOT3DGetGeneralInfo`
`Silo`	`GetGeneralInfo`	`SiloGetGeneralInfo`

Adding the standard interface wrapper

The second utility, `gendl`, takes a set of plugin object files and adds special-purpose versions of the standard `dlopen`, `dlsym`, and `dlclose` routines. The new `dlopen` examines a table of available plugins statically linked into the program, and returns status to the caller indicating whether the requested plugin is available or not. The `dlsym` returns the requested plugin external symbol via a similar table lookup, or it returns an error if no such symbol exists. The `dclose` routine frees up the table memory allocated in `dlopen`. The lookup table for `dlopen` is created by `gendl`.

To extend the Silo plugin example, a call to `dlopen` to access the Silo plugin causes the `dlopen` routine to check its internal table to determine if the Silo database plugin was available. If it is, a later call is made to `dlsym` to access the `GetGeneralInfo` symbol from this library. The replacement `dlsym` looks up the name

`SiloDatabaseGetGeneralInfo` and returns a pointer to this data structure to the caller.

Handling external symbol name collisions

Although the well-known plugin interface symbols are renamed to unique identifiers by the `bldplugin` utility, libraries may, in general, define other external symbols. Thus, it is possible that two libraries define symbols with the same name. This presents no difficulty with regular dynamic linking, since each library is managed as an independent symbol name space. But when static linking is used, duplicate symbol names cause a link error.

In practice, name space collisions between plugin library external symbols are rare. One reason is that the coding conventions used by the VisIt developers avoid giving two routines the same name. Outside of a plugin, symbols with duplicate names create linker errors as well.

There is one case within the standard VisIt plugins that a name collision occurs. An investigation revealed that two items were copied from a library and into a plugin. This code extract was fairly small, so it may have been easier to copy it into the plugin rather than change the plugin build to include the extra library.

Once the parallel compute engine (which includes the library) was statically linked with the plugin library containing the duplicated source code, a duplicate symbol error occurred. There are many ways to work around this error, but it is easiest to make the symbols private (by declaring them with the C *static* keyword) in the plugin source. Another approach would be to use a linker option to ignore the duplicate symbols, or to delete the plugin copy of the code.

Plugin discovery at run time

There is one installation issue related to plugins. VisIt determines which plugins to open by reading a directory in `$VISITDIR` (the VisIt installation directory). Since regular plugins are not built for the parallel compute engine, `$VISITDIR` must point to the plugin directory for the Linux components. This means that `$VISITDIR` must be in a file system accessible from the compute nodes, such as Lustre or UFS.

While they are not full implementations of the standard dynamic linking routines, these replacement routines provide the functionality required to support VisIt plugins. Extra build steps beyond the standard VisIt build are required to run `bldplug` and `gendl` for all plugins to be included in the parallel compute engine. The Linux-based components continue to use the regular plugin shared libraries. Because the replacement routines effectively emulate their standard counterpart, no source changes to the component are required.

Socket Communication

Interprocess communication with TCP/IP sockets allows unrelated processes to exchange data within a given computer system or across a network. This is not available for Catamount, however. Rather, MPI communication is the standard communication for Catamount programs; but MPI only works within a domain of compute nodes of related processes. So, MPI cannot be used as a lower-level communication layer to emulate sockets.

Portals

The Cray XT3 system uses Portals, a low-level communication facility that can support the essential features of sockets. The Portals library allows arbitrary processes to exchange information between node types (login or compute nodes). It is used by both MPI and throughout the Cray XT3 system software for data transfer between nodes.

It would be possible to replace the socket-based communication within VisIt with an alternative implementation based on Portals. This would involve some source code changes, but they would be fairly localized because the organization of the VisIt software isolates the communication implementation in a module from the bulk of VisIt. This would add a restriction to VisIt that would limit all components to execute on a given Cray XT3 system, since Portals only operates within a single system.

Another approach would be to use Portals to implement a socket-like interface for the Catamount component only; and introduce a new program to act as a bridge between the Catamount socket emulation and the Linux components using standard sockets. In this way, the Linux components could be spread across a TCP/IP-based network of computers, although the parallel compute engine would still be tied to the Cray XT3 system.

