HPCx
CAPABILITY COMPUTING

Parallel Visualisation on HPCx

I. Bethune
EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, UK

April 30, 2008

Abstract

In order to visualise very large data sets it is desirable to use a parallel visualisation tool to obtain results more quickly than would be possible using conventional serial visualisation tools. A comparison of some current available parallel visualisation tools is made, and the tools now available on HPCx are described along with examples of how they can be used.

This is a Technical Report from the HPCx Consortium

© HPCx UoE Ltd 2008

Neither HPCx UoE Ltd nor its members separately accept any responsibility for loss or damage from the use of information contained in any of their reports or in any communication about their tests or investigations.

Contents

1 Introduction **2**
1.1 HPCx 2

2 Parallel Visualisation Tools **2**
2.1 ParaView 3
2.1.1 Mesa 3
2.1.2 Qt 4
2.1.3 Visualisation Toolkit (VTK) 4
2.1.4 Python 4
2.2 VisIt 4
2.3 OpenDX 5

3 Parallel Visualisation on HPCx **5**
3.1 Interactive Rendering 5
3.1.1 Comparison of connection methods 7
3.1.2 Scalability 8
3.2 Batch Rendering 9

4 Summary **11**

5 Acknowledgements **11**

1 Introduction

This report explores the rationale behind parallel visualisation and reviews some of the currently available tools (Section 2). ParaView [2] is covered in greatest detail as it has been chosen to be installed on HPCx. The main underlying technologies are also described.

Section 3 describes how to use ParaView on HPCx, and compares the performance for a range of configurations and problem sizes.

1.1 HPCx

A fuller overview of the architecture of HPCx (Phase 3) as of April 2008 can be found on the HPCx Website [1]. For the purposes of this report it is sufficient to know that it is made up of 160 compute nodes for batch processing, each of which is a 16-way SMP of IBM Power5 processors, for a total of 2560 CPUs. There are also 2 nodes (32 CPUs) available for interactive jobs run from the login node. Executables may also be run on the login node but this should be avoided as its resources are shared by all logged in users. Each node has 32 GB of main memory (2 GB per CPU for a fully utilised node). Inter-node communication is via IBM's "Federation" High Performance Switch network.

2 Parallel Visualisation Tools

Parallel visualisation has similar a goal to High Performance Computing (HPC) generally, that is, to allow larger problem sizes to be tackled, and to do to so within a feasible time. Applied to visualisation, the aim is to enable very large (10^8 data points or more) data sets to be drawn within a few seconds. Additionally, in the case of visualisation, a user might well wish to interact with the visualisation of their data, for example rotating, zooming, or otherwise altering their viewpoint of the data, before saving a final image to disk. Often, the user might wish to view only a subset of their data, so the visualisation tool should provide methods for filtering out unneeded data, for example by allowing slices through a data set, iso-surfaces to be calculated, or by discarding data points based on user-defined criteria. It is also desirable for users to be able to change the representation of the data, such as the way colours are mapped to the data, or transparency is applied.

Standard data formats exist such as NetCDF [4] and HDF [5] that might be used by scientific codes. However, some codes will use less well-known formats, or their own proprietary data formats. To allow broad applicability of a parallel visualisation tool it should support as many formats as possible, and be extensible to user-defined formats e.g. by a plugin system.

At a high level the tasks performed by a visualisation tool are reading in a data set, performing some transformations on the data, rendering the data to an image, and saving that image to disk. In each of these stages there is scope for parallelism, and the tools investigated each take different approaches to exploiting this.

2.1 ParaView

ParaView is an open source application, the result of collaboration between Kitware Inc.[3], Sandia National Laboratories, Los Alamos National Laboratory, CSimSoft, Army Resarch Lab and others. The latest stable version, ParaView 3.2.1, released in Nov 2007 is used as the basis for this investigation.

