# Massive Remote Batch Visualizer
## *Porting AVS/Express to HECToR*

George Leaver, Martin Turner
Research Computing Services, Devonshire House,
University of Manchester, Manchester, UK, M13 9PL

20 July 2010

**Abstract**

The AVS/Express Distributed Data Renderer visualization product has been ported to HECToR to allow visualization of large datasets using parallel software rendering. A number of changes to the AVS code were required to run in the HECToR environment, the most significant being the ability to make MPI calls from the main AVS/Express user interface executable running on a login node, and the addition of an MPI-based image compositing scheme. Initially the MRBV project was only considering non-interactive batch execution of AVS/Express. However scalability improvements in the code allow interactive use with higher process counts than any other previous installation of the product.

# Contents

# 1 Introduction

The AVS/Express visualization product [1] is a scientific visualization application allowing the rendering of many forms of data. It uses the visualization pipeline method [2] in which data is read in to the system, possibly filtered, and then mapped to geometry for rendering. At each stage the user is able to control how that stage processes the data, whether it be choosing an appropriate file reader, deciding how the data is filtered (for example whether it is down-sampled or cropped) and then how that data is converted to geometry. This last stage usually employs a suitable visualization technique such as isosurfacing or volume rendering to produce an image of the dataset. AVS/Express provides a user interface in which these stages are represented by modules and allows the user to connect the modules together forming a data-flow style *network*. Figure 1 shows an example network in which data is read in via a file reader module before being cropped (filtered) and then mapped to geometry via various isosurface modules and an orthoslice module. All geometry is sent to the viewer module at the bottom of the network. A module may have its own parameters which are controlled in the user interface panel on the left. The user can interact with the visualization in the viewer window.

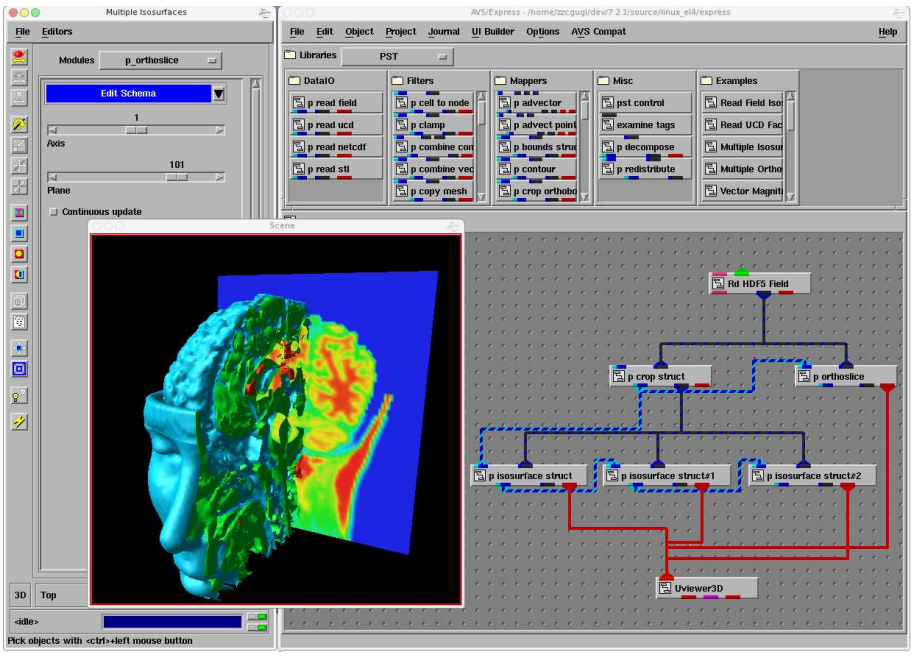


Figure 1: AVS/Express Network Editor and Visualization Window.

The particular version of AVS/Express to be ported is the AVS/Express Distributed Data Renderer (DDR) Edition, version 7.2.1. This product provides parallel module computation by allowing the AVS modules to execute on decomposed datasets, whereby the dataset is split in to smaller domains of data and distributed to a number of compute processes which then process the data according to the AVS modules in use. In addition to the standard AVS *Software*

and *OpenGL* renderers, the DDR edition also provides a parallel renderer (referred to as the *MPU renderer* for historical reasons) that is capable of rendering the geometry generated by the parallel modules. Separate rendering processes receive geometry from particular compute processes, render an image of the geometry and then composite the images together to form a final visualization of the entire dataset. This technique is referred to as sort-last rendering [3] and allows a dataset to be rendered that exceeds the capabilities of a single rendering process (whether that be a GPU or software renderer). This version of the product is able to render data on distributed compute nodes where no GPU hardware is available using the MesaGL [4] software implementation of OpenGL [5].

A motivation for this project is that the visualization of large datasets has long been a bottleneck in applications where validation of data acquisition from scientific equipment is required at an early stage. Such validation would allow correctness of methods (such as the set up of a physical experiment) to be determined prior to further computational or imaging-machine resources being spent. This is particularly important to our target end users in Materials Science. Despite advances in GPU hardware, researchers are able to produce datasets that are too large to visualize using modern graphics workstations and clusters. An alternative to multi-GPU systems is to use the large memory and core counts offered by supercomputers such as HECToR even if GPU hardware is not available.

# 2 The dCSE Project

The main goal of this dCSE project is to port AVS/Express DDR to the Cray XT4 hardware provided by HECToR (Phase2a). This will allow the visualization of large datasets that currently exceed the capabilities of our GPU-based visualization systems. Working with Materials Science we have access to datasets that are typically 50–500GB in size. The datasets provide material density data acquired by CT scanning equipment, some of which is provided by the Henry Moseley X-Ray Imaging Facility at Manchester [6]. We also expect to acquire datasets from the I12 JEEP beamline at the Diamond Light Source, RAL. It should be noted that AVS/Express DDR is not limited to this type of data and so will be of use to researchers working with other forms of data. In particular AVS has NetCDF [7] and HDF5 [8] readers in addition to the generalised AVS Field reader which allows many forms of data to be described and read in to the application.

