Optimal Visualization of Scientific Data

Yuzhe Zhou
S1471339

5th May 2016

1 PERSONAL STATEMENT

The basis for the following work started with a meeting between myself and Dr Donghyuk Shin during which he explained the prospect of big data visualization and the lack of corresponding investigation on visualization performance in the University of Edinburgh high performance computers. As a new project starting from scratch, I was willing to accept this challenging project. Following a literature survey and meetings with Dr Shin, it was proposed to finish a big data scientific visualization video using parallel I/O lustre filesystems and compute nodes on ARCHER. I finished ARCHER driving test myself and did a quick study of Linux command lines as preparation. During this time, Dr Shin gave me patient instructions and guidance. Video examples and MATLAB scripts to manipulate 2D datasets were provided by Dr Shin to help me to get a quick understanding and be familiar with procedures. In project interim evaluation, the progress was assessed and after discussing with Dr Shin and Prof Jason Reese I realized the project was ambitious and the results were difficult to be finished entirely due to the time limitations of the project. Therefore, the focus of the project was adjusted and narrowed on investigating and optimizing the parallel rendering performance and finishing a 3D flow visualization material. It was proposed to conduct a strong scaling test to evaluate the visualization performance. All scaling performance benchmarking and testing were conducted by myself. ParaView Client/Server configuration was finished by myself with the help of Dr Shin. All results and comments were created by myself. All the 3D render work was finished originally by myself. The discussion points were all my own ideas except where labelled with references. Dr Shin gave me his advices on the discussion points during meetings about how to structure them in the report. The further work was my proposals based on my experiences in the project. Overall the targets of were effectively completed with an analysis of render performance and a 3D flow visualization material. I feel I have contributed to the optimal visualization on Data Analytic Cluster at Edinburgh and really hope the research on ParaView optimal visualization could move forward for the next BEng project. I have learnt a great deal from this BEng project not only technically about this subject but also the art and training of an effective research.

Yuzhe Zhou 5th May 2016

2 SUMMARY

Optimal Visualization of Scientific Data
Yuzhe Zhou – Thursday 5th May 2016

Scientific visualization is an indispensable tool to create insight from a mass of data resulting from constantly complex numerical models. Visualizing such massive amounts of datasets is a technique-demanding job which requires the support from high performance computers and the structure of parallel computing. As a powerful visualization tool, ParaView was chosen to carry out the project and conduct the render performance tests.

A literature survey was conducted and found that the process of parallel visualization relied on parallel I/O and parallel rendering. Threes modes of ParaView were also introduced and Client/Server mode was used in the project. The literature survey also introduced the data structure of the dataset used in the project and two modes of rendering, interactive rendering and still rendering.

The overall aims of the project were to carry out a render scaling test and investigate the parallel rendering performance of the tests. Moreover, a 3D flow visualization animation was produced based on the experiences from the previous testing.

The test was prepared and conducted on Data Analytic Cluster located at the University of Edinburgh. As a start basis, ParaView server was built step by step. Then a strong scaling test was prepared. The size of input file was fixed and the number of processors were increased during the test. Timing information for each case were collected from ParaView timer log function. The results did not indicate obvious improvement in the I/O and still render efficiency as the number of cores increases. It was then analysed and discussed to find that the serial file I/O and the communication sockets between client and server could cause a bottleneck on the performance. So another test was run on a standalone version of ParaView to verify the assumption. The modified results showed a clear improved render speedup and an approximate speedup of 12 could be achieved when 64 cores were in use, which was thought as a not bad performance.

A 3D flow animation was produced and saved in a USB drive. Crashing problems happened during the process. It was then found that the settings of rendering parameters and data cached in the buffer were two reasons causing the crash problems. An investigation was conducted to solve the problems and the animation was finished finally.

Overall parallel visualization is very complicated to be carried out, which relies on systematical knowledge and experience. The project has successfully contributed to investigation and optimization of parallel rendering on Data Analytic Cluster. And it could be a good starting point to conduct the further investigation on parallel I/O and off-screen rendering in the future work.

3 AKNOWLEDGEMENTS

I would like to extend my sincere gratitude to all those who helped me with this projects:

- To my supervisor, Dr. Donghyuk Shin, for, above all, his patience, and his help, guidance, advice and resources he supplied me with over this BEng individual project.
- To my wonderful parents, for their generous love and support and the opportunity for me to finish the study at Edinburgh.
- To my fabulous friends, for their selfless care and help to encourage me and let me get out of the depressing period.
- To my great roommates, for their time spent with me to make me enjoy the life and be strong enough to face the challenges from the study.
- To the university, for their supportive help in the facilities and services to provide strong technique support for the project. This work used ARCHER UK National Supercomputing Service (http://www.archer.ac.uk). This work used the UK Research Data Facility (http://www.archer.ac.uk/documentation/rdf-guide).
- Last, but by no means least, to my grandparents for their mental support to make me stay optimistic about the difficulties in study and life.

4 TABLE OF CONTENTS

5 INTRODUCTION

Scientific visualization is an indispensable tool to create insight from a mass of data resulting from constantly complex numerical models. By illustrating the scientific data graphically, scientific visualization could enable researchers to understand, illustrate and glean insight from their data [1]. Interest in scientific visualization has increased in recent years largely due to its use in a wide range of applications including: natural science (e.g. molecular rendering), geography (e.g. terrain visualization) and applied science (e.g. aircraft plot) [1].

