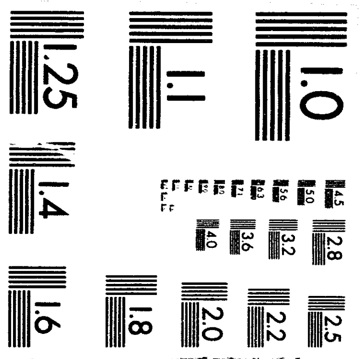
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

1 of 1

# COMMUNICATION LIBRARY FOR RUN-TIME VISUALIZATION OF DISTRIBUTED, ASYNCHRONOUS DATA

**John Rowlan**
**Argonne National Laboratory**
**Mathematics and Computer Science Division**
**Argonne, IL 60439**
**rowlan@mcs.anl.gov**

Brian T. Wightman
University of Wisconsin
Oshkosh, WI 54901
wightman@sol.acs.uwosh.edu

**Abstract**

In this paper we present a method for collecting and visualizing data generated by a parallel computational simulation during run time. Data distributed across multiple processes is sent across parallel communication lines to a remote workstation, which sorts and queues the data for visualization. We have implemented our method in a set of tools called PORTAL (for Parallel aRchitecture data-TrAnsfer Library). The tools comprise generic routines for sending data from a parallel program (callable from either C or FORTRAN), a semi-parallel communication scheme currently built upon Unix sockets, and a real-time connection to the scientific visualization program AVS. Our method is most valuable when used to examine large datasets that can be efficiently generated and do not need to be stored on disk. The PORTAL source libraries, detailed documentation, and a working example can be obtained by anonymous ftp from info.mcs.anl.gov from the file portal.tar.Z from the directory pub/portal.

**Key Words:** Scientific visualization, networking, graphics, AVS, sockets, distributed communication

## 1 Background

Many large-scale computational science simulations, such as global climate modeling, structural biology, and superconductor modeling, are done on massively parallel supercomputers. These simulations are usually time-dependent and often generate megabyte-sized (or larger) datasets at each time step. A typical program execution might require the generation of many hundreds or even thousands of timesteps to produce a meaningful animation. The result can be many terabytes of data.

Such data is often stored on disk to be visualized and reviewed on a graphics workstation at a later date. This process is often called ***batch-mode*** visualization, since the visualization occurs after all the data has been created. Scientific visualization tools such as AVS are designed to work in this manner. Modern large-scale computations, however, can produce amounts of data that are prohibitively expensive to store on magnetic disk.

We have created a set of tools called PORTAL (for Parallel aRchitecture data-TrAnsfer Library) for visualizing data directly as the calculations occur. The use of these tools avoids the need for large disk storage. Such a real-time display mechanism can also be used to direct execution parameters during run time, something that cannot be done in batch-mode visualization.

We note that effective visualization also requires careful selection of graphics parameters, such as viewing angle, object transformations, object color and other properties. When visualizing static, precomputed datasets, interactive graphics tools are used to manipulate the image. In a many-frame, time-dependent visualization such as those described in this paper, these parameters must be selected before the animation begins. Typically, a sample dataset is used to select the graphics parameters.

## 2 Motivation

The development of the PORTAL tools was motivated by our work with a parallel climate modeling simulation. After successfully implementing a parallel version of the MM-5 and CCM-2 climate models, we wished to run the simulation creating a visual animation of the results. Having no