10 months of development time were allocated to the project, beginning May 2009, with the possibility of 2 months of NAG CSE support to assist with optimisation during the 10 month period. This option was not taken up as it was deemed unnecessary to conduct very low level optimisation due to the interactive nature of the application.

The main development task was to modify the AVS/Express code such that it could operate within the HECToR runtime environment. This was broken down in to the following subtasks:

- Provide a mechanism to allow the AVS/Express main application (network editor, module user interfaces, visualization window) to operate on

the login nodes, where X11 functionality is available, yet communicate with parallel module and rendering processes executing on the backend nodes. Modifications to the AVS source tree should be kept to a minimum, avoiding significant architectural changes that would impact other platforms. This will make AVS/Express appear more like the open source ParaView [9, 10, 11] application in structure.

- Modify the image compositing architecture within AVS/Express so that it can use MPI on the backend nodes. At the time of development an open source image compositing library is used by AVS/Express. This provides no MPI functionality and only supports dynamically linked applications.
- Optimise the existing MPI communication within the AVS/Express. It is known that the parallel renderer uses point-to-point communication which is expected to reduce scalability to large processor counts.
- Provide AVS networks that demonstrate the common visualization tasks that users will perform.

AVS/Express must be run from a login node with X11 forwarded over the ssh connection to the user's X server. The X server must support the `GLX` protocol. This is not a particularly efficient method of running an X11 application and results in the overall rendering frame rate having an upper bound that cannot be improved by parallel rendering because the limiting factor becomes the time needed to transfer the image from the login node to the user's X server. In the case of the AVS renderer the upper bound is approximately 5.0 frames per second (fps) for a 512x512 window and 1.1 fps for a 1024x1024 window when rendering remotely to a test desktop system running linux. While this appears to be low the visualization does in fact remain sufficiently responsive that the user can interact with the visualization. We are more concerned with the ability to scale out the memory usage to increase the size of dataset that can be visualized.

The AVS/Express source tree has been compiled with the default GNU compiler (currently 4.4.2). This is one of the compilers supported by the AVS build process for 64bit linux platforms (the other being the Intel compiler). The group in Manchester have a source code agreement with AVS allowing access to the AVS/Express source tree although certain components are only available as libraries. The build process is to compile the AVS `base` executable which then reads the *V* module description files (AVS/Express uses its own module description language named *V*). Processing of the V files generates C/C++/Fortran code for those modules which is then compiled as part of the build process. This produces the final `express` executable.

## 2.1 Licencing

AVS/Express DDR requires an MPE Developer Edition licence to run. UK academic users with an existing license should contact AVS UK for details about accessing their licence on HECToR. Users without a licence should note that AVS/Express is available under the Eduserv CHEST agreement [12].

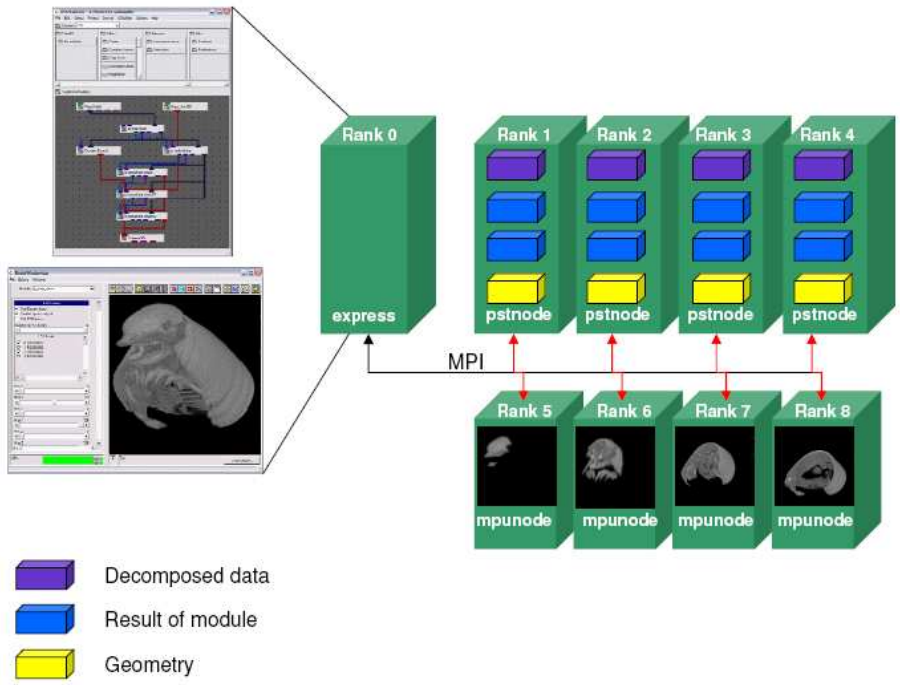


Figure 2: AVS/Express DDR architecture.

# 3 MPI Forwarding

We now describe the main development task of this dCSE project. It is useful to consider the current architecture of AVS/Express DDR first.

## 3.1 Existing DDR Architecture

The existing visualization code is an MPI application that comprises a number of executables. The main executable is `express` which provides the AVS network editor, module user interface and visualization window. This is always rank 0 in the MPI job. The other components are two types of MPI executables, namely the `pstnode` and `mpunode` executables. described below.

