AVS for the CRAY T3D

Mitchell Roth, Dmitrii Zagorodnov, and *Karen Woys,* Department of Mathematical Sciences, University of Alaska, Fairbanks, AK

ABSTRACT: *AVS is a widely used package for scientific visualization which runs on platforms ranging from workstations to Cray Research parallel vector supercomputers. AVS employs a dataflow network architecture in which modules from several extensive libraries are interconnected to perform the desired visualization functions. This paper discusses the adaptation of AVS modules to the CRAY T3D for parallel execution. The implementation issues discussed in the paper are: (1) T3D I/O and control, (2) module generation and (3) parallel computational algorithms for the modules using MPI. Examples of AVS modules from the supported library as well as user-generated models which execute on the T3D are described.*

1 Introduction

A well recognized advantage of high performance computing environments is the ability to fully utilize advanced visualization techniques. Employing scalable parallel processing is no exception. Visualization techniques enhance the understanding of physical phenomena through the viewing of important relationships of information not normally seen by the scientist or engineer. Further, this information can more easily be communicated to others involved in the exploration process or who use the information to make important decisions, including the consumer who will decide whether to buy a product or not. However, many of the visualization processing steps are computationally expensive or require long production lead times.

In this project, we extend visualization capabilities to the CRAY T3D to take full advantage of parallel processing. In so doing, processing runtimes will be significantly reduced, opening new interactive avenues for the scientist and engineer to more completely explore and understand his data. To accomplish this, we will leverage the existing base of visualization software capabilities. This effort focuses primarily around the Application Visualization System (AVS) distributed by AVS, Inc.

2 T3D Architecture

The CRAY T3D is a parallel computer built around DEC Alpha RISC processors. The T3D is available in sizes ranging from 32 to 2048 processors (PEs). The memory architecture is physically distributed and logically shared (globally addressable). The name of the machine is derived from the the processor interconnect network, which is a three-dimensional torus [3].

The T3D requires a frontend host computer system to provide support for applications running on the T3D. All applications written for the T3D are compiled on the host system in MPP Fortran, Fortran 90, C or C++. The host system also provides job scheduling and I/O control for the T3D. Host systems include the CRAY Y-MP and CRAY C90 series computers. The Arctic Region Supercomputing Center (ARSC) operates a T3D with 128 PEs running at 150 Mhz and providing a peak speed of 19.2 GFLOPS. Each PE contains 8 MW (64 MB) of local memory, which provides 1 GW (8 GB) of globally addressable physical memory. The host machine for the ARSC T3D is a CRAY Y-MP M98 with 8 CPUS and 1 GW (8 GB) of memory.

The T3D processors are dynamically divided into partitions which appear to the user as logically separate dedicated machines, allowing multiple users to share the T3D. The number of processors allocated in a partition must be a power of 2. T3D jobs are initiated on the host system and may be run interactively or using batch queues. To run a T3D job, the host runs a process called *mppexec* which spawns an agent process on the host for each PE in the partition and then loads the T3D executables. These agents provide the control and I/O interface for the T3D through the frontend host.

3 AVS Overview

In order to understand how AVS can be adapted to run on the T3D, it is necessary to review the basic structure of AVS. AVS employs a distributed dataflow architecture which allows simulation models to be incorporated into visualization applications. The network paradigm employed by AVS lends itself extremely well to heterogeneous computations where, for example, a simulation model might be executed on a supercomputer while the visualization and control of the model parameters and visualization of the model output is performed either locally or remotely on a workstation.

The AVS structure embodies the principles of modularization, abstraction, and information hiding. The package is composed of *modules* written in C and FORTRAN, each of which performs a specific function. Using these building blocks, AVS users can interactively construct their own visualization applications by combining modules into executable dataflow networks. The modules in the network may be executed locally on the same machine which is hosting the AVS application, or they may be executed remotely on a different machine.

AVS includes several hundred supported modules organized into module libraries. The system is extensible and users may generate new modules using the Module Generator. The International AVS Center (IAC) maintains an archive of contributed modules which may be downloaded to augment the standard libraries.

We propose to extend the scalability and applicability of AVS by producing an environment for the creation and execution of AVS modules on the CRAY T3D parallel supercomputer. This environment will allow users to create AVS modules for the T3D which can then be incorporated in standard AVS module libraries for general usage. For the user of these libraries, access and use of the T3D is nearly transparent. The appropriate T3D module icon from the module library is simply connected into the AVS dataflow network. If the AVS application is hosted on the T3D frontend, typically a Y-MP, the module will be a local module, otherwise it will execute remotely on the T3D frontend. In either case, the AVS module will provide the user interface and execution control for the T3D code.