Visualizing such massive amounts of datasets is a technique-demanding job which requires high performance computer systems. Core concepts to improve visualization efficiency are the integration of parallel computing, including parallel I/O (Input/Output) and parallel rendering [2]. The structure of parallel visualization is shown in Figure 1.

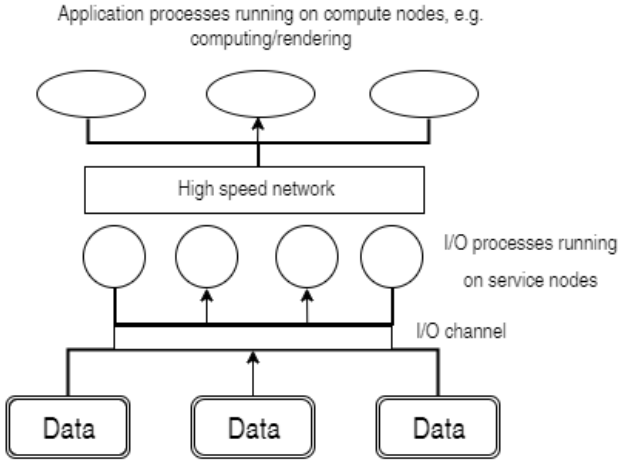

Figure 1. Parallel Visualization Structure

Parallel computing is a kind of computation where calculations are carried out simultaneously. That means that a large problem can be divided into smaller ones, which are then solved at the same time. In reality, the efficiency of parallel computing is not linear to the number of processors while money and energy consumption are proportional to the resources used[3]. Accordingly, it is necessary to make a balance and make use of computing resources to realize optimal visualization.

This project was performed on ARCHER (Advanced Research Computing High End Source) and UK-RDF DAC (Data Analytic Cluster) using ParaView as the visualization tool. Literature survey suggests that files systems are different on ARCHER and RDF and therefore they have distinct available modules (different versions of ParaView) and thus corresponding operations are different on these two cluster systems.

The focus of this project is on the parallel rendering performance during visualization. Parallel scalability tests were carried out on high performance computer (Data Analytic Cluster). The measurements of tests will indicate how efficient a visualization process is when using increasing numbers of parallel processing elements (CPUs/cores). In addition, the effect of I/O performance will be discussed based on the testing results.

The aim of this project is to finish a visualization material (flow jet, 80 GB) and provide render scaling efficiency with different computing resources. Further to this, a comprehensive investigation will be performed to suggest an optimal visualization in consideration of computing efficiency, I/O speed and corresponding cost.

6 LITERATURE SURVEY

6.1 Data Structure

To recognize the data structure is important in the visualization process, which can help users identify which parameter will be visualized and thus conduct further manipulation. 3D flow datasets used in the project consist of three formats: flow data files (RAW format), grid files and XDMF files.

Flow data files contain variable information of each point in the domain, which take up the most of storage space [4]. Flow data files store data in binary format, where the floating point numbers can be written in single (32-bit) precision or double (64-bit) precision and integers are written as 32-bit integer [5]. However, it would be an efficient way to transfer the data format from double precision to single precision to contract datasets. The flow file names are composed by the name tag, the blockid, the number of contained variables and the time step in the following form [5]:

<tag>_<blockid>_var_<#timestep>_.raw

In normal conditions, eight variables are contained in a flow file, parts of which can be defined and customized by users. The information included in a flow file is shown in the following table:

Table 1. Data information contained in raw flow file

Information	Number of values (each takes up 4 bytes)
Number of points in x direction (Nx)	1
Number of points in y direction (Ny)	1
Number of points in z direction (Nz)	1
Mach number	1
Undefined value	1
Reynolds number	1
Time Step	1
Variable 1: density	Nx*Ny*Nz
Variable 2: first velocity component (u)	Nx*Ny*Nz
Variable 3: second velocity component (v)	Nx*Ny*Nz
Variable 4: third velocity component (u)	Nx*Ny*Nz
Variable 5: temperature	Nx*Ny*Nz
Variable 6: z0	Nx*Ny*Nz
Variable 7: z1	Nx*Ny*Nz
Variable 8: a	Nx*Ny*Nz

A grid file is a point access method splitting the space into a non-periodic grid where one or more grid refers to a small set of points, which provides an efficient method to perform complex data lookups by these indexes [6]. Grid files do not contain any data information but references to the correct bucket for the data instead. The grid files are composed in the following form:

<tag>_GRID_<blockid>.xyz

The information included in a grid file is shown in the following table.

Table 2. Data information contained in a grid file

Information	Number of values (each takes up 4 bytes)
Number of points in x direction (Nx)	1
Number of points in y direction (Ny)	1
Number of points in z direction (Nz)	1
Coordinate position	Nx*Ny*Nz

XDMF (eXtensible Data Model and Format) is a standardized format to exchange scientific data between high performance computing codes and widely used visualization tools like ParaView [7]. A XDMF file serves as a path guidance to the flow data (with the whole time steps) for ParaView [5].