Figure 2 shows the scheme now described: the `pstnode` processes execute parallel module codes according to the modules in the visualization network. A key concept is that a dataset is never accessed directly by the `express` process. Instead a dataset is decomposed in to a number of smaller sub-domains, one for each `pstnode` process. The `express` process will instruct the `pstnode` processes on how they should process their sub-domain of data. For example, a min/max filtering module may pass its parameters (minimum and maximum values) to the `pstnode` processes. They will filter their sub-domains of data accordingly. A small amount of information may need to be returned to the `express` process from each `pstnode` process so that it can update the user interface. For example, a reduction on the actual minimum and maximum data values present in each sub-domain can be used to display the dataset's overall minimum and maximum

values in the user interface. Similarly a parallel isosurface module will receive parameters from the user interface (e.g., the isosurface level to compute) but the computation will take place within the `pstnode` processes on their current sub-domain of data. The sub-domain of data within a `pstnode` process remains fixed. It is this decomposition of data and encapsulation within the pstnode process that allows AVS/Express to work with large datasets. At no point should sub-domains be gathered and recomposed in the main `express` process. Doing so would almost certainly exceed the memory resources of the node on which the `express` process is running.

The visualization network will specify which modules should produce renderable geometry. Any geometry produced by the `pstnode` processes will be passed directly to an assigned `mpunode` process. The `mpunode` MPI processes execute the AVS/Express rendering methods in parallel. They receive global scene graph data from `express` and insert in to the scene graph the geometry received from their assigned `pstnode`. Hence each `mpunode` process only renders a fraction of the total geometry in the scene. The images produced by the `mpunode` processes are composited together (using either depth testing or alpha blending) and the final image is sent back to the `express` process for final display in the user interface. All communication between the various MPI processes is performed using MPI point-to-point or collective communication facilities depending on whether the message is domain-specific or common across all domains. However, at this stage the compositor uses tcp/ip communication as no MPI layer exists in the open source compositing library used by AVS (see section 4).

## 3.2 MPI Proxy

In order that the `express` user interface process can be run on the HECToR login node a number of changes to AVS/Express are required. Most significantly all MPI functionality must be removed from the executable so that it can be run outside of the MPI job. Two strategies were considered, the first being to add another communication API to `express` and the parallel module framework, removing any dependency on MPI. This strategy was rejected because it would have resulted in a significant rewrite of large sections of AVS code, in particular the framework used to manage the parallel modules and rendering. Also, users developing their own parallel modules would potentially have to be aware of both the MPI and non-MPI communication methods.

The second strategy, implemented in this project, is to provide an alternative MPI library that does not use the Cray MPI layer but still allows the `express` executable to be linked without major source code changes. The `express` executable can then continue to make MPI function calls that don't require the Cray MPI layer found on the backend nodes.

Our replacement MPI library is referred to as XPMT (Express MPI Tunnel). `express` source includes `xpmt_nompi.h` (rather than `<mpi.h>`) and is linked against `libxpmt.so`. It is compiled as a serial login node executable using the dual-core programming environment. `libxpmt.so` contains our MPI functions which communicate with a *proxy* MPI process via a standard tcp/ip socket. This proxy process is a genuine Cray MPI process (always rank 0 in the MPI job) running on the backend nodes. As shown in Figure 3, the non-MPI `express` sends requests for MPI functions to be called on the compute node on which the

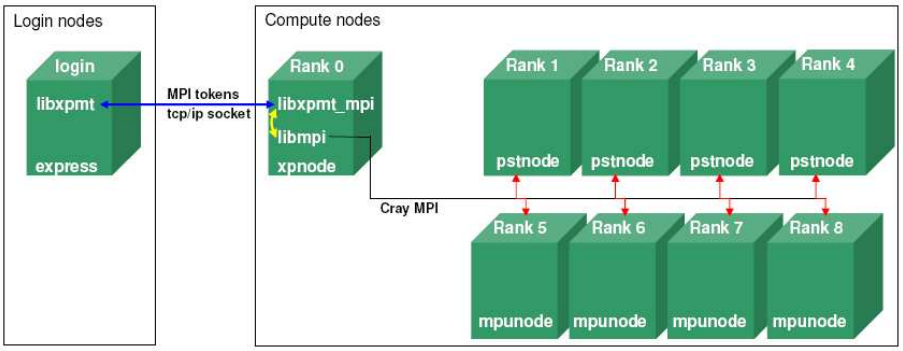


Figure 3: Forwarded MPI from login nodes to compute nodes.

proxy `xpnode`[1] is running. This process receives the request and any required arguments to the requested MPI function. For example, a request for `MPI_Send()` requires the buffer, count, datatype, destination rank, tag and communicator arguments expected by the MPI function. Upon receiving the request the `xpnode` process calls the Cray MPI function with these arguments. Any results of the function (return type, buffer content etc.) are sent back to the `express` process via the socket. Hence the non-MPI `express` process is unaware that it is calling MPI functions via a proxy.

The MPI proxy process (`xpnode`) includes `xpmt_mpi.h` and `<mpi.h>` and links `libxpmt_mpi.a` and the Cray MPI libraries. This allows it to map XPMT's representation of MPI types to Cray MPI types. In our implementation the XPMT representation of MPI types are all integers that act as indices in to a table of real Cray MPI types within the `xpnode` proxy process. When `express` creates new MPI objects (communicators, datatypes, statuses etc.) the proxy creates the equivalent objects using the Cray MPI layer and a mapping between the two representations is maintained.

The `pstnode` and `mpunode` MPI processes are unchanged (they are standard Cray MPI executables) and will communicate with the `xpnode` process as though it were the `express` process. This is because they think the rank 0 process is `express` (and `xpnode` is always rank 0) and only communicate with it in response to MPI functions being called by `express`. For example if `express` posts an `MPI_Recv()` the proxy `xpnode` will make the same function call from rank 0. When the `pstnode` or `mpunode` processes make a corresponding `MPI_Send()` call the proxy `xpnode` will receive the data and pass it back to the non-MPI `express` process. Hence the sending processes are completely unaware that the `xpnode` process is a proxy for `express`.