This requires the porting of key AVS modules that can fully utilize the computational power of the T3D. For example, volume rendering of large 3D data sets, such as environmental or MRI data can consume hours of CPU time. Using the parallel processing power of the T3D, it is possible to render large volumetric data sets interactively in seconds or less. Such a renderer opens the possibility of creating powerful new learning and diagnosis tools using high performance computing and networking. Other examples are image processing, and iso-surface generation. Many computationally important AVS modules have been identified and will be ported to the T3D and be made available to the public through the IAC.

3.1 AVS Kernel

The AVS kernel is a proprietary product of AVS, Inc. The kernel controls the execution and communication of AVS modules. The machine which hosts the kernel for an AVS application is the local host and all modules in a AVS network running on that machine are local modules. Modules in an AVS network which execute on a machine other than the one where the kernel resides are remote modules.

The kernel also provides the user interface to several AVS subsystems, including the *Image Viewer*, *Graph Viewer*, and *Geometry Viewer*. These subsystems are used to display bit-mapped images, plots and contours, and three-dimensional geometries, respectively. In addition, the kernel provides the interface for the *Network Editor*, which is used to construct AVS networks. Like the AVS modules, the AVS kernel has been ported to many workstations and mainframes, including the CRAY Y-MP.

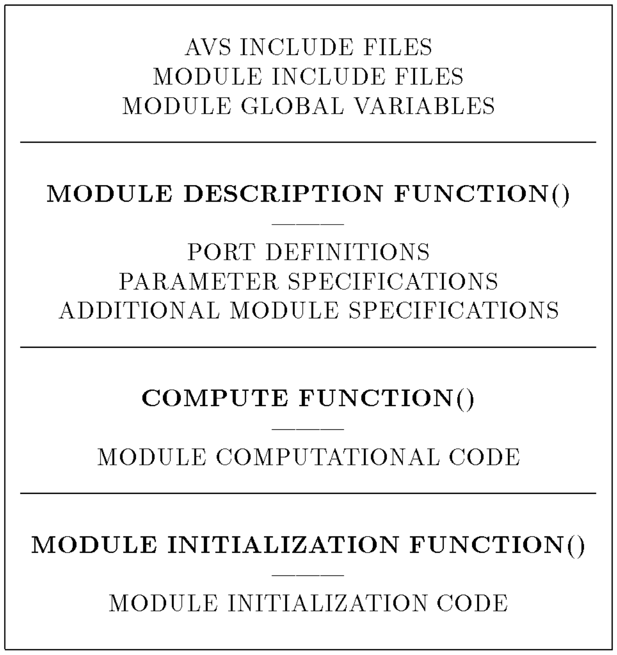
3.2 AVS Modules

Modules are the computational units in an AVS dataflow network. AVS applications are created by connecting modules from several extensive libraries to perform the desired visualization functions [4]. There are four types of modules in AVS: (1) input, (2) filters, (3) mappers and (4) output. A module may have one or more *ports*, through which it is connected to other modules in the network. The ports have data types associated with them and connections between ports represent the flow of data in the network.

Most modules also have control panels which allow parameters to be set for each instance of a module. The parameters on a module control panel can be assigned values interactively, through command line scripts, or by assigning them to ports which are connected to other modules in an AVS network.

Another convenient feature of AVS is that it supports the execution of modules on remote AVS hosts of heterogeneous hardware types. Multiple hardware types are possible because the network communication and data transfer mechanism between the AVS kernel (see below) and the remote module are based on standard Unix TCP/IP network protocols and the data representation is based on Sun's External Data Representation (XDR) [5].

Remote module execution involves three aspects: remote system requirements; the local *hosts* file that AVS uses to locate remote modules; and the AVS *Network Editor* user interface to remote modules. Remote modules cannot be part of the AVS kernel and must be compiled and linked on the remote machine. On the local system the AVS kernel refers to the *hosts* file to locate remote module host names and directories. The *Network Editor* is used to access remote modules from the local machine by selecting the desired remote host and modules from the *Module Tools* menu. The module icon for a remote AVS module is identified in an AVS network by coloring the module control button pink.



AVS INCLUDE FILES
MODULE INCLUDE FILES
MODULE GLOBAL VARIABLES

MODULE DESCRIPTION FUNCTION()

PORT DEFINITIONS
PARAMETER SPECIFICATIONS
ADDITIONAL MODULE SPECIFICATIONS

COMPUTE FUNCTION()

MODULE COMPUTATIONAL CODE

MODULE INITIALIZATION FUNCTION()

MODULE INITIALIZATION CODE

Figure 1: AVS Module Template

3.3 Module Generator

One of the most powerful tools in AVS is the Module Generator, which allows an AVS user to create new AVS modules through a point and click GUI. The Module Generator creates a wrapper for user code and supplies the data structures, communications and control required for modules to function in an AVS network. Presently there are variations of the Module Generator which allow modules to be created in FORTRAN, C, and C++ [6][7].