6.2 Client/Server Mode in ParaView

ParaView is one of the most common and powerful visualization tools in computer-vision-related fields. ParaView can be applied in multiple-processor distributed-memory supercomputers or workstation clusters [8]. Moreover, Client/Server architecture lets ParaView runs on variety of platforms from netbooks to the largest machines in the world. As an open-sourced application, ParaView comes with a Python application that allows users to automate the visualization and post-processing with Python scripting.

Achieving an interactive visualization relies on a three-tier client-server architecture, including data server, render server and client. The data server is responsible for data reading, filtering and writing [8]. All of pipeline objects in the application browser are contained in data server. The data server can be parallel. Same as the data server, the render server can also be configured on parallel applications. Client is responsible for establishing visualization and controlling. The client is always a serial application and in the project, the client ParaView is run on a personal laptop. Three running modes of the architecture in ParaView which are shown in Figure 2 [8].

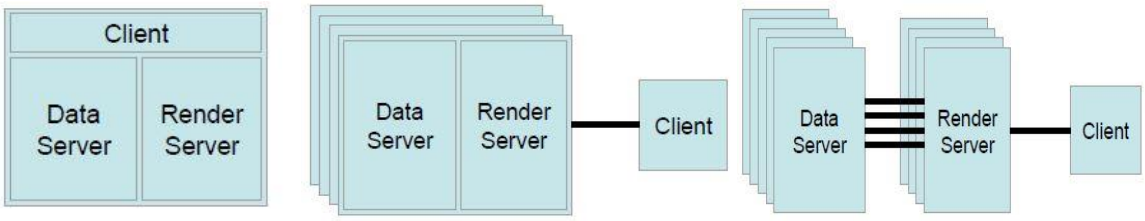

Figure2. Standalone mode, client-server mode and client-render server-data server mode (from left to right) [8]

In client-server mode, pvserver program is executed on a parallel machine and connect to it with client application interface. The pvserver program has both the data server and render server embedded in it, so both data processing and rendering take place there. In practice, data processing takes up more resources than rendering, so it is more efficient to customize the number of data serves and render servers and build up the optimized connection. However, the research from *Cedilnik et al., 2006* [9] indicates that the interaction among servers will become challenging so most projects employ client-server mode. In the project, client-render mode is chosen as the method of parallel visualization and client-render server-data server mode will be discussed in the further work section.

Message Passing Interface (MPI) is a standardized and portable message-passing system design to function on a wide variety of parallel computers [10]. MPI implemented ParaView is required for large data parallel processing. However, the large variety of hardware, operating systems and MPI implementations make it impractical for all users to employ universal precompiled binaries of ParaView [11]. On ARCHER, it is not currently possible to use pvserver interactively on the compute nodes, which means it is necessary for users themselves to compile ParaView themselves from source in MPI support. Two versions of ParaView are installed on the RDF: normal ParaView with Graphical User Interface (GUI) installed and parallel ParaView with MPI enabled (but no GUI). MPI-enabled parallel ParaView can be loaded using paraview-parallel module on the RDF DAC:

module load paraview-parallel

Paraview offers rich scripting support through Python. This support is available as part of the ParaView client, an MPI-enabled batch application (pvbatch), and the ParaView python client (pvpython). Accesses to ParaView objects and commands are provided via *servermanager* module [12]. ParaView scripting has great advantages in automating repetitive tasks and executing operations when GUI is not available.

6.3 Rendering Mode in ParaView

Rendering of large data is the process to synthesize the images based on the dataset. The efficiency of the visualization of large flow datasets is expected to depend highly on the speed of the rendering, which is proportional to the scale of data being rendered. ParaView provides two modes of rendering, still rendering and interactive rendering [13].

Still rendering is a rendering process where the data is rendered at the highest level of detail. This mode provides that the entire data are represented accurately. When interaction of 3D view is not taking place, a still rendering mode is in use to keep the full detail of the data.

Interactive rendering focus more on speed of rendering than accuracy. A quick rendering rate is finished in this mode regardless of data size. It is significantly user-friendly that a quick preview can be realized to avoid the time-consuming process of detailed rendering when users drag the mouse in 3D view [13].

In the project, the camera position and parameters of rendering are recorded in a log file before visualization performance testing. Few mouse motions will be tracked during the rendering, so the emphasis of analysis in discussion section will be put on still rendering mode.

6.4 Jobs on ARCHER and RDF DAC

ARCHER, the latest UK National Supercomputing Service, is based around a Cray XC30 supercomputer that provides the central computational resource. This 4920 node supercomputer is supported by a number of additional components including: high-performance filesystems, pre-processing and post-processing facilities, login nodes and a large data facility [14]. Technical parameters of ARCHER are shown as follows:

- 4920 compute nodes, each of which contains two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) series processors and has 64 GB of memory shared between the two processors.
- 8 external login nodes, each of which contains two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) series processors.
- Service nodes, e.g. job launcher nodes
- /work filesystems with a total of 4.4 PB storage and /home filesystem with 218 TB storage

A schematic diagram is drawn to indicate the architecture of ARCHER in Figure 3.