It should be noted that the `pstnode` and `mpunode` processes communicate with each other via the Cray MPI layer and so benefit from this optimised library and its use of the Cray interconnect. The largest data transfer occurs between a `pstnode` and its associated `mpunode` when geometry is passed for rendering. This communication never touches the proxy `xpnode` process, occurring entirely

[1]During this discussion we refer to the proxy as `xpnode`. In practice we allow `pstnode` rank 0 to provide the proxy functionality. This makes writing the `aprun` command simpler. See section 3.4.

within the Cray MPI domain and so suffers no change in performance as a result of removing Cray MPI from the `express` user interface. The amount of data sent by the non-MPI `express` process via the socket is in general small because it is mainly command-and-control messages from the `express` user interface. The global scene graph information sent from `express` to the `mpunode` render processes is also small because most of the geometry is generated by the `pstnode` processes.

### 3.3 XPMT Performance

Use of the MPI proxy imposes a performance penalty on any MPI communication to and from the `express` process. Figure 4 shows execution times for various tests using the proxy and the standard Cray MPI layer. A Cray MPI executable was used entirely on backend nodes. The same tests were compiled against the XPMT replacement library so that what was rank 0 now runs on the login node and communicates via the `xpnode` proxy processes (which becomes rank 0 in the MPI job). The remaining ranks are standard Cray MPI executables. This matches exactly the changes made to AVS/Express. The tests perform common communication calls (MPI_Bcast, MPI_Gather, MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv, MPI_Waitall) that are all found in AVS/Express. They each send or receive using an array of `MPI_INT`s where the size of the array increases as: 64, 128, 256, 512, 1024, $1024^2$, $2\times1024^2$ and $8\times1024^2$ (the Gather test uses array sizes up to 512). The graphs show the total time for all array sizes used for each test. The use of the MPI proxy can slow communication by a factor of three, approximately, in some cases. Within AVS/Express most of the messages from the `express` process are less than 1K in size. The largest message sent back to the `express` process from the compute or render nodes is usually that containing the final composited rendered image, which for a $512\times512$ window is 1Mb (assuming 4 bytes per pixel). Given that AVS/Express is an interactive application (rather than a numerical simulation) we find this performance penalty acceptable.

### 3.4 Rank Placement

To simplify the `aprun` command needed to start the various MPI processes (the `xpnode` proxy, several `pstnode` and `mpunode` processes) we merge the functionality of these processes as follows. A new `pstmpunode` process is created by merging `pstnode` and `mpunode`. This new process will behave as a `pstnode` if it has an even rank or as `mpunode` if it has an odd rank. If the rank is 0 it behaves as the MPI proxy process. We retain the `pstnode` executable and allow it to also behave as the MPI proxy if running as rank 0. It can also operate as a dummy process to help rank placement (discussed below). The combined `pstmpunode` executable allows a simple `aprun` command to be used. For example, the command for a dataset decomposed in to 15 domains and being rendered by 15 rendering processes is:

```
aprun -n 31 -N <mppnppn> pstmpunode <args>
```

where `<args>` are the command line options required by the MPI proxy process to contact the `express` process running on a login node (switches specifying *hostname* and *port number*). Note that the MPI proxy rank 0 process prevents