Figure 1 shows the template for a module created by the Module Generator. Modules may be edited, compiled and debugged using the Module Generator environment, or a template may be generated and modified as desired using standard compilation procedures. The template includes three functions which are used to define a module. These are: (1) the description function, (2) the compute function, and (3) the initialization function. These functions contain the actual code which is executed by the AVS executive when a module is connected and executed in an AVS network.

The description function contains calls to AVS kernel routines which define the name and type of the module, the specifications for the ports and control panel parameter widgets for the module and the names of the compute and initialization functions for the module. The initialization function contains code which is executed when an instance of a module is first loaded into a network. The compute function contains the actual executable code for a module. The arguments to the function consist of the module parameters, whose values are read from the control panel, and pointers to the data structures corresponding to the ports on the module. The compute function is called whenever the AVS executive "fires" the module during the execution of a network.

4 T3D Modules

In order to create modules for T3D execution, we first create a Y-MP module in the format shown in Figure 1 using the Module Generator. This module will provide the AVS interface to the T3D though a process running on the Y-MP frontend. The compute function of this Y-MP module will spawn a T3D process to perform the actual computations. The Y-MP compute function will also communicate parameter values and data as required to and from the T3D process. The Y-MP module may be created in any of the Y-MP programming environments supported by the Module Generator, including C, C++ or Fortran.

The T3D process, on the other hand, must be implemented in the programming environments available on the T3D. These environments include MPP Fortran, Fortran 90, C, and C++. Figure 2 summarizes the relationships between the processes which run cooperatively on the Y-MP and the T3D.

5 T3D/Y-MP Communication

A variety of techniques may be used to exchange information between the T3D and the frontend host. The best technique to use depends on the amount and type of data to be exchanged. Both machines employ 64 bit processors, but they use different floating point formats. Thus, format conversion becomess a concern when exchanging floating point data. The available methods by which the T3D can communicate to the frontend are: (1) files, (2) Parallel Virtual Machine (PVM) messages, (3) process memory tables, and (4) Unicos signals. The applicability of each of these techniques is discussed below and is summarized in Figure 3.

Files are the easiest interface to implement, but also the slowest. All data types can be exchanged between any of the programming environments using ASCII files. But the transfer rate using formatted I/O between the T3D and the Y-MP frontend was only about 25 KB/s in tests we conducted by sending a file of image data from the T3D to the Y-MP. If performance is important (as it usually is) then file I/O can only be considered when exchanging very small amounts of data.

PVM messages are much faster than file I/O and allow all data types to be exchanged. The price for the speed of PVM is the additional programming required to implement the PVM message encoding, decoding and exchange of the messages. Since PVM has become one of the defacto standards in parallel processing, this type of communication is quite portable and even allows the T3D

CONTROL PROCESS ON Y-MP	**COMPUTE PROCESS ON T3D**
PROVIDES AVS INTERFACE	PERFORMS COMPUTATIONS
CONTROLS T3D PROCESS	CONTROLLED BY Y-MP PROCESS
SENDS PARAMETERS & DATA TO T3D	RECEIVES DATA & PARAMETERS FROM Y-MP
RECEIVES T3D OUTPUT	SENDS OUTPUT TO Y-MP
CREATED BY MODULE GENERATOR	T3D DEVELOPMENT ENVIRONMENT
WRITTEN IN C OR FORTRAN 77	WRITTEN IN C OR MPP FORTRAN

Figure 2: Comparison of Y-MP and T3D AVS processes.

FILES
- Easy
- All Data Types
- Slow (25KB/SEC)

PVM MESSAGES
- Portable
- All Data Types
- Moderate Speed (2MB/SEC)

PROCESS MEMORY FILES
- Integer Data Only
- Fast (50MB/SEC)

UNICOS SIGNALS
- Semaphores
- Fast (0.01 SEC)

Figure 3: T3D/Y-MP communication techniques.

to exchange messages directly with hosts other than the frontend (although the messages still pass through the frontend agent). Using PVM we have obtained I/O rates between the T3D and Y-MP frontend of approximately 2 MB/s.

The fastest way to exchange data between the T3D and the Y-MP is through the use of the process memory file system, which gives users access to the address space of a running process. This file system consists of files named */proc/nnnnn*, where *nnnnn* is a process ID formatted in decimal. Each file contains the address space of the process it represents. Process files may be read and written using the C system I/O routines *open*, *lseek*, *read*, and *write*. Reads and writes to a process file perform reads and writes on the address space of the process involved. By exchanging pointers to their data areas, processes on the the T3D and Y-MP are able to read and write directly into each other's memory. No data translation is performed, but library routines are available for converting between the floating point formats. In our tests involving the exchange of integer data, the transfer rate was about 50 MB/s. This is clearly the method of choice when megabytes of data must be rapidly exchanged.