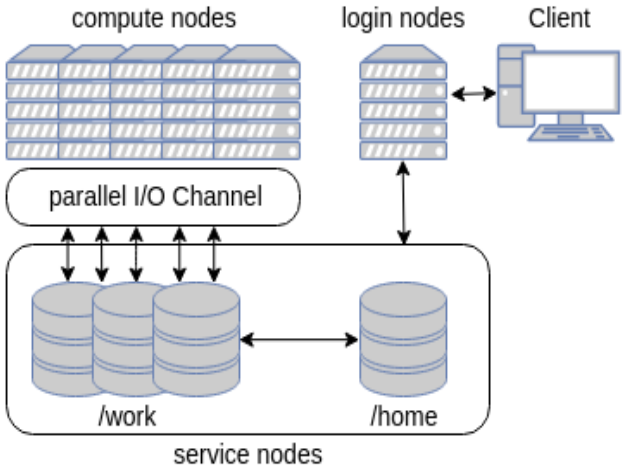

Figure 3. Architecture of ARCHER hardware

UK Research Data Facility (RDF) is collocated with ARCHER consisting of 23 PB of usable storage. Data Analytic Cluster (DAC) nodes have direct connections to the UK RDF disks [14]. Technical parameters of DAC are:

- 1 login node, each of which contains two Intel Ivy Bridge 10-core processors and 128 GB memory.
- 12 standard compute nodes each with two Intel Ivy Bridge 10-core processors and 128 GB memory.
- 2 high-memory compute nodes each with four Intel Westmere 8-core processors and 2 TB memory.

Hyper Threads are enabled on all nodes meaning that standard compute nodes each have 40 CPUs available and the high-memory compute nodes each have 64 CPUs available [15].

The work for BEng project is mainly based on UK RDF with DAC and parts of visualization experiments are conducted on ARCHER.

6.5 Parallel Scaling Performance

Scaling efficiency measurements are widely used in high performance computers to illustrate the efficiency of an application when utilizing increasing number of parallel processing elements (e.g. CPUs/cores). Two scaling tests are commonly used to measure the parallel performance, which are strong scaling test for cpu-bound case and weak scaling test for memory-bound case respectively [16].

In strong scaling test, the number of parallel processing CPUs are increased while the data size stays fixed. The goal of strong scaling to find the position where computation is finished in a reasonable time range with a balance of numbers of cores in use. Theoretically the computation is considered to scale linearly if the speedup is equal to the number of processing elements used (N) []. Strong scaling efficiency is given as:

$$\frac{t1}{N \times tN} \times 100\%$$

t1 is the amount of time to complete the computation of fixed work with one processing core. N represents the number of cores used in the computation of the same size of work and the corresponding time consumed is tN.

In weak scaling test, the number of cores and data size will be increased. The size of data to be computed stay fixed for each computation core. The execution time in each test is measured and linear scaling is achieved if execution time keeps constant while the data size is increased in proportion to the number of computation cores [17]. Weak scaling efficiency is given as:

$$\frac{t1}{tN} \times 100\%$$

t1 is the amount of time to complete the computation of fixed work with one processing core. N represents the number of cores used in the computation of the same size of work and the corresponding time consumed is tN.

In general, strong scaling test aims to improve the computation to run faster while weak scaling test puts emphasis on computation on larger scale of problems. The efficiency test in the project focuses on strong scaling test to investigate parallel rendering efficiency.

7 TECHNICAL PROCEDURE

7.1 Build Client/Server

Two versions of ParaView are installed on the RDF: normal ParaView with Graphical User Interface (GUI) installed and parallel ParaView with MPI enabled (but no GUI). RDF does not have a specific server for ParaView, so a job needs to be submitted to request some resource on RDF. Steps to launch ParaView server is as following:

1) Connect to RDF using ssh:

```
yuzhe@zhoupc:~$ ssh zhou@login.rdf.ac.uk
```

2) Load ParaView with MPI enabled version:

```
[zhou@rdf-comp-ns10 zhou]$ module load paraview-paralell
```

3) Submit a job to request sources from a queen. The RDF facility uses PBS (Portable Batch System) to schedule jobs:

```
[zhou@rdf-comp-ns10 zhou]$ qsub –q hm04 yuzhe.pbs
```

PBS file is written in following format:

```
#!/bin/bash –login

#PBS -N flow-visualization
#PBS -l ncpus=4
#PBS -l walltime=1:00:0
#PBS -l select=2

export PBS_O_WORKDIR=$(readlink -f $PBS_O_WORKDIR)
cd $PBS_O_WORKDIR
mpirun -n 8 pvserver --mpi > output.${PBS_JOBID}.out
```

4) Name of this project is “flow-visualization”. Two nodes each with four processors are requested in this job. Walltime is the maximum time that the project will be run, which means the job will automatically be killed after one hour (walltime in abovementioned PBS file). Mpirun pvserver will launch the ParaView server in parallel using the resources.