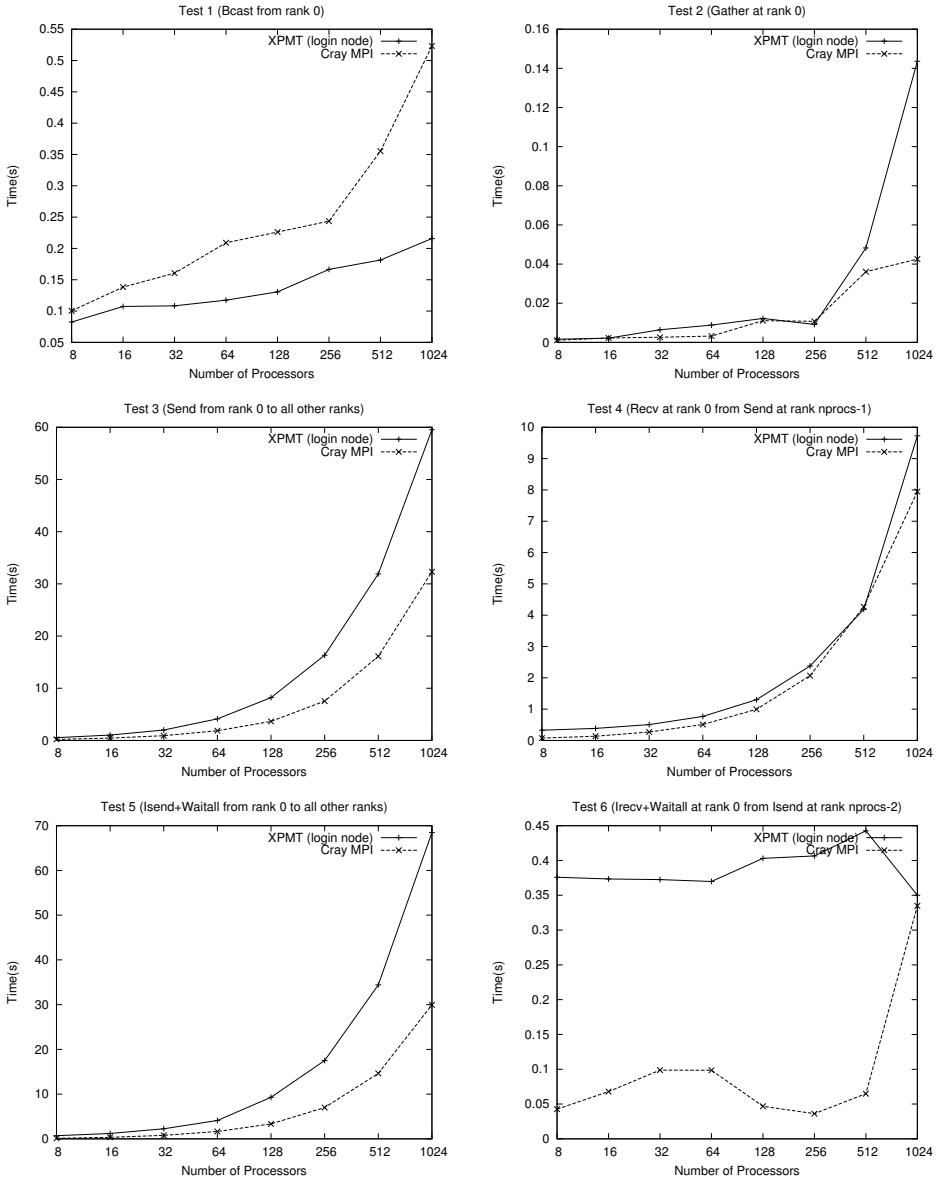


Figure 4: Tests performed using a Cray MPI executable entirely on backend nodes (rank 0 times shown) vs an XPMT executable with one rank running on a login node (times shown) communicating via the MPI proxy.

us using all 32 processes in the 32-core queue (in this example) because the number of `pstmpunode` processes must be an even number (so that half can act as `pstnode` processes and half as `mpunode` processes). Hence in this example we use $15 + 15 + 1$ processes (the 1 being the MPI proxy rank 0 process).

AVS/Express DDR automatically assigns a `pstnode` style process to an `mpunode` style process (they will all be `pstmpunode` executables) so that the `pstmpunode`(m) can act as the rendering process for that single `pstmpunode`(p) process. By using the odd/even scheme we ensure AVS/Express always pairs `pstmpunode`$(\mathrm{p})_{2i}$ with `pstmpunode`$(\mathrm{m})_{2i-1}$. Ideally we would like these paired processes to be on the same physical backend node so that they communicate using intra-node communication. However, the MPI proxy (rank 0) creates an off-by-one layout where a `pstmpunode` may be paired with another such process on a different physical node. This may result in a small performance reduction. It is possible to overcome this off-by-one problem at the expense of a few dummy processes. By using the `pstnode` executable we can specify a number of dummy processes that do not contribute to AVS module processing. They are there simply to pad out the physical node on which the MPI proxy process is running. This then allows all `pstmpunode` process pairs to be placed on the same physical node. The number of dummy `pstnode` processes is always `mppnppn`$-1$. When using dummy `pstnode` processes we also use `pstnode` as the MPI proxy process. Hence the total number of `pstnode` processes is equal to the `mppnppn` value. However, the small gain in performance may not be worth the cost of running a few dummy processes. The `aprun` command when using dummy `pstnode` processes (for the 32-core queue) when using `mppnppn=2` becomes:

```
aprun -n 2 -N 2 pstnode <args> : -n 30 -N 2 pstmpunode
```

or, using four processes per node, the number of domains is reduced to 14:

```
aprun -n 4 -N 4 pstnode <args> : -n 28 -N 4 pstmpunode
```

A `ddr` shell script is available that generates the PBS jobscripts given the number of domains required, the `mppnppn` setting and whether dummy `pstnode` processes are to be used to help rank placement. The script will submit the `express` job to the serial queue on the login node. When this process executes, it will submit the `pstmpunode` job to the relevant parallel queue. The `express` process then listens for the parallel job starting. The MPI proxy rank 0 process will connect to the `express` process via a socket. At this point the AVS/Express user interface will appear and the application can be used. This is similar to the ParaView start-up procedure but is completely automated by the `ddr` script. The jobs scripts can be generated but not submitted if customisation is required.

## 3.5 Summary

We have implemented a method of running the AVS/Express user interface on the login nodes while performing parallel module processing and rendering on the backend nodes. Minimal changes to the AVS source code have been made by allowing the `express` process to make MPI function calls from the login node. This is done by forwarding those calls to a proxy rank 0 process running on the backend nodes. Communication between the various AVS/Express processes

via the proxy is transparent to those processes. While there is some performance reduction associated with this mechanism, the reduction is not critical for an interactive application and not so great that the application becomes uninteractive.

The need to forward the X11 user interface over the ssh connection to HECToR is another performance reducing factor. It would be possible to run the `express` process on the user's local desktop (as a client) provided an ssh tunnel could be established from the HECToR login node to the user's desktop. The MPI proxy process could then have its connection to the login node tunnelled to `express` running on the user's desktop. This would be more efficient than forwarding the X11 connection from the login node. However, such connections are not permitted on HECToR.

# 4 Image Compositing

AVS/Express currently uses the open source Paracomp [13] compositing library, initially developed by HP. The library requires dynamic linking (it has a framework that requires the use of `dlopen()`) and supports tcp/ip, InfiniBand and Mellanox network layers. It also uses multiple pthreads for networking, control and image operations. The basic compositing method employed is the Scheduled Linear Image Compositing [14] method. While this has proved to be an effective compositing library on render clusters that have InfiniBand or Mellanox networking, the lack of MPI support, coupled with the use of multiple pthreads and dynamic linking, resulted in our decision to remove it from AVS/Express on HECToR. We have compiled the library to provide just the core image compositing routines. This can be compiled statically and provides a method of compositing two images together, using either depth testing or alpha blending. This is the fundamental image operation required of any compositor.

Having removed the Paracomp communication facilities we have implemented the 2-3 Swap Image Compositing method [15]. This allows all image communication between render processes (which perform the compositing operations) to take place within the Cray MPI layer. The 2-3 Swap method is similar to Binary-Swap Compositing [16] but removes the need to have a power-of-2 number of render processes. This is important in AVS/Express due to the use of the MPI proxy process.