The final form of communication between the T3D and the frontend host is Unicos signals. Signals are semaphores which allow a single bit of data to be exchanged and are primarily useful for synchronization of processes on the T3D and the frontend. The execution time for a signal between the T3D and the Y-MP is on the order of 10 milliseconds.

6 Mandel Module

As an example of a T3D AVS parallel module, we developed a module which calculates the image of a mandelbrot set. The image is based on the iteration

$$z_{i+1} = z_i^2 + c,$$

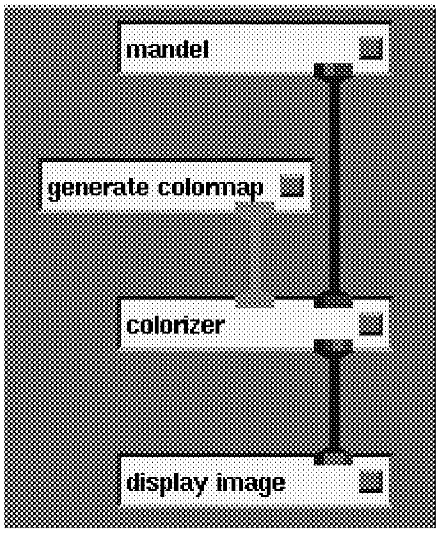


Figure 4: T3D **mandel** module in AVS network.

where z and c are complex and $z_0 = 0$. The values of c in the complex plane for which z_n is bounded as $n \rightarrow \infty$ form the interior of the set. The limit can be determined computationally because it is known that if $|z_n| \geq 2$, then $z_n \rightarrow \infty$. An image of the mandelbrot set is created by assigning the pixels in the image to a region in the complex plane. Each pixel corresponds to a particular value for c for which the iteration is performed until $|z_n| \geq 2$ or an iteration limit is reached. Exterior points which are distant from the set boundary diverge most rapidly, while points in the interior reach the iteration limit. Complex and colorful images of the set are obtained by using the iteration count for each pixel as an index into a color map for the pixel.

6.1 AVS Interface

The **mandel** module we developed has one output port which returns the computed mandelbrot set as an AVS byte image. The module can be connected into an AVS network as shown in Figure 4. The **generate colormap** and **colorizer** modules are used to assign a colormap to the image generated by **mandel** and displayed using the **display image** module.

The control panel for the **mandel** module is shown in Figure 5. The slider widgets set the value of c in the center of the image where $c = x + iy$. The size of the image in the complex plane is controlled by the zoom widget. The zoom factor causes the mandelbrot set to be magnified by 2^M, where M is the zoom factor. For $M = 0$, each side of the image is 2 units in the complex plane.

An image of the mandelbrot set produced by the **mandel** module is shown in Figure 6. This 512x512 pixel image requires that the iteration be performed on each of

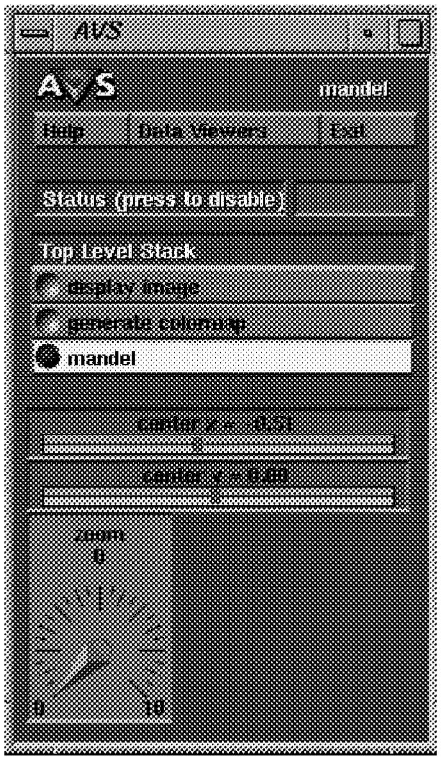


Figure 5: Control panel for T3D **mandel** module.