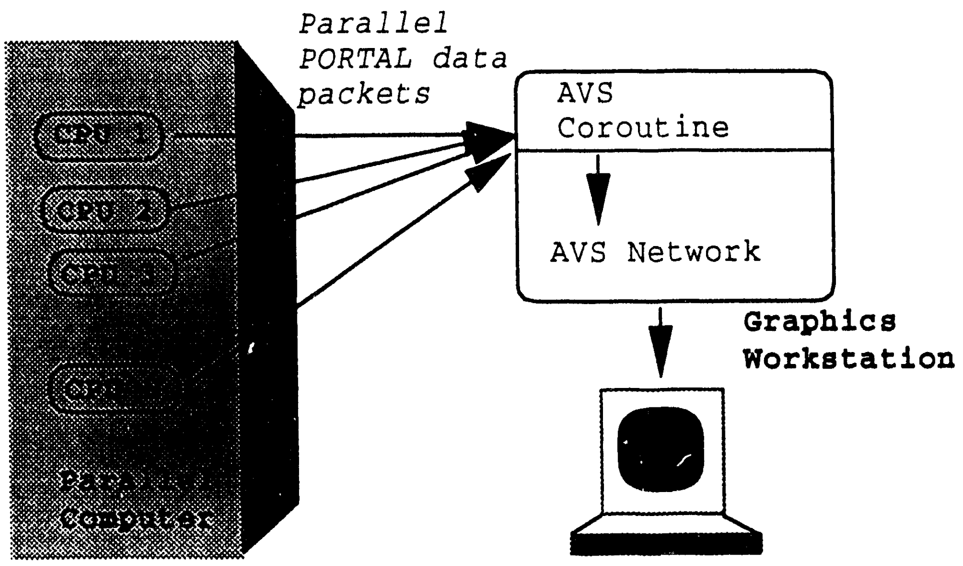

Figure 1: Overview of the PORTAL system

tools available to do this during run time, we were forced to write data to disk. However, since most animations of the climate model required many thousands of timesteps, at five megabytes per data set, writing each to disk was impractical.

As an alternative, we decided to send the data directly to an AVS process over Unix sockets. This approach proved quite successful, and we were able to create many video animations from the two parallel climate models.

Our initial efforts produced application-specific tools that required significant reworking to be applied to different simulations. Subsequently, we have made a communication library that is extensible and allows the user to define the type and amount of data to be sent.

## 3 The PORTAL Parallel Data Library

PORTAL (see Fig. 1) was developed to assist users of parallel programs to visualize the results of their computations. To make the tools easy to use, we felt it essential to remove the burden of socket programming from the user and to abstract the socket handling routines from the actual methods used to control sockets. We achieved this objective by providing the user with a small number of function calls that automatically package and send data from the parallel machine to the workstation for visualization.

Because of their portability and ease of use, Unix TCP/IP sockets were selected as the transport mechanism to send data from the remote parallel computation to the visualization process running on a local workstation.

Although PORTAL was designed with ease of use as a foremost consideration, the programmer must learn the PORTAL function calls used to initialize the sockets for sending, initialize the receiving software, appropriately package PORTAL data structures, and send the data to the receiving machine. These are listed in Table 1.

Table 1. PORTAL function calls

| |
|---|
| DS_init_socket_write() |
| DS_create_atom_handle() |
| DS_create_array_1D_handle() |
| DS_create_array_2D_handle() |
| DS_create_array_3D_handle() |
| DS_AVSinit() |
| DS_send_data_vectors() |
| DS_close_socket_write() |

To use PORTAL, a user first calls the DS_init_socket_write() function, which opens a communications port to a remote host and takes care of handshaking.

Next, the program must create handles to the data that the PORTAL system will be able to use. These are created with the DS_create_*datatype*_handle() functions. The function DS_create_atom_handle() is used for atomic data; one-, two-, and three-dimensional data types use DS_create_array_nD_handle() (where $n \in \{1, 2, 3\}$).

The receiving program is then told what to expect from the sender or senders through the DS_init function. This information includes the total number of processors sending data and the number and type of information

```
main(void)
{
  /* Initialize the socket for sending */
   DS_init_socket_write(&sock, &port, host );
...
  /* Create a handle for the three-D dataset.
      DS_create_array_3D_handle(&handle,  &DS_REAL,"3D  Data",&one,
                                 &XDIM,  &YDIM,  &ZDIM,
                                 &zero,  &zero,  &zero,
                                 &XDm1,  &YDm1,  &ZDm1,
                                 &DS_Z_DIM,  &DS_Y_DIM,  fracdata);
...
  /* assign this data subset to the vector datavector. */
      datavector[0]  =  &handle;
...
  /* Now send the AVS coroutine the appropriate parameters
     describing the format and type of data being sent. */
   DS_AVSinit(sock, &one, &one, (int*)datavector, &one);
...
  /* start the calculation loop */
  for(i = 0; i < MAX_i; i++)
   {
     calculate(&i,fracdata);
     /* Send the dataset to the receiver */
       DS_send_data_vectors(sock, &one, (int*)datavector, &one, &one);
   }
...
    DS_close_socket_write(&sock);                  /* close socket */
  }
```