Parallel image compositing reduces the time required to blend the rendered images of the sub-domains (recall the dataset is divided in to sub-domains of data) from every render process. Sending full sized images from all render processes directly to one process for blending (using either depth testing or alpha blending) would introduce a bottleneck at that process. By dividing images at every process in screen-space (i.e., discarding rows of pixels) and exchanging these sub-images, *all* processes can take part in the blending operation. Eventually every process will have a sub-image that contains a fully rendered dataset. The final step is to gather these sub-images at a single process and copy them in to the image buffer. This final gather step is not such a bottleneck because the screen-space sub-images are small at this stage (each sub-image contains $1/n^{th}$ of the total number of pixels in the final image, when $n$ is the number of render processes). All of this communication takes place on the backend nodes. We then have to send the final image through the MPI proxy process to the

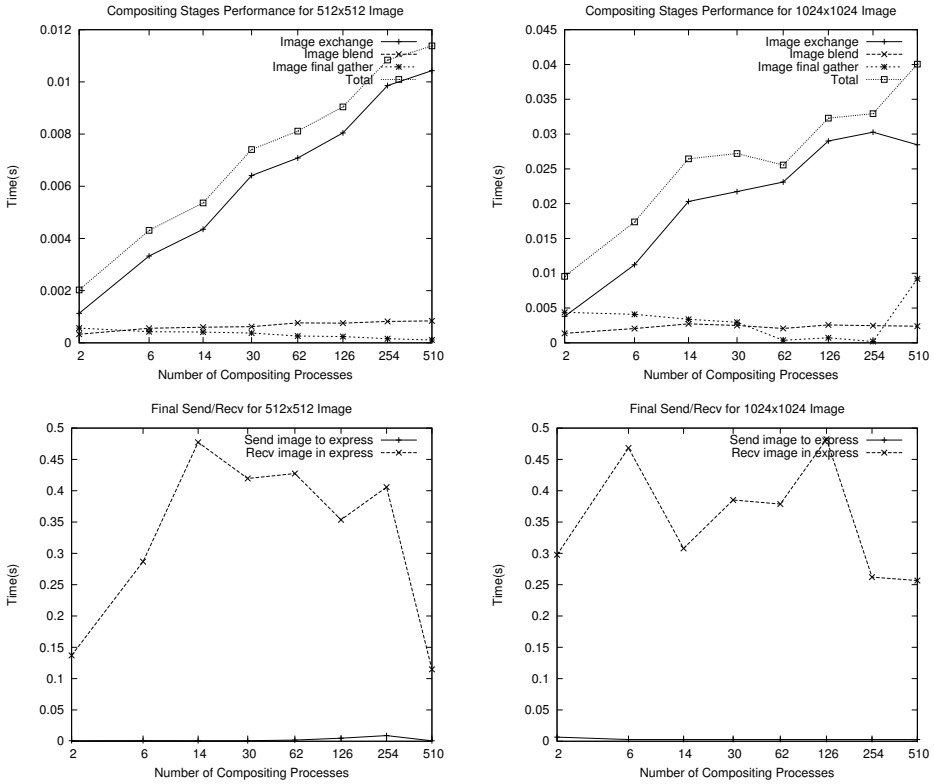


Figure 5: Image compositing operations for two window sizes (top row). Final composited image send / recv through the MPI proxy (bottom row).

`express` process so that it can display it in the user interface.

## 4.1 Compositor Performance

Figure 5 shows the performance of various compositing operations. The top row shows timings for one particular rank for an image of size $512^2$ pixels and $1024^2$ pixels. The rank chosen is the middle rank for each number of render processes (any arbitrary rank could be used). This gives a snapshot of what one particular render process is doing during image compositing. All times are totals for the entire image compositing operation. The *Image exchange* time is the total time the middle rank process spends exchanging screen-space images with other processes during the compositing operation. The *Image blend* time is the total time spent blending image pixels (using depth testing). The *Image final gather* time is time spent sending the blended sub-image to the single process (usually the first `pstmpunode` render process) that gathers the sub-images from all other render processes. Note that the gather operation writes the sub-images directly in to the final image buffer. The *Total* time is the sum of the previous three operations. Even the slowest compositing times correspond to a frame rate of 90fps for a $512^2$ window and 30fps for a $1024^2$ window.

The second row shows the time required for the `express` process to receive the image through the MPI proxy, and the time for the single render process to send the image to the proxy. The sending time is small because this is occurring within the Cray MPI layer. The receiving time is significant and will limit the overall frame rate of the renderer. This image transfer occurs over the tcp/ip socket through which `express` and the MPI proxy process communicate.

We have added an OpenMP wrapper around the Paracomp library's core image compositing routine. This allows the blending time to be reduced in the compositing operation for large images. A trade-off is needed between the number of `pstmpunode` processes running on a node (usually two or four) against the number of OpenMP threads dedicated to image blending. However, given that the image transfer from the MPI proxy to `express` is the limiting factor, it is recommended that only one OpenMP thread be used. This allows the number of `pstmpunode` processes on a node (`mppnppn`) to be increased.

# 5 Parallel Renderer Communication

Examining the AVS parallel renderer code reveals that point-to-point communication calls are used where collective calls might be more efficient. The MPI patterns within the renderer were initially developed to match sections of the code where another non-MPI communication library was used. This library did not provide collective calls. The main tasks within the renderer where this communication style occurs are the distribution of the scene graph from the `express` process to the `pstmpunode` render processes and the sending of messages to update process caches.

The scene graph contains general information (such as camera position, lighting, background colour) and also placeholders representing every renderable object that will eventually be created by the `pstmpunode`(p) parallel module processes. Renderable objects could be arrays of triangles for an isosurface or texture data for volume rendering, for example. The renderable objects are stored in a cache inside the rendering process and the placeholders in the scene graph reference objects in the cache. The scene graph is sent to every `pstmpunode` render process, as are messages that initiate update operations on the caches within those processes (for example, messages to delete out-of-date geometry objects). It is the communication of the scene graph and these messages that can be optimised.