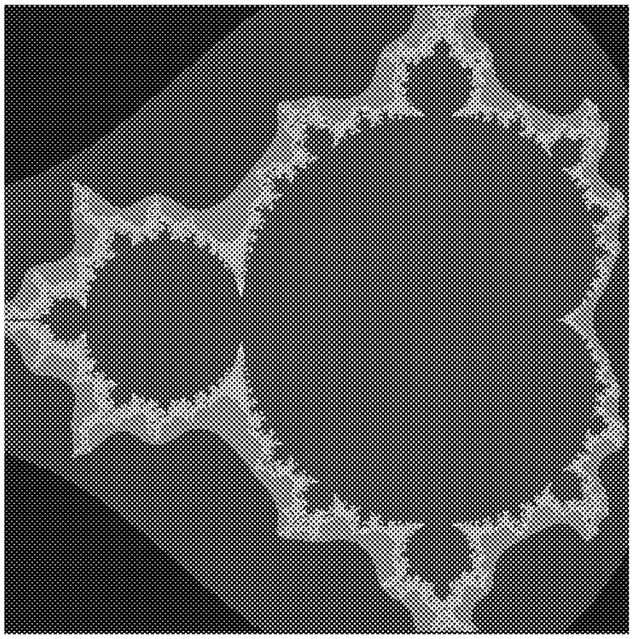


Figure 6: Image from **mandel** module on T3D.

the 256K pixels in the image for a total operations count of slightly more than 1 billion floating point operations.

Figure 7 shows the execution output from the **mandel** module using 8PEs. The output shows the input parameters that are passed to each PE, followed by the completion messages from each PE including the number of floating point operations (FLOPS) performed by each PE, the execution time and the resulting MFLOPS per second. For this run, the mandelbrot computation required 3.9 seconds, which gives 265 MFLOPS per second, or 33 MFLOPS per second per PE.

Note that the processor loads in Figure 7 are balanced almost perfectly through the use of MPP Fortran data and work sharing in this implementation. Since there is essentially no interprocessor communication, the run time scales linearly with the number of PEs. Using 32 PEs, the execution time can be reduced to less than one second, which equates to a speed of just over 1 GFLOPS per second. The details of the **mandel** module implementation are discussed in the following two sections.

6.2 T3D Process

The T3D process for the **mandel** module is implemented in MPP Fortran [1] using the data sharing and work sharing models. These models automatically distribute data and loop computations among the available processors and eliminate the need for explicit message passing.

The T3D process receives its input parameters `X`, `Y`, and `ZOOM`, from a file written by the Y-MP process prior to calling the T3D process. Since this file consists of only one short line containing the three parameters, the I/O time required is negligible.

The declarations and computational loop for the T3D process are shown in Figure 8. The 512 x 512 shared array `PIX` is used to store the iteration counts for each pixel in the image. The `SHARED` directive for this array tells the compiler to distribute the array over the PEs such that successive elements are stored in successive PEs, modulo N, where N is the number of PEs.

The `DO SHARED` directive preceding the iteration loop tells the compiler to assign the iterations of the nested loops with indices `J` and `K` to the PEs where `PIX(J,K)` is stored. This guarantees that the mandelbrot iteration for `PIX(J,K)` will be performed on the same PE where `PIX(J,K)` is stored. `FLOPS` is also a shared array and is used to keep track of the operation counts performed by each PE. The intrinsic function `MY_PE()` is used to obtain the PE number, which is stored in the private variable `ME` for indexing the `FLOPS` array.

Since the mandelbrot iteration diverges rapidly for external points away from the boundary of the set and not at all for internal points, the amount of computation to be performed varies greatly according to the position of a pixel in the image. Since the iteration for each pixel is

```
PROGRAM MANDEL
PARAMETER (IX=512)
PARAMETER (IY=512)
PARAMETER (MAXPE=128)
COMPLEX Z, C
REAL TIME, X, Y, ZFACX, ZFACY, ZSQR
INTEGER ME, I, J, K, COUNT, ZOOM
INTEGER PIX(IX,IY),FLOPS(MAXPE)
COMMON /SHARED/PIX,FLOPS
CDIR$ SHARED PIX(:BLOCK(1),:BLOCK(1))
CDIR$ FLOPS(:BLOCK(1))
INTRINSIC MY_PE
C  Initialize PE# and FLOPS counter
ME = MY_PE()
FLOPS(ME+1) = 0

•
•
•

CDIR$ DOSHARED (J,K) ON PIX(J,K)
DO 200 J = 1, IX
  DO 100 K = 1, IY
    COUNT = 0
    Z = (0.0,0.0)
    C = CMPLX(X + (J-1-IX/2)*ZFACX,
              Y + (K-1-IY/2)*ZFACY)
C  Mandelbrot iteration
    DO I = 1, 1023
      Z = Z*Z + C
      ZSQR = REAL(Z)**2 + AIMAG(Z)**2
      COUNT = COUNT + 1
      IF (ZSQR .GT. 4.0) GO TO 10
    END DO
10     PIX(J,K) = COUNT/4
    FLOPS(ME+1) = FLOPS(ME+1) + COUNT*10 + 4
100   CONTINUE
200 CONTINUE
```