ParaView has a modular, extensible architecture. It is written mainly in C++ and has been run on many platforms including Windows, Mac OS X, Linux and UNIX. Notably, not all of the components necessarily have to run on the same platform, as we shall see.

The main components that make up ParaView are the Client, the Data Server, and the Render Server.

The Client provides a GUI by which the user controls the visualisation. Only one Client process will exist as interaction with the user is an intrinsically serial operation. Unless connected to a user-specified server, the Client will start a local server and connect to it instead.

The Data Server is responsible for reading the data from disk and applying transformations to the data to prepare it for rendering. The Data Server can be run in parallel using MPI for interprocess communication. Each instance of the data server performs logically the same operations on the data, but will operate on a subset of the total data set. ParaView handles the distribution of the data to each process, including creation of 'ghost' or 'halo' data, automatically, although after the application of some filters, the data distribution can become unbalanced, and can be restored by the application of a 'Distributed Data Decomposition' filter.

The Render Server takes a representation of the original data from the Data Server and generates an image, which is then sent back to the client for display. The Render Server can also be run in parallel using MPI, and may have a different number of processes to the Data Server, for example if a specific number of nodes have graphics hardware attached. This is not the case on HPCx, and as we will see later, it is possible to launch the Data Server and Render Server together, in which case they are collectively referred to as the Server. It is generally recommended to launch the Data and Render Servers together as this avoids the communications latency between the processes. For a parallel Render Server, each process renders it's own subset of the data and the sub-images are composited by the client to form the final output image. Communications between seperate components are done via TCP/IP socket connections, as illustrated in figure 1.

ParaView is built on top of several existing libraries and toolkits, which are detailed below.

2.1.1 Mesa

Mesa[6] is an open source implementation of the OpenGL[7]. It provides the ability to render 2D and 3D graphics in software where dedicated graphics hardware is not available, for example on the compute nodes of HPCx. Furthermore, it can do rendering in an offscreen memory buffer, rather than rendering directly to a window framebuffer or backbuffer, which is required since not every node being used as a Render Server will necessarily have access to an X Server on which to open a window.

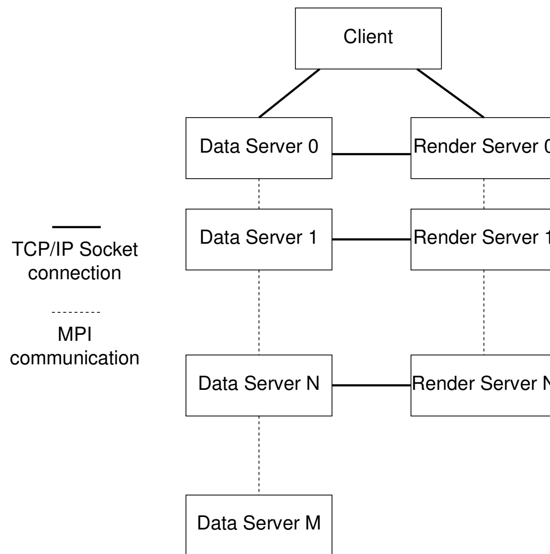

Figure 1: ParaView component architecture

2.1.2 Qt

ParaView's GUI is built using the Qt[8] open source, cross-platform application development library. This has enabled ParaView to be easily ported to several platforms, including Windows, Linux, Mac OS X, and various UNIX distributions. Using Qt, it is also possible to modify and extend the ParaView GUI.

2.1.3 Visualisation Toolkit (VTK)

VTK[9], from Kitware, is the an open source toolkit for transforming, combining, and visualisating data. ParaView can be considered a wrapper over the underlying VTK infrastructure, adding parallelism using MPI, the client-server architecture, a scripting interface and a large number of file format readers. The underlying work of applying filters and rendering data is done by VTK.

2.1.4 Python

Python[10] is a very-high-level programming language. It is object oriented, and is extensible via a module interface. ParaView 3 uses Python as a scripting language, and provides access to ParaView objects and commands (methods) via the `servermanager` module. Python commands can either be entered interactively through the `pvpython` interpreter or batch processed using `pvbatch`.