When an image is to be rendered `express` determines whether it should generate a scene graph containing new objects and geometry or whether it is simply re-rendering an existing scene graph, perhaps with a change of camera position. The former case occurs if, for example, the user changes an isosurface level value or modifies a volume render transfer function. The renderer must delete the existing cached objects and receive new ones from the module processes. Messages must be sent to all render processes asking them to update their caches so that all caches are consistent across the rendering processes. In the latter case, where no new geometry is generated, a much simpler scene graph can be distributed to the render nodes and no messages requesting changes to the caches are required. In this case the scene graph will reference existing geometry in the render nodes' caches but will include new settings, such as the camera position. The graphs in figure 6 show times for the communication of a

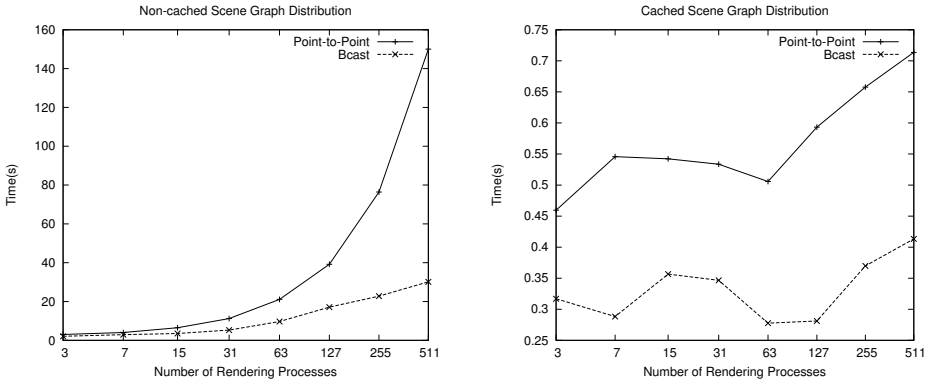


Figure 6: Scene Graph build and communication for a new non-cached scene graph and a subsequent cached scene graph, using MPI_Bcast and point-to-point communication methods.

non-cached new scene graph and a cached scene graph, using the original point-to-point methods and the new broadcast method. The non-cached scene graph times includes the time to generate and communicate cache update messages to all rendering processes.

The visualization for this test contains three isosurfaces of a standard AVS test dataset. The amount of geometry generated to represent the isosurface is not important because that geometry is communicated from the `pstmpunode`(p) processes to the render processes. We are interested in the messages from the `express` process to control render process caches and the distribution of the overall scene graph.

Replacement of point-to-point methods with `MPI_Bcast` is a simple but worthwhile optimisation. This change will benefit AVS/Express DDR on other platforms, not just HECToR, and has been submitted to AVS for inclusion in their source tree.

# 6 Example Dataset

A scientifically significant dataset has been provided by the Materials Science users with which we are working. At this time we have been requested not to publish any images of that dataset until their work has been considered for publication elsewhere (it is being submitted to a high-impact publication). The images are considered key to the impact and novelty of the paper. We intend to update this report with images once they have been published. However, details of the size of data set and performance figures can be given.

The dataset is a uniform volume of dimensions $7150 \times 7150 \times 7369$ containing byte data (density values in the range 0–255). The dataset is to be volume rendered and the large size of dataset makes it a useful test case because the parallel volume render module is particularly memory hungry. Each `pstmpunode`(m) render process takes a copy of the sub-domain from the `pstmpunode`(p) process (which is typically only executing the parallel file reader code). It also

allocates another volume that is 1/32 the size of the volume for voxel lighting calculations. For gigabyte volumes this can be significant. It is memory usage that dictates how many processes are needed to render the dataset. Volume rendering takes place at image resolution and so performance is mainly influenced by the size of the final image and whether the transfer function produces semi-transparent regions in the data (this increases the execution time of the volume render algorithm). However, the AVS volume renderer does not tile the image and so adding more processors does not necessarily improve frame rate. It does, however, reduce the size of sub-domain of data that each `pstmpunode` render process will have to store.

| | $512^2$ | $1024^2$ | $512^2$ | $1024^2$ |
|---|---|---|---|---|
| Domains | 127 | | 255 | |
| Total procs | 255 | | 511 | |
| mppnppn | 2 | | 4 | |
| GB per Domain | 2.8 | | 1.4 | |
| Build+Distrib(s) | 0.044900 | 0.045211 | 0.186890 | 0.050382 |
| Render(s) | 0.469087 | 0.695153 | 0.201461 | 0.672819 |
| Total(s) | 0.513987 | 0.740364 | 0.388351 | 0.723201 |
| Frames per sec | 1.9 | 1.3 | 2.5 | 1.3 |

Table 1: Volume rendering a 351GB dataset for two image sizes. The number of domains used is the minimum number required to volume render this dataset for the given `mppnppn` value.

Table 1 shows various statistics obtained when rendering the volume dataset. Rendering was performed for image sizes of $512^2$ and $1024^2$. The number of domains in which to divide the data is the minimum number required to volume render this data. Dropping down to a smaller queue size (and hence fewer sub-domains) resulted in the `pstmpunode` rendering processes running out of memory. The times given are for operations performed by the `express` process on the login node. The Build+Distrib operation is the construction of the scene graph and distribution to all rendering processes, as discussed in section 5. The render time is the total time taken from sending out the scene graph to receiving a composited image back from the render processes. The render processes will have rendered their data during this operation so it is dependent on the AVS parallel volume render module. The total time is the sum of the two times. The Frames per second time is the best time that can be achieved when rendering the data if no transfer of the image to the user's remote desktop is required.

Despite the low frame rate we are able to manipulate the visualization interactively. This includes manipulation of the volume render transfer function to reveal details within the volume. We have not been able to render this dataset prior to running AVS/Express on HECToR.