5) Use ouput file to check which node is being use for ParaView server. An example is given as:

```
Waiting for client…

Connection URL: cs://rdf-comp-hm04:11111
Accepting connection (s): rdf-comp-hm04:11111
```

6) Establish a tunnel between local computer and RDF DAC by the following command:

```
yuzhe@zhoupc:~$ ssh –L 11111: rdf-comp-hm04:11111 zhou@login.rdf.ac.uk
```

Established port can be checked by running [18]:

```
yuzhe@zhoupc:~$ netstat -lnt
```

Figure 4 indicates that a new port address 127.0.0.1:11111 has been established.

```
yuzhe@zhoupc:~$ netstat -lnt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:11111         0.0.0.0:*               LISTEN
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp6       0      0 ::1:11111               :::*                    LISTEN
tcp6       0      0 ::1:631                 :::*                    LISTEN
yuzhe@zhoupc:~$
```

Figure 4. Internet Connection Lists

7) Connect to ParaView server on client side. A successful client-server mode of ParaView is shown in Figure 5, where the left box contains information of port address while the right box (memory inspector) indicates how many cores are in use.

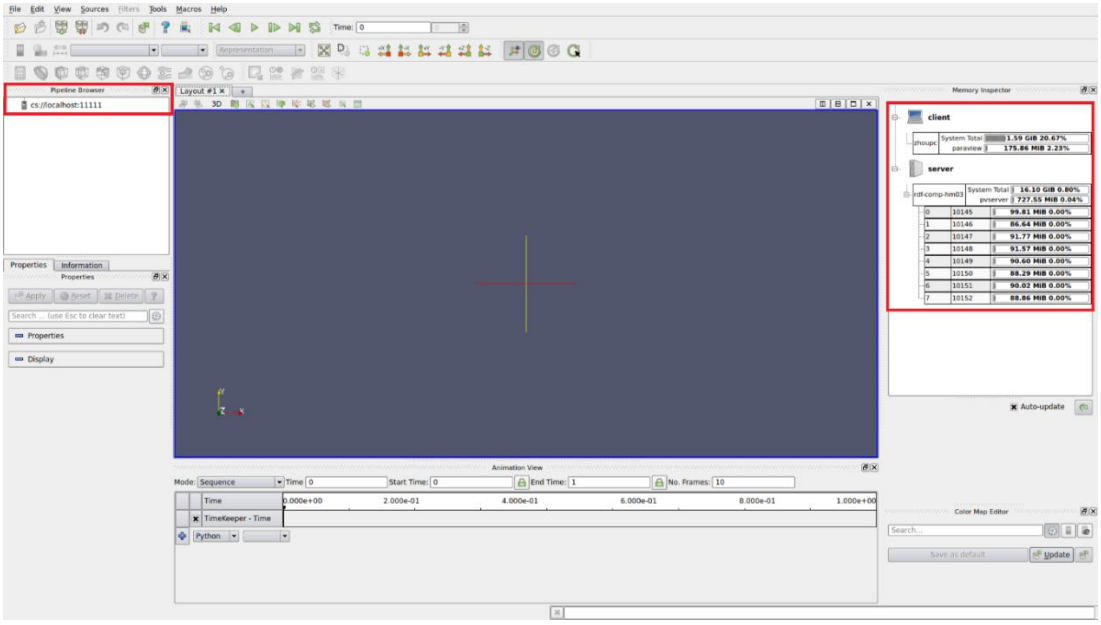

Figure 5. Successfully Connected ParaView Server

7.2 Scaling Efficiency Test

As mentioned in section 6.5, strong scaling test aims to improve the computation to run faster. In the project, the test was focused on strong scaling.

The structure of dataset consists of two blocks each with 81 time-step flow raw files, taking up 493 GB disk storage in total. A single flow file has eight variables and contains a number of 3,060,033,600 points. Testing such massive datasets is impossible due to the time limitations of the project. Therefore, the size of data is cut by choosing velocity u as the only parameter in the visualization. Data extraction is manipulated with a Python script in Appendix 13.1 by reading the whole binary values, tracking the position of desired variable and then saving as a testing flow file.

Input file size is fixed as the number of computational processors are increased as shown in Table 3.

Table 3. Parameter Settings for the Test

Case.No	Input File Size *(GB)*	Number of Nodes	Number of Cores (in total)
1	62	1	1
2	62	1	2
3	62	1	4
4	62	1	8
5	62	1	16
6	62	1	32
7	62	2	64

Timing information for each case were collected from ParaView timer log function [19]:

```
paraview.servermanager.vtkPVTimerInformation ()
```

The time consumed in the testing mainly consists of three parts, time for I/O, time for contour and time for still rendering. In timer log function, timing information are written in following formats:

Table 4. Timing Information Formats

Time for I/O (reader) (s)	Execute vtkXdmfReader
Time for filter (s)	Execute vtkPVContourFilter
Time for still rendering (s)	Still Render

Timer log files recorded every execution which took more than 0.1 seconds, so there was too much information, which was impossible to be include in the appendix. The parts of timer log files were attached and shown in Figure 6 to indicate the timer log format and the parameters recorded in the tests.