Figure 8: MPP Fortran code for **mandel** module

```
mandel_compute: (cx,cy) = (-0.509259, 0.000000), zoom = 0
 T3D: (CX,CY) = (-0.50925900000000002,0.), ZOOM =0
 T3D: (CX,CY) = (-0.50925900000000002,0.), ZOOM =0
 T3D: (CX,CY) = (-0.50925900000000002,0.), ZOOM =0
 T3D: (CX,CY) = (-0.50925900000000002,0.), ZOOM =0
 T3D: (CX,CY) = (-0.50925900000000002,0.), ZOOM =0
 T3D: (CX,CY) = (-0.50925900000000002,0.), ZOOM =0
 T3D: (CX,CY) = (-0.50925900000000002,0.), ZOOM =0
 T3D: (CX,CY) = (-0.50925900000000002,0.), ZOOM =0
PE 0 done:  129114612 FLOPS,   3.87 sec,   33.363 MFLOPS/SEC
PE 4 done:  129224232 FLOPS,   3.87 sec,   33.366 MFLOPS/SEC
PE 2 done:  129111942 FLOPS,   3.87 sec,   33.354 MFLOPS/SEC
PE 6 done:  129111942 FLOPS,   3.87 sec,   33.354 MFLOPS/SEC
PE 3 done:  129083902 FLOPS,   3.87 sec,   33.354 MFLOPS/SEC
PE 7 done:  129083902 FLOPS,   3.87 sec,   33.354 MFLOPS/SEC
PE 1 done:  129189202 FLOPS,   3.87 sec,   33.365 MFLOPS/SEC
PE 5 done:  129251952 FLOPS,   3.87 sec,   33.366 MFLOPS/SEC

TOTALS:   1033.172 MFLOPS in   3.89 sec =    265.447 MFLOPS/SEC
mandel_compute: xsize = 512, ysize = 512, maxpix = 255
```

Figure 7: Execution output from T3D **mandel** module using 8 PEs.

performed on the PE where the pixel is stored, the distribution of the pixel array controls the PE assignments and hence the load distribution for the PEs. By assigning successive elements of the array to succesive PEs, every PE processes a subset of the pixels which is uniformly distributed over the entire image. This accounts for the nearly perfect load balancing observed in Figure 7.

When the computation loops on all PEs have finished, PE 0 writes the entire pixel array into the Y-MP process memory table using the MPP Fortran callable versions of the C language I/O functions *open*, *seek*, and *write*.

6.3 Y-MP Process

The Y-MP process for the **mandel** module was developed in C using the standard AVS Module Generator. This module follows the outline of the template shown in Figure 1. The module description function which defines the output port and control panel for the module was created entirely by the Module Generator. The description function defines an output port for image data and a control panel which controls two sliders for the (x,y) coordinates of the center of the mandelbrot set and a dial for the zoom factor.

The module compute function contains code which receives the module parameters from the control panel widgets and writes a small file containing these parameters. The compute function then executes a Y-MP system call to initiate the T3D process. The T3D process reads the parameter file created by the Y-MP process, computes a mandelbrot image and returns the image to the Y-MP process via its process memory file. The Y-MP then allocates memory for a scalar byte field to hold the image and copies the returned integer data into the field structure for the output port.

The initialization function created by the Module Generator was used without modification.

7 VolRender Module

A module developed at ARSC performs real-time volume rendering of large 3D data sets, such as environmental or MRI data [8]. Using the parallel processing power of the T3D, it is possible to render large volumetric datasets interactively at a rate of several images per second, compared to minutes or longer per image on other machines. Such a renderer opens the possibility of creating powerful new learning and diagnosis tools using high performance computing and networking. For example, interactive MRI images could form the basis for a virtual reality dissection laboratory. The availability of such a tool on the national network would also allow medical specialists to perform analyses and consultations on high resolution MRI images which would otherwise be impossible. This level of visualization complexity can only be achieved with the fast generation of data provided by the T3D.

The **VolRender** module is shown in an AVS network in Figure 9. The output from the module is an image field which is displayed using the **image viewer** module. The Y-MP and T3D processes for the **VolRender** module are both written in C. The T3D process employs **shmem**

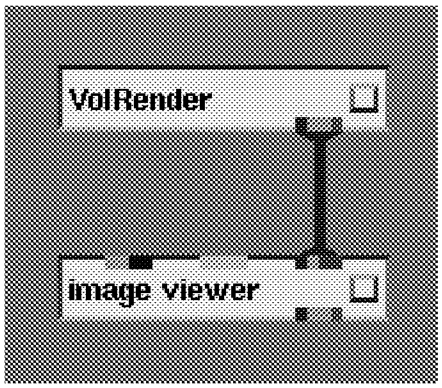


Figure 9: T3D **VolRender** module in AVS network.

message passing and communicates with the Y-MP via process memory files.