## 6.1 Batch Rendering

It is possible to use AVS/Express to render image without interacting with the application user interface. A *write_image* module can be used to capture the image from the renderer. The application can be started with the `-offscreen`

flag to prevent the user interface appearing. However AVS/Express still requires an X server to connect to even if no interface windows are visible. This means that the user cannot disconnect from HECToR once a batch rendering job has been submitted. This is a known problem in AVS/Express that has not been addressed in this project due to the extensive source code changes required to remove all X11 dependencies in offscreen mode.

# 7 Conclusion

This project has ported the AVS/Express DDR commercial visualization code to the Cray XT4 architecture provided by the HECToR service. Using this platform we have visualized data acquired by CT X-Ray scanning that is not able to be rendered on available GPU hardware. The system allows the user to interact with the visualization, often at an approximate rate of 5 frames per second. This is just about sufficient for interactive manipulation of the visualization. The inclusion of a new compositor layer and improvements to the parallel rendering code have allowed AVS/Express to be used with processor counts much higher than any previous installations of the product.

The use of the MPI forwarding layer and proxy has eliminated the need for a significant redevelopment of the communication code within AVS/Express, allowing the communication between the parallel module and rendering processes to remain within the vendor MPI domain. This allows users to develop their own AVS parallel modules using the existing AVS framework and move them without modification to the HECToR installation.

The ability to create custom applications with AVS/Express is one of its key strengths and will allow domain-specific applications to take advantage of its parallel renderer on HECToR. There are possible uses of AVS/Express in computational steering applications, where in-place visualization of large datasets is needed. Additionally simulation code may be developed as AVS parallel modules, allowing direct visualization of the simulation data.

An example of a user-developed application is the ParaFEM Viewer [17] used to visualize finite element analysis data generated by the ParaFEM library [18] available on HECToR. The viewer is an AVS/Express application with a custom user interface. It currently uses the standard AVS *OpenGL* renderer, which does not offer any parallel rendering facilities. Given the size of dataset that can be generated by FEM code it would be useful to build the ParaFEM Viewer application against the AVS/Express DDR product on HECToR. This would give the viewer access to the parallel renderer within DDR. Work would be needed to modify the application so that it used the AVS parallel modules rather than the standard serial modules. There are several custom modules used within the ParaFEM Viewer application and so these would need to be modified to use the parallel module framework.

The XPMT library and the 2-3 Swap compositing implementations can be used in other projects (most usefully in visualization applications) where the user interface is run as part of an MPI job. Non-AVS custom viewer applications (possibly small, lightweight viewers for specific data formats) can be developed where the user interface process runs on the login node and communicates with backend rendering processes. The developer can use MPI for all communication, avoiding the need to develop a non-MPI communication layer

for user interface communication to the backend nodes. The 2-3 Swap code allows backend rendering processes to composite their images to form a final complete image.

We are hoping to acquire more datasets once the Diamond Light Source I12 beam line is operational. As imaging hardware improves we expect to see larger datasets becoming available.

Aspects of the work detailed in this report have been accepted for publication in *Proceedings of the Theory and Practice of Computer Graphics Conference 2010 (EGUK)* and have been submitted for publication to the *Philosophical Transactions of the Royal Society A* as part of the All Hands Meeting 2010. The work has also been presented at the JISC vizNET 2009 Conference [19].

# 8 Acknowledgements

The authors would like to thank James Perrin for his advice on the AVS/Express Parallel Edition and to Mark Mason of AVS UK for his support. We also thank Prof. Philip Withers, Dr. Paul Mummery and Dr. Phil Manning for their support and access to facilities and data.

# References

[1] AVS website `http://www.avs.com`

[2] Haber R., McNabb D.: Visualization idioms: A conceptual model for scientific visualization systems. In *Visualization in Scientific Computing* (1990), Shriver B., Neilson G., Rosenblum L., (Eds.) IEEE Computer Science Press, pp. 74–93.

[3] Molnar S., Cox M., Ellsworth D., Fuchs H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4), July 1994, pp. 23–32.

[4] MesaGL: Open source software OpenGL implementation. Website `http://www.mesa3d.org`

[5] OpenGL website `http://www.opengl.org`

[6] Henry Moseley X-Ray Imaging Facility website `http://www.materials.manchester.ac.uk/research/facilities/moseley/`

[7] Network Common Data Format `http://www.unidata.ucar.edu/software/netcdf`

[8] Hierarchical Data Format `http://www.hdfgroup.org/HDF5`

[9] ParaView website `http://www.paraview.org`

[10] Bethune, I. Parallel Visualization on HPCx. Tech Report, STFC, 2009. `http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0803.pdf`

[11] Moreland K., Rogers D., Greenfield J., Geveci B., Marion P., Neundorf A., Eschenberg K.: Large scale visualization on the Cray XT3 using ParaView. In Cray User Group (May 2008), Shriver B., Neilson G., Rosenblum L., (Eds.). http://www.cs.unm.edu/~kmorel/documents/PVCrayXT3/PVCrayXT3.pdf

[12] Eduserv CHEST Agreement website http://www.eduserv.org.uk/licence-negotiation/agreements

[13] Paracomp Open Source Compositor Library, HP. http://paracomp.sourceforge.net

[14] Stompel A., Ma K.-L., Lum E., Ahrens J., Patchett J.: SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003) IEEE Press, pp. 33–40.

[15] Yu, Hongfeng and Wang, Chaoli and Ma K.-L.: Massively parallel volume rendering using 2-3 swap image compositing. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (2008) IEEE Press, pp. 1–11.

[16] Ma K.-L., Painter J. S., Hansen C. D., Krogh M. F.L: Parallel Volume Rendering Using Binary-Swap Compositing. In *IEEE Computer Graphics and Applications*, 14(4), 1994, pp. 59–67.

[17] ParaFEM Viewer website http://wiki.rcs.manchester.ac.uk/community/Projects/ParaFEM-Viewer

[18] ParaFEM Library website http://www.rcs.manchester.ac.uk/research/Parafem

[19] vizNET 2009 Conference website http://www.viznet.ac.uk/viznet2009/