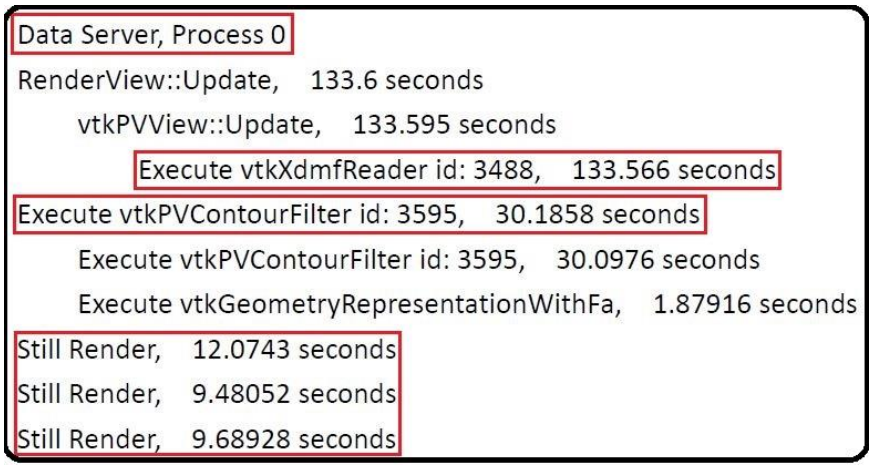

Figure 6. Timer Log File Format

All the testing was finished employing interactive visualization on Data Analytic Clusters within university eduroam wireless network (average speed of 20 Mbps). ParaView Client was run on a personal laptop with 2 cores (1.8 GHz) and 8 GB memory. Settings in ParaView were all in default forms, including filter settings, colour scales, camera position and so on [20]. The testing was repeated twice and the average values were applied as the final result. After each testing, time information for the visualization of five time steps were recorded for further analysis.

Python scripting was adopted to conduct the testing. By running the script in Python shell of ParaView, repetitive tasks could be automated. In consideration of similar scripts for each testing case, a Python script for 8 cores testing is attached as a representative example in Appendix 13.2.

7.3 Animation

The last part of the project focuses on producing a 3D flow animation with built-in animation tools in ParaView. The dataset contains two blocks each with 81 frames, taking up 493 GB disk storage. In normal conditions, second invariant of velocity gradient tensor (q-criterion) is regarded as a good simulating parameter. Render view with isosurface of q-criterion has advantages in its constant and smooth shape. ParaView has a powerful filter to calculate unstructured gradient called *GradientOfUnstructuredDataSet* [21]. Using this filter can compute vorticity and q-criterion of a three component array. However, the storage limit for student users on RDF is 70 GB, which is much lower than the scale of dataset. Under this condition, there were two options to continue the process, extracting dataset to a smaller scale or finding another desirable parameter to be rendered. Without a compromise of visualization quality, the gradient of velocity u component was chosen to be visualized in the project. Using the same method mentioned in section 7.2, velocity u was exported to a new flow file independently and loaded in the ParaView. Velocity u is the main component velocity in the flow motion and computing the gradient of velocity u will also provide better continuity in the render view.

Interactive visualization (Client/Server mode) was employed in the process. One node with 16 cores was requested to finish the visualization. A problem was met that the rendering of the data was not stored as a single frame of final output video but as values in a buffer on the memory. For a processing frame all the information of previous frames needed to be cached in the buffer, which meant larger size of datasets would require more cached space. This would cause a bottleneck on the performance. To achieve a relatively higher efficiency, the dataset

was divided into eight groups each with nine frames to be visualized and saved as PNG files. Finally, these output pictures were combined and exported as an AVI format video.

Scripting in ParaView can automate repetitive tasks. In the project, Python scripting was used to ensure the accuracy and efficiency of these groups of visualization. ParaView provides a tracing function to record the operations in a Python script, which is very convenient [22]. So a single frame was run as a trial to create a Python template script and then the script was edited with key parameters modified such as general parameters (e.g. camera position, frame numbers, colour scales) and render parameters (e.g. remote render threshold, subsample rate, LOD threshold) [23] [24]. General parameters determined what the final image looked like while render parameters had significant effect on the rendering performance, which will be discussed in section 9. These two types of parameters used in the visualization are shown in Table 5 and Table 6 respectively. Finally, the Client/Server mode was established and scripts were loaded in the Python shell to automate visualization process.

Table 5. General Parameters in the Visualization

Parameter Name	Value		
Camera Position	-22.062 (x)	-9.712 (y)	24.058 (z)
Camera Focal Point	24.495 (x)	0.440 (y)	-7.932 (z)
Camera View Up	0.229 (x)	0.781 (y)	0.581 (z)
Camera View Angle	30 degrees		
Opacity	0.8		
Headlight	0.96		
ColorMaps (RGB)	Rainbow Desaturated		

Table 6. Render Parameters in the Visualization

Parameter Name	Value
Immediate Mode Rendering	On
LOD Threshold	On (resolution of 10 by 10 by 10)
Remote Render Threshold	On (3 Mbytes)
Subsample Rate	On (2 Pixels)
Image Compression	On (squirt 16 bits)
Client Outline Threshold	On

8 RESULTS

8.1 ParaView Strong Scaling Tests

8.1.1 Five frames render with single core

The first test was run with single core on Data Analytic Cluster. Local desktop was not suitable for this test because its low memory would make visualization of last two frames stuck and thus cause ParaView to crash. The test was repeated twice and time consumed in each part per frame was averaged. Five images were output and saved in the local computer to ensure that test visualization test was run successfully and the result in Table 7 was valid for further analysis. The render view of the fifth frame in single test was shown in Figure 7.