Figure 10 is an example of a volume rendered CT image from the Visible Human dataset[9]. Bone material is white, muscle is red and softer tissues appear in yellow. The Visible Human dataset from which this image was created consisted of 1800 slices with a resolution of 512 by 512 voxels per slice. At four bytes per voxel, the complete head-to-toe dataset is about 1.8GB in size.

The VolRender module has a multilevel control panel which allows interactive control of all 3D volume viewing parameters, such as lighting, shading, color and transparency. When **VolRender** is used with the **display tracker** module, the view can be rotated, scaled and translated interactively using the AVS mouse conventions for viewing 3D objects. Using 64 PEs on the T3D, the time to render the entire 1.8GB Visible Human volume dataset from a new point of view is about 5 seconds.

8 Library Modules

Having laid the foundation for creating AVS modules on the T3D, we are in the process of implementing selected modules from the standard AVS library. Some of the modules which are being implemented on the T3D are: **field math**, **interpolate**, **downsize**, **compute gradient**, **fft**, **convolve**, **tracer**, and **volume render**. These modules are all computationally intensive and several of them have also been succesfully optimized for vector execution on the Y-MP [10]. The parallel implementations for several of these modules are described in the following sections. All of these modules are written in C. The parallel versions use MPI[11] to perform the necessary data distribution functions.

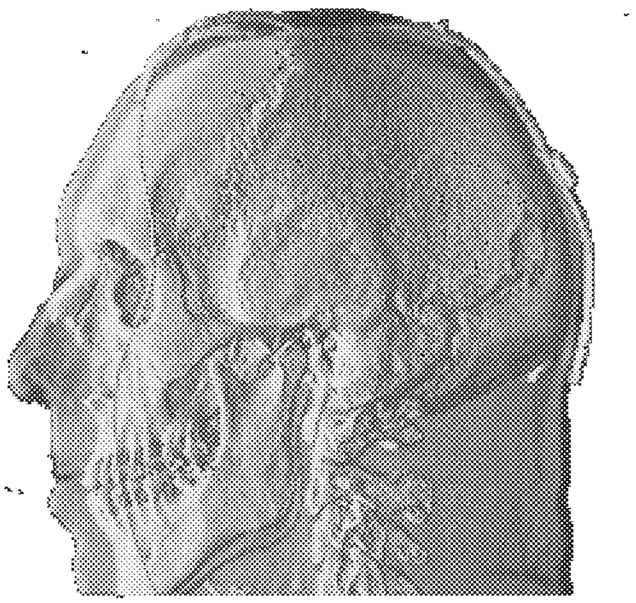


Figure 10: Visible Human image rendered by **VolRender** module on T3D.

8.1 Field Math Module

A *field* is an AVS data type which stores generalized multidimensional arrays. The **field math** module is used to perform arithmetic and logical operations (+, -, *, /, NOT, AND, OR, XOR, shift, square, square root, and root-mean-square) on AVS fields. The inputs consist of one or two fields of any type (byte, integer, float, double) and size. Binary operations may be performed either with a single field and a constant or between two fields of the same size. If the types differ, bytes are promoted to integers and integers, floats and doubles are converted to doubles.

Because of the diverse data types involved and large number of possible operators, the **field math** module provides a good model for the conversion and exchange of data that is required for AVS data types. The parallel implementation uses data parallelism based on the *owner computes* rule. After any necessary type conversions are performed by the Y-MP process, the T3D process is called and the input fields are transferred to the T3D master PE using the process memory file. The master PE partitions and distributes equal-size blocks of the input fields to the other PEs using the MPI_Scatter function. The processors perform the required operation on their own blocks in parallel and return the results to the master PE using MPI_Gather.

The primitive nature of the operations performed by the **field math** module makes it difficult to do enough computation in a single T3D processor to justify the cost of distributing the fields to the T3D processors. However, the structure of the T3D **field math** module is applicable to other modules which perform more intensive computations, as described below.

8.2 Interpolate Module

The **interpolate** module uses either point sampling or bi/trilinear sampling to compute intermediate values to change the size of a field. The input consists of any 2D or 3D scalar field of any type. The output is a proportionally resized field matching the type and size of the input.

In the **field math** module, the operation performed on each element of the field involves only a single element and the MPI Scatter/Gather functions can be used to partition the input fields into non-overlapping blocks, which are distributed over the available PEs. In the **interpolate** module, the interpolation operators involve several neighboring elements in the field. For this reason, each PE must contain one or more extra rows and columns of data surrounding the block of the field that is owned by the PE.

As in **field math**, the input field is transferred from the Y-MP to a master PE on the T3D. The master PE then uses the MPI_Send function to distribute blocks of the field with the necessary overlap to all the processors. These blocks are received by the MPI_Recv function in the slave processors. All processors perform interpolation on their own blocks in parallel and then return the results to the master PE using another pair of MPI_Send and MPI_Recv calls.