Figure 2: Pseudo-code example

being sent (e.g., sixty processors, each sending two three-dimensional fields, one two-dimensional field, and five integers (atoms)).

Before the calculation is begun, a data structure is created to hold the data that will be sent to the receiver. This is done using the DS_create() function calls.

At present, the only receiving program supported by the PORTAL tools is AVS. Thus the DS_AVSinit() function is used to send initialization data. DS_AVSinit() takes a data handle vector and passes the information from the header needed to configure the AVS module through the socket. The AVS module then configures itself properly (see Section 5).

After the computation module has calculated its datasets for a particular time-step, the data is then sent to the receiving workstation by calling the DS_send_data_vectors() routine and supplying the address of the data structure. This function returns when it has passed all of the data to the remote process.

Finally, after all data has been sent, the function DS_close_socket_write() is called to tell the remote receiving program that the sending program is disconnecting. This functions calls the appropriate socket commands for notification of the remote program and the socket commands for closing the actual socket.

A pseudo-code example of the PORTAL process is shown in Figure 2.

## 4 Packet Structure

All communication between the parallel nodes and the graphics workstation is done through the use of a data-structure abstraction called a *packet structure*.

In PORTAL, these packets are used as the most basic transmission unit; all communication is of the packet form. The simple structure, appending only tag and length fields to the actual data, adds very little overhead to the transmission, but allows a remote program to identify, properly sort, and read all of the data that is sent to it.

The receiving machine is assumed to have no knowledge of the actual type and order of data being sent. Therefore, all packets have an eight-byte header, called a *tag field*, that describes the structure of the packet (see Table 2).

The first four bytes of the tag field are used to identify the type of data being sent. The next four bytes represent the length of the data being sent. Data is sent following the tag field.

There are three types of packets: control packets, AVS initialization packets, and data packets. Each is specified by the first four bytes of the tag field.

*Control packets* are used to open and close sockets, monitor and maintain connections, define byte fail counts, and identify other functions internal to underlying PORTAL processes to communicate between themselves.

*AVS initialization packets* contain information that is used to prepare AVS for incoming data; this information includes the type of data being sent and the total number of CPUs sending data.

*Data packets* include information about the particular CPU sending the data, a time stamp, array major and minor ordering, and the compositional relationship between the data being sent and the overall data (thus allowing a subset of data from a single node to be properly reconstructed within the larger field being calculated by all the nodes). Data packets are always sent in pairs. The first packet is a descriptor specifying the sending CPU ID, the full dimensions of the data field across all nodes, the subset of data being sent, and array major and minor ordering information. The second packet contains the data itself.

Table 3 shows a typical series of packets sent by PORTAL during execution of a program.

| tag field flags | possible values |
|---|---|
| tag<br>12-bits | control, avs_init, data |
| tag type<br>4-bits | start, end, header, data |
| avs data<br>4-bits | avs data, avs header |
| storage type<br>4-bits | atom, 1D, 2D, 3D |
| data type<br>8-bits | int16, int32, real, real8, char<br>complex8, complex16, logical |
| length<br>4-bytes | length in bytes of data section |
| DATA<br>variable | variable length byte stream |