Table 7. Scaling Test with Single Core

No. Cores = 1	Values			
No. Frame	I/O Time (s)	Filter Time (s)	Still Render Time (s)	Total Time (min)
1	337.386	32.152	123.220	8.213
2	310.656	32.978	118.647	7.705
3	305.906	32.071	118.647	7.610
4	322.107	32.024	118.647	7.880
5	307.713	32.256	118.647	7.644
Total Time	1583.768	161.482	597.808	39.051
Average Time Per Frame	316.754	32.296	119.562	7.810

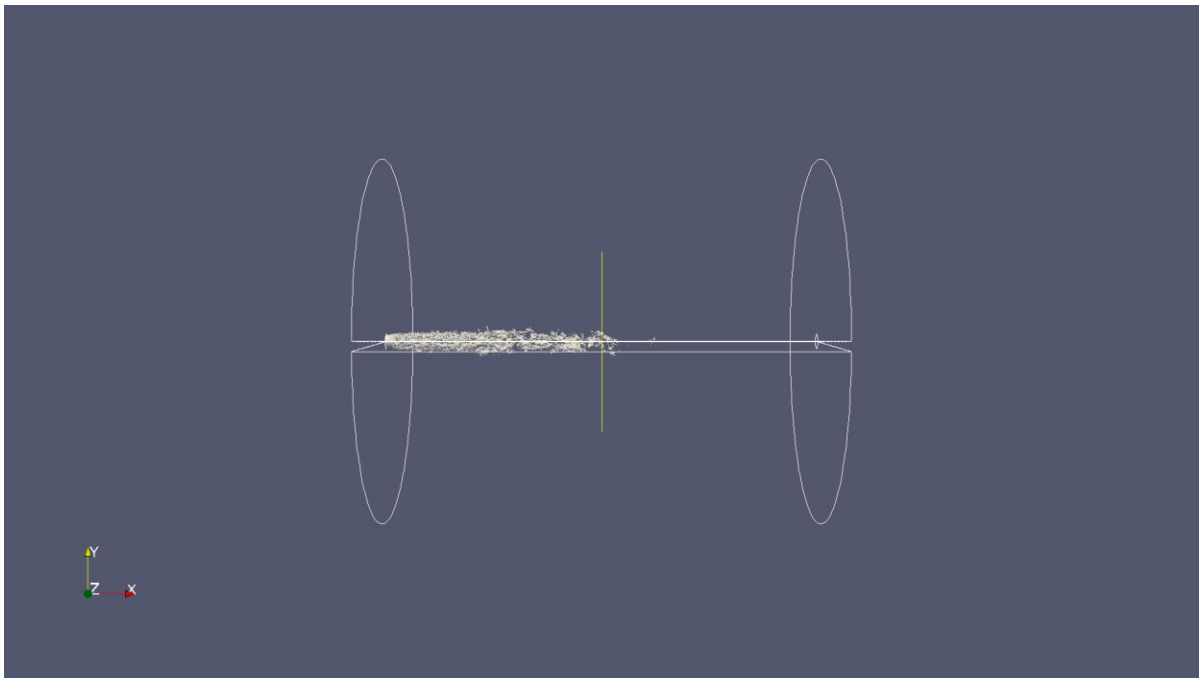

Figure 7. Render View of Fifth Frame with Default Setting

8.1.2 Five frames render with two cores

One compute node with two cores was requested to visualize five frames of the dataset. The output images are exactly same as those from the first test, so the following result is thought to be valid for further analysis.

Table 8. Scaling Test with Two Cores

No. Cores = 2	Values			
No. Frame	I/O Time (s)	Filter Time (s)	Still Render Time (s)	Total Time (min)
1	317.244	31.607	94.378	7.387
2	301.139	32.716	88.436	7.038
3	316.038	31.614	88.442	7.268
4	310.651	31.858	89.538	7.201
5	271.719	32.077	91.112	6.582
Total Time	1516.791	159.872	451.906	35.476
Average Time Per Frame	303.358	31.974	90.381	7.095

8.1.3 Five frames render with four cores

One compute node with four cores was used in this test. The render images were exactly same as those from the first test, so the following result was thought to be valid for further analysis.

Table 9. Scaling Test with Four Cores

No. Cores = 4	Values			
No. Frame	I/O Time (s)	Filter Time (s)	Still Render Time (s)	Total Time (min)
1	335.234	31.487	76.849	7.393
2	280.569	31.272	72.865	6.412
3	243.633	31.443	73.161	5.804
4	289.105	32.397	73.161	6.578
5	289.105	32.397	73.992	6.592
Total Time	1437.646	158.996	370.028	32.778
Average Time Per Frame	287.529	31.799	74.006	6.556

8.1.4 Five frames render with eight cores

Eight cores were requested to visualize five frames of the dataset. The output images were exactly same as those from the first test, so the following result was thought to be valid for further analysis.

Table 10. Scaling Test with Eight Cores

No. Cores = 8	Values			
No. Frame	I/O Time (s)	Filter Time (s)	Still Render Time (s)	Total Time (min)
1	294.444	30.186	72.962	6.627
2	283.833	30.049	65.198	6.318
3	271.778	30.151	65.164	6.118
4	292.038	35.136	69.628	6.613
5	283.566	34.757	69.620	6.466
Total Time	1425.659	160.280	342.572	32.142
Average Time Per Frame	285.132	32.056	68.514	6.428

8.1.5 Five frames render with sixteen cores

Again five frames render were repeated with sixteen cores. The testing result was correct to be adopted.