8.3 Compute Gradient Module

The **compute gradient** module is used to compute gradient vectors for 2D or 3D scalar byte fields. The output is a field of the same dimension as the input, but each field element is a 3D vector of floating point values representing the gradient at that point.

The **compute gradient** module requires the immediate neighbors of each point for which the gradient is computed. Thus, each PE requires exactly one extra row or column on every side of the block of elements to be calculated. The data distribution scheme was implemented using the MPI_Send and MPI_Recv functions in the same manner as in the **interpolate** module.

9 Future Work

A major portion of our development effort for the modules which have been completed has been directed at the communications interface between the Y-MP and the T3D. This includes data conversion, exchange, and distribution, as well as process control. All of the T3D modules employ a data parallelism model based on the owner computes rule. As we attempt to implement additional modules from the AVS module library, we expect to devote more effort to developing and benchmarking parallel implementations for the algorithms which are employed in modules such as **fft**, **convolve**, **tracer**, and **volume render**.

In the future, our work will also address the creation of new T3D application modules using AVS as the user interface for the application. To accomplish this, we are in the process of developing a Module Generator targeted for the T3D. With this tool it will become possible to quickly incorporate T3D code into an AVS module.

We intend to use the Module Generator to create T3D modules for new modelling and visualization applications on the T3D. In particular, we propose to adapt a large-scale 3D reservoir model to the T3D AVS environment using the proposed T3D Module Generator. This type of model is one of the fundamental tools which the oil industry in Alaska, and around the world, uses to model and enhance petroleum production. However, the underlying reservoir modelling technique is not limited to hydrocarbons and may also be applied to other subsurface flow problems, such as groundwater contamination.

Another application of interest is the processing of synthetic aperture radar (SAR) images from earth-orbiting satellites. Production code is already running on the T3D which performs the computationally expensive process of calibrating and rectifying SAR images so they may be used in large area mosaics and combined with other satellite imagery, such as LandSat and AVHRR [12].

10 Conclusions

Our experience has shown that it is relatively easy to develop new AVS modules from T3D applications. In addition, AVS provides a powerful user interface for these applications. Using the T3D, significant performance gains are possible for existing modules relative to other AVS platforms. The limiting factor is usually the speed at which the AVS data can be transferred between the frontend host and the T3D. In situations where the amount of parallel computation to be performed is substantial, our results indicate that the resulting speedups can be dramatic. Thus, execution of compute intensive AVS modules on the T3D appears to be an extremely promising approach to the problem of harnessing the power of parallel computation.

11 Acknowledgments

Jon Genetti and Greg Johnson of ARSC participated in the development and testing of the codes described in this paper. Dan Katz of Cray Research assisted with the implementation of the process file I/O between the Y-MP and T3D.

This project was funded by Cray Research, Inc. and by the Strategic Environmental Research and Development Program (SERDP) under the sponsorship of the Army Corps of Engineers Waterways Experiment Station.

References

[1] *Cray MPP Fortran Reference Manual*, SR-2504 6.1, Cray Research, Inc., June, 1994.

[2] *Standard C Reference Manual for MPP*, SR-2506 4.0, Cray Research, Inc., June, 1994.

[3] *CRAY T3D System Architecture Overview*, HR-04033, Cray Research, Inc., September, 1993.

[4] *AVS Module Reference Manual*, Release 5, Advanced Visual Systems, Inc., February, 1993.

[5] *AVS User's Guide*, Release 4, Advanced Visual Systems, Inc., May, 1992.

[6] *AVS Developer's Guide*, Release 4, Advanced Visual Systems, Inc., May, 1992.

[7] Jiang, T. Ming and Sarnowski, Beata, "C++ Module Generator for AVS," AVS '94 Conference Proceedings, pg 186-198, April, 1994.

[8] Johnson, G. and Genetti, J., "Medical Diagnosis using the CRAY T3D," Proceedings of the 35th Semiannual Cray User Group Meeting, pg 70-77, March, 1995.

[9] The Visible Human Male Dataset, Visible Human Project, National Library of Medicine, Bethesda, MD, 1995.

[10] Woys, K. and Roth, M., "AVS Optimization for CRAY Y-MP Vector Processing," Proceedings of the 35th Semi-annual Cray User Group Meeting, pg 78-89, March, 1995.

[11] Clarke, L.,"The MPI Message Passing Standard on the CRAY T3D," Edinburgh Parallel Computing Centre, University of Edinburgh, 10pp, 1994.

[12] Logan, T., "Terrain Correction of Synthetic Aperture Radar Imagery Using the CRAY T3D," Proceedings of the 35th Semi-annual Cray User Group Meeting, pg 268-274, March, 1995.