2.2 VisIt

VisIt[11], developed by the US DoE Advanced Simulation and Computing Initiative(ASCI), is in many ways similar to ParaView. It is composed of a GUI and Viewer that provide control over the visualisation and output to screen. There is also a Database Server for reading files and a Compute Engine for transforming and rendering the data. These may

reside on a different machine to the GUI and Viewer, and communication is done via a socket connection. The Compute engine can be run in parallel using MPI. Rendering can either be performed by the Compute Engine(s) and resulting images sent back to the Viewer, or the geometry itself can be sent to the Viewer and the rendering done locally.

After some experimentation with VisIt on HPCx, it was abandoned in favour of ParaView for several reasons:

- ParaView provides more flexible methods of making connections between local and remote components.
- ParaView provides a configurable built-in reader for data files containing raw binary array data, whereas VisIt requires a hand-coded module to be written.
- Building VisIt on HPCx was problematic as the correct options for 64-bit AIX were not available in the `build_visit` script and a successful build of VisIt with parallel support was not made.

2.3 OpenDX

OpenDX[12], is an open source project based on IBM's Visualisation Data Explorer. It provides sets of modules which represent transformations such as isosurface generation, and colouring. Modules can be linked together to create a visual program, which is then executed to generate a final image. Parallelism is supported in two ways. Firstly, an individual module may be executed in parallel in shared memory using pthreads. Seperate modules may also be executed on seperate machines, with communication over TCP/IP. For example, computationally intensive modules could be run on a powerful CPU, while rendering tasks could be run on a specialist graphics machine, or the user's workstation which might have a GPU.

OpenDX can open files from a number of file formats via libraries which can be optionally compiled in. It also has a generic loader where the user can describe the file format, e.g. number of data points, endian format, data types.

OpenDX was not chosen to be installed on HPCx primarily because the code was found to be prone to crashing, and very little further investigation could be done.

3 Parallel Visualisation on HPCx

This section describes the ParaView installation on HPCx and instructions on how to use it. ParaView is installed in `/usr/local/packages/paraview`. Both serial and parallel versions are available. Further instructions are available in the HPCx user guide [13]

3.1 Interactive Rendering

The client and server components of ParaView can be run anywhere provided a TCP/IP socket connection can be opened between the two components. Ideally, the client should be run on your workstation, and the server run in parallel on HPCx. There are binaries available from the ParaView website[2] for Windows, Linux and Mac OS X. If it is not possible to run the client locally, the client can be run on the login node of HPCx itself, but this is not recommended as the login node is a shared resource, and it also requires

a connection to an X Server which can cause the GUI to perform sluggishly. For a comparison of the various connection methods see section 3.1.1. Furthermore, the X Server is required to support the GLX extension. This extension can be obtained by installing Hummingbird Exceed3D or Cygwin XWin on Windows. To check that your X Server is compatible, run `xdpyinfo | grep GLX`.

To start the paraview client on the login node run the script:

```
/usr/local/packages/paraview/launch_paraview_client
```

To allow the client-server communication to take place, users should use ParaView's reverse connection method. This allows the server to initiate the connection back to the client and works around two difficulties. Firstly, because it is not known at job submission time which HPCx node(s) the server will run on, it is not possible to specify an IP address for the client to attempt connection with. Secondly, HPCx has a firewall which would block inbound communication attempts from the client. Depending on where the client is run, there may be also be a firewall around it, and it is up to the user to allow inbound connections from HPCx, either by adding a firewall rule to allow the connection, or by using port forwarding or VPN to allow communication. By default, all communication will be on TCP port 11111, unless otherwise specified. Currently, it is not possible for the compute nodes to make a network connection to the client if it is not running on the login node. To overcome the connection can be forwarded across the login node to the remote IP address where the client is running. This is accomplished by the `launch_paraview_server` script using the `proxycontrol` tool.