Table 2: Packet structure (8-byte tag field followed by data)

| tag | tag type | avs data | storage | data type type | length | DATA |
|---|---|---|---|---|---|---|
| **At program startup:** | | | | | | |
| control | header | none | none | hello | zero | no data |
| control | header | none | none | set_fail_count | var | fail count |
| AVSinit | start | none | none | none | fixed | #CPUs, CPU id, num data packets to follow |
| **For each data packet being sent:** | | | | | | |
| AVSinit ... | none | data | type | type | var | data header |
| **After all the descriptors are sent:** | | | | | | |
| AVSinit | end | none | none | none | fixed | #CPUs, CPU id, number of packets |
| **For each data field to be sent within the calculation loop:** | | | | | | |
| data | start | none | none | none | zero | no data |
| data | header | none | type | type | zero | no data |
| data | data | none | none | none | var | DATA field |
| data | end | none | none | none | zero | no data |
| **At program completion:** | | | | | | |
| control | header | none | none | goodbye | zero | no data |

Table 3: Typical packets sent during program execution

## 5 AVS Implementation

AVS is a popular scientific visualization program in use at Argonne National Laboratory. It provides many tools for data visualization but no support for run-time visualization of large-scale parallel codes. Although originally designed, and primarily used, as a batch-mode visualization tool (i.e., on precomputed data), we felt that AVS could be enhanced to meet our need for run-time display of data.

In the AVS implementation of PORTAL, the data receiver is package within an AVS coroutine (called *Read Socket*). This module understands the PORTAL socket packet structure; and it reads, sorts, and queues the incoming PORTAL data packets. Read Socket outputs the data in AVS field format.

Since the sockets receiving data must be opened before the sending sockets can be opened, the AVS routine is the first process to the started. The AVS Read Socket module adapts itself to the type of data being sent and configures its output ports accordingly. However, at startup, the AVS module does not know what type of data it will receive, so it must start up in a wait state until the sending nodes each make a DS_AVSinit() function call.

After successful receipt of an DS_AVSinit() call from each processor, the AVS module learns the type of data it will be reading from the sockets, configures its output ports appropriately, and enters a run state.

Since the AVS module is in a wait state until the module is initialized, AVS will appear to hang until the remote program starts and has sent the initialization parameters. Thus the *Read Socket* module cannot be attached to an AVS network until the initialization stage has completed. This causes a problem when saving a network, since the Read Socket module cannot save its dynamic connections. However, it is a fairly simple matter to build and save the rest of the network, pull the read socket module onto the palette, start the remote application, and attach a couple of modules once the read socket module is configured.

Since multiple processors are sending data to the receiving workstation, and since these transmissions are potentially arriving at different rates, the AVS module must also queue up and synchronize the data packets it receives, to ensure that all data is ordered correctly.

Synchronizing data is accomplished via a data queue. By the use of a user tag, which is passed to the DS_send_data_vectors() function, the data is marked at a certain "time step" and then grouped with other packets from the same time step. The front item in the queue is not released until all of the machines have sent data for that time step. This means that if a processor does not have data for a particular time step, it must send at least a null packet.

Each queue entry is constructed of an ID marker, a list of machines that information is still needed from, and the list of data that has been received. This offers a quick check of whether or not the data can be taken from the queue and placed on the output ports of the module by examining the machine list to see whether there are any machines remaining to read from.

## 6 Conclusions

PORTAL allows a parallel program to send asynchronous distributed data to a remote graphics workstation for run-time visualization. Thus, it can offset the need to store many gigabytes or terabytes of data on disk. The immediate feedback of a run-time system also offers greater efficiency in identifying program errors or inaccurate parameters.

## 7 Future Directions

Future enhancements to the PORTAL package will include feedback from the graphics package to determine the behavior of the remote module. This could also be used to enhance the error recovery mechanisms of the system by allowing non-acknowledged packets, retries, and reconnects.

Support will also be added for other transport mechanisms including IPI-3, EUI, and FCS.

Finally, we plan to port the socket reader module to other graphics packages, allowing a user to pick a visualization package that best suits the tastes or needs of the programmer.

## Acknowledgements

This work was supported be the office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

## References

[1] "AVS Developer's Guide," Part 320-0011-02 Rev B, Release 4, Advanced Visual Systems Inc., May 1992.

[2] "Sun Network Programming Guide," Part 800-3850-10, Release A, Chapters 10,11, Sun Microsystems Inc., March 27, 1990.

DATE
FILMED
5/16/94

END