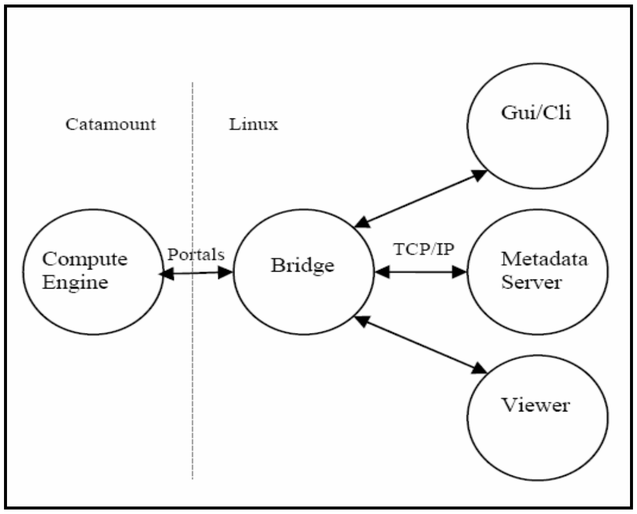

Figure 2: Socket Emulation via Portals

The socket emulation for Catamount can be crafted to minimize or eliminate source code changes to VisIt. To do this, a somewhat lengthy but well-defined set of routines is required to provide standard socket and network functions (`connect`, `bind`, `gethostbyaddr`, and so on). Since none of these calls are implemented within Catamount, there are no side effects for other parts of VisIt to introduce these functions. This is simplified because VisIt uses the socket-specific `sendmsg` and `recvmsg` calls for data transfer, rather than the general purpose `write` and `read` system calls.

SOLD Implementation

A purpose-built library and daemon were created to implement a sufficient socket emulation capability for VisIt. Called SOLD (Socket OffLoad Daemon), the library provides the necessary socket and network functions. The functions use Portals to pass the requests on to the Linux-based daemon (a helper process), which implements the functions (that is, it offloads the implementation from Catamount to Linux).

For example, to communicate with another VisIt component, the parallel compute engine first calls the `socket` function. This function passes the request to the daemon, which creates a socket on the Linux node, and passes back a handle (file descriptor placeholder) to VisIt. Visit uses this handle to call the `connect` function, which causes the daemon to connect the previously created socket to the TCP/IP port of the other VisIt component.

In this way, the parallel compute engine is unaware that it is running in a special partition of the Cray XT3 system, since it can communicate normally with all VisIt components, even those running on other computers.

There are performance implications for this type of communication. If all processes of a large parallel job were to attempt to communicate simultaneously, they would easily overwhelm the SOLD daemon. In the case of VisIt, only MPI process rank 0 actively uses sockets to communicate with the other components. Because there is an extra user-domain process in the communication path (the daemon), both communication latency and bandwidth are negatively impacted versus a direct communication path. For the purposes of supporting VisIt, this is not a performance bottleneck.

One interesting realization did arise during the SOLD implementation. It became apparent in the debugging phase that all of the parallel compute engine processes performed some network-related calls. This put stress on the SOLD daemon process. Later investigation revealed that this work was unnecessary, and only occurred because of an omitted check during initialization. Only MPI process rank 0 needed to make the call. A simple change to VisIt – a bug fix, actually – resolved this problem.

Performance

A benchmark suite provided by AWE was used to test VisIt performance. The input dataset was a 32 gigabyte file in TyphonIO format. A set of four Python scripts was used as input to the VisIt cli to simulate typical interactive usage. The test included visualization of pseudocolor, mesh, filled boundary, wire frame, and contour plots with rotate, pan, and zoom operations. Image files were written at several points.

The performance goal was to update frames on the user display at a rate of at least 1 per second. When the test was run on a Cray XT3 system with dual core 2.4 GHz processors, the goal could not be consistently met with 16 compute engine processes, but at 32 processes the goal was demonstrated.

Figure 3 shows the maximum and average seconds per frame for the six tests within the first of the four benchmark runs.

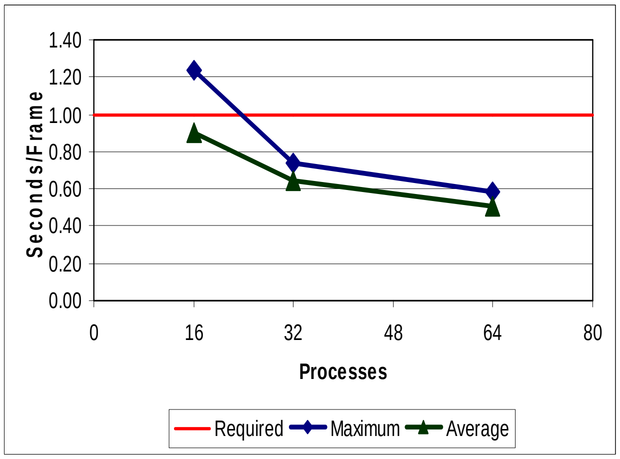

Figure 3: Freund Benchmark Scaling

Conclusion

VisIt, including its parallel compute engine, now runs on the Cray XT3 system at AWE. The implementation is efficient and scalable, and it meets the performance goals of the project. Minimal source code changes to the application were required to port it, although extensive changes to the build process were necessary.

To accomplish this, two special-purpose implementations of standard system features were created, one to emulate dynamic linking, the other to emulate TCP/IP sockets. Both of these implementations perform well within AWE's performance goals.

Acknowledgments

The author thanks Sean Ahern of Oak Ridge National Laboratory, Hank Childs of Lawrence Livermore National Laboratory, and Matthew Wheeler of the Atomic Weapons Establishment for valuable technical consultations as well as material contributions during this project. Tony Booker of Cray Inc. designed and implemented SOLD. His contributions were crucial.

About the Author

Kevin Thomas is a staff member of the Performance Team at Cray Inc.