To start the server, first ensure the client is running and ready to accept a connection using the "client-server reverse connection" mode. Then start the paraview server using the script:

```
/usr/local/packages/paraview/launch_paraview_server -l <llfile>
```

The `<llfile>` specified should be a LoadLeveller script for an interactive job. An example file from the HPCx User Guide [13] is:

```
#@ job_type = parallel
#@ job_name = hello
#
#@ cpus = 5
#
#@ node_usage = shared
#
#@ wall_clock_limit = 00:10:00
#@ account_no = z001
#
#@ notification = never
#
#@ class = inter32_1
#
#@ queue
#
```

The full range of options can be accessed by passing `-h` to the script. Users wishing more control over the launching of the server should read the script and run the relevant executables directly.

The LoadLeveller script should specify a number of processors (up to 32) for an interactive job. An appropriate wall clock limit should be specified, because if the limit is reached, the server processes will be terminated immediately and any visualisation data that has not been saved to disk will be lost.

3.1.1 Comparison of connection methods

Because of the variety of possible configurations and connection methods between the client and server several methods were tested and compared. For this purpose the time taken to render a still image of 2 iso-surfaces through a 3 dimensional regular lattice of scalar data points was measured using ParaView's inbuilt timer system. The data set was sampled at a number of sizes, from 32x32x32 up to 128x256x128 and the time taken to perform a still render was recorded at each size in each configuration as shown in figure 2. All measurements were made with a single processor (except in the case of the workstation), so the results should be used to compare the overhead of the different connection methods. For a discussion of performance on multiple processors, see section 3.1.2.

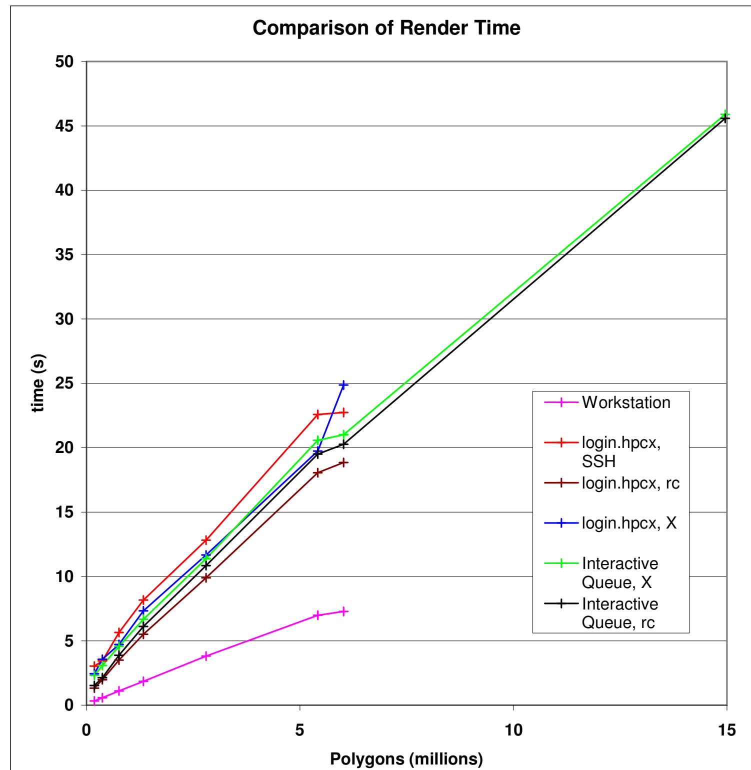

Figure 2: Comparison of time taken to render a pair of iso-surfaces through a data set on 6 different configurations

The configurations are as follows:

- Workstation: Client and server are run on a local Windows workstation.
- login.hpcx, SSH: Client is run on workstation, server is run on HPCx login node, with communication port-forwarded over SSH
- login.hpcx, rc: Client is run on workstation, server is run on HPCx login node, with communication direct over TCP/IP
- login.hpcx, X: Client and server are run on HPCx login node, using X session forwarding over SSH
- Interactive Queue, X: Client is run on HPCx login node using X session forwarding over SSH, server is run on the interactive queue
- Interactive Queue, rc: Client is run on workstation, server is run on interactive queue with communication fowarded across the login node direct to client over TCP/IP

In this test, the fastest configuration is when the server and client are run on a workstation, in this case a Intel Core 2 1.86GHz system. This is not surprising as the workstation has 2 CPU cores, as well as a GPU, compared to a single core on HPCx. However, as shown in section 3.1.2, it is possible to outperform this using 4 or more processors on HPCx. Also, as the number of processors is scaled out on HPCx, the available memory also increases, allowing larger systems to be visualised.

The configurations where the server is run on the login node of HPCx and in the interactive queue (single process only) are very similar. Some performance is lost by running on the interactive queue due to the extra network latency between the compute nodes and the login node, but it is not significant especially as typically multiple processes would be used. However, the methods utilising a direct socket connection consistently outperform those using a forwarded X session.

Out of consideration for other users of the system it is preferred not to run the client on the login node, but depending on any local firewall at the user's site it may not be possible to make a direct socket connection. Methods such as SSH forwarding or VPNs may then be used, however this will incur a larger performance penalty as all communication will be encrypted and decrypted.

3.1.2 Scalability

Since an individual node of HPCx does not have any dedicated graphics hardware and can be out-performed by a desktop workstation, the intention is to improve performance by applying more processors to the task. To investigate how well the performance does scale as the number of nodes applied to the task increases the same data set was used as for the comparison of connection methods and the same still render was performed for a several data set sizes and numbers of processors. For each test, the speedup was calculated by dividing the time taken for one processor to perform the task by the time taken for N processors. Results are detailed in figure 3.

These results indicate that as more data is given to each processor, the speedup approaches ideal. For the smaller data sets, the potential speedup acheived by adding more processors is limited.

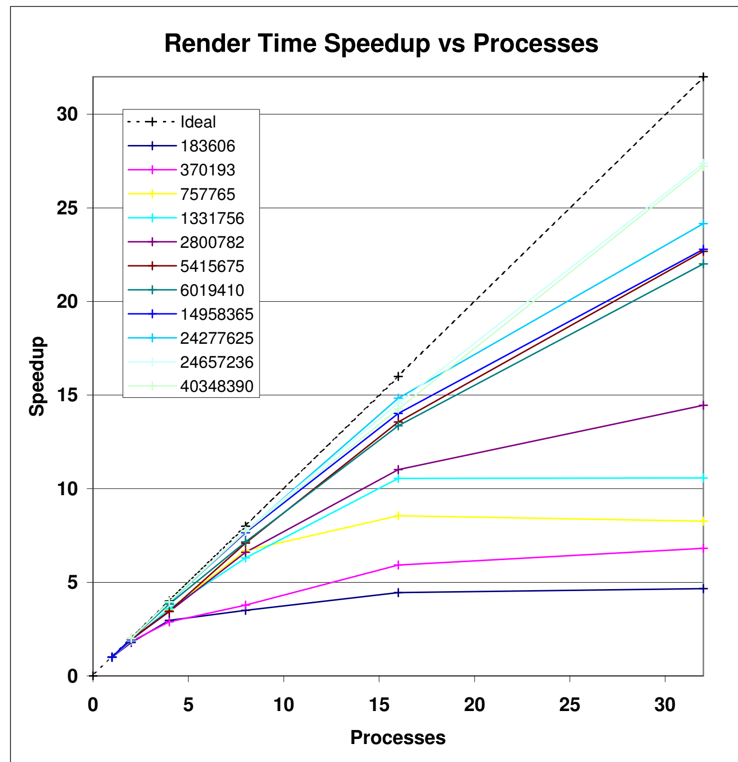

Figure 3: Speedup against number of processors for a variety of data set sizes (measured in number of polygons rendered)

As mentioned earlier, one other benefit of scaling up the number of processors is the ability to render larger data sets. For example, in this case the largest three data sets ($>$ 2 million polygons) required two or more processors to provide enough memory to store the data.

Aside from this graph, the workstation measured in section 3.1.1 was around 3-4 times faster than a single processor on HPCx. From figure 3 we can see that it takes four HPCx processors to exceed this performance for all but the smallest jobs.

3.2 Batch Rendering

NB. Due to known issues in the 3.2.1 release when using the batch interface ParaView 3.3.0 or later must be used

As well as the interactive client-driven rendering described above, ParaView also provides a Python[10] scripting interface, which can be used to generate visualisations without using the client GUI. For example, if there are a large number of data files, all of which require the same operations to be performed, this would be a good candidate for automation via a python script.

All the ParaView-specific functionality is accessed through the `servermanager` Python module. Further details on this can be found by entering `help(servermanager)` while in a python interactive session, or on the ParaView web site [14].

There are two python executables in ParaView. Firstly, `pvpython` is a python com-

mand line interpreter, which can be run on its own or be accessed through the Tools menu in the ParaView GUI. It can be used for writing, testing and debugging scripts for later use with the batch interface. Secondly, `pvbatch` takes a python script and runs it without any interaction from the user. `pvbatch` can be run in parallel on the interactive or batch queues on HPCx. For example the command to run a batch script is:

```
/usr/local/packages/paraview/3.3.0/serial/bin/pvbatch
 --use-offscreen-rendering <path-to-python-script>
```

In parallel, the following could be included in a loadleveller script:

```
poe /usr/local/packages/paraview/3.3.0/parallel/bin/pvbatch
 --use-offscreen-rendering <path-to-python-script>
```

When running in parallel, be aware the following error message will be displayed: `vtkXOpenGLRenderWindow (1131a7750):  bad X server connection.  DISPLAY=`. This is normal and will be removed in a future release.

4 Summary

ParaView 3.2.1 and 3.3.0 (development version) are now installed on HPCx and available for users to perform a wide range of visualisations utilising the parallel capability of HPCx. The launch scripts and executables are in `/usr/local/packages/paraview` and are accessible to all users. ParaView has been demonstrated to scale well up to 32 processes on the interactive queue, and for very large or repetitive visualisation tasks, the batch queues can be used via ParaView's Python scripting interface.

5 Acknowledgements

I am grateful to Dr. Kevin Stratford and the Edinburgh Soft Matter Project for providing the data files used in the production of this report. Dr. Stephen Booth and Dr. Kenton D'Mellow both provided useful advice on port forwarding. Dr. Alan Gray provided comments and corrections prior to publication.

References

[1] HPCx web site, http://www.hpcx.ac.uk

[2] ParaView web site, http://www.paraview.org

[3] Kitware web site, http://www.kitware.com

[4] Network Common Data Form web site, http://www.unidata.ecar.edu/software/netcdf

[5] Heirarchical Data Format web site, http://www.hdfgroup.org

[6] The Mesa 3D Graphics Library web site, http://www.mesa3d.org

[7] OpenGL web site, http://www.opengl.org

[8] Trolltech Qt web site, http://trolltech.com/products/qt

[9] The Visualization Toolkit web site, http://www.vtk.org

[10] Python Programming Language – Official Website, http://www.python.org

[11] VisIt web site, https://wci.llnl.gov/codes/visit/

[12] OpenDX web site, http://www.opendx.org

[13] HPCx User Guide, http://www.hpcx.ac.uk/support/documentation/UserGuide/HPCxuser/HPCxuser.htm

[14] ParaView Scripting with Python, http://www.paraview.org/Wiki/images/f/f9/Servermanager2.pdf