Table 11. Scaling Test with Sixteen Cores

No. Cores = 16	Values			
No. Frame	I/O Time (s)	Filter Time (s)	Still Render Time (s)	Total Time (min)
1	261.408	33.574	56.390	5.856
2	246.317	34.542	58.000	5.648
3	256.093	33.734	56.432	5.771
4	255.815	33.657	58.811	5.805
5	234.300	33.572	69.194	5.618
Total Time	1253.933	169.079	298.827	28.697
Average Time Per Frame	250.787	33.816	59.765	5.739

8.1.6 Five frames render with thirty-two cores

One compute node with four cores was used in this test. The render images were exactly same as those from the first test, so the following result was thought to be valid for further analysis.

Table 12. Scaling Test with Thirty-two Cores

No. Cores = 32	Values			
No. Frame	I/O Time (s)	Filter Time (s)	Still Render Time (s)	Total Time (min)
1	286.387	33.401	44.806	6.077
2	257.299	33.364	45.059	5.595
3	243.994	33.161	45.814	5.383
4	267.893	33.142	46.705	5.796
5	271.887	32.888	56.065	6.014
Total Time	1327.460	165.955	248.449	29.031
Average Time Per Frame	265.492	33.191	49.690	5.806

8.1.7 Five frames render with sixty-four cores

Two nodes each with sixteen cores were request to render five frames. Again the testing result was correct to be adopted.

Table 13. Scaling Test with Sixty-four Cores

No. Cores = 64	Values			
No. Frame	I/O Time (s)	Filter Time (s)	Still Render Time (s)	Total Time (min)
1	281.249	37.630	45.906	6.080
2	263.488	34.587	35.320	5.557
3	274.485	33.852	36.730	5.751
4	280.268	31.673	38.940	5.848
5	279.833	43.322	36.600	5.996
Total Time	1379.323	181.064	193.496	29.231
Average Time Per Frame	275.865	36.213	38.699	5.846

8.1.8 Multiprocessor performance comparison

The results for all the tests were put together in Table 14. Data server was in charge of I/O (reader) and filter while render server worked for still rendering. Theoretically the time used in the visualization process decreases exponentially as the number of cores increases. However, in reality ParaView scales in terms of data size instead of speed. That means the real speedup is not so dramatic as expected. From Table 14, time for filter has slight change in spite of increasing cores, which will be discussed in the next section. I/O time and still render time are chosen as two important parameters to evaluate visualization performance. Total time and operating cost are then assessed to achieve a balance of parallel efficiency and energy consuming. In consideration that running of high performance computers always consumes a tremendous amount of electric energy, operating cost of the equipment should be taken in account when evaluating the efficiency improvements as increasing number of cores. I/O performance, still render performance and total time used comparison are indicated in Figure 8, Figure 9 and Figure 10 respectively.

Table 14. Summary of Multiprocessor Performance

		Average Time Per Frame (min)			
No. Test	No. Core	I/O	Filter	Still Render	Total
1	1	5.279	0.538	1.993	7.810
2	2	5.056	0.533	1.506	7.095
3	4	4.792	0.530	1.233	6.556
4	8	4.752	0.534	1.142	6.428
5	16	4.180	0.564	0.996	5.739
6	32	4.425	0.553	0.828	5.806
7	64	4.598	0.604	0.645	5.846

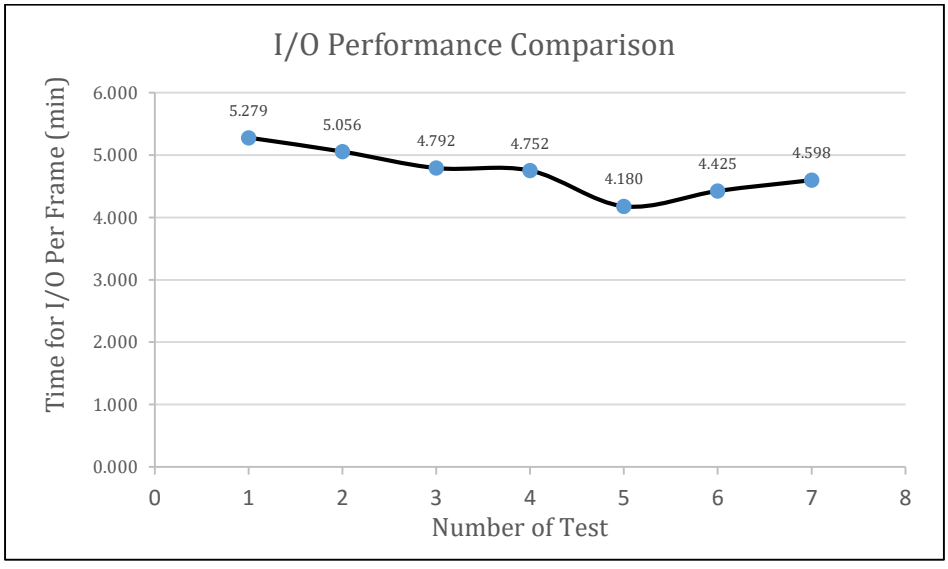

Figure 8. I/O Performance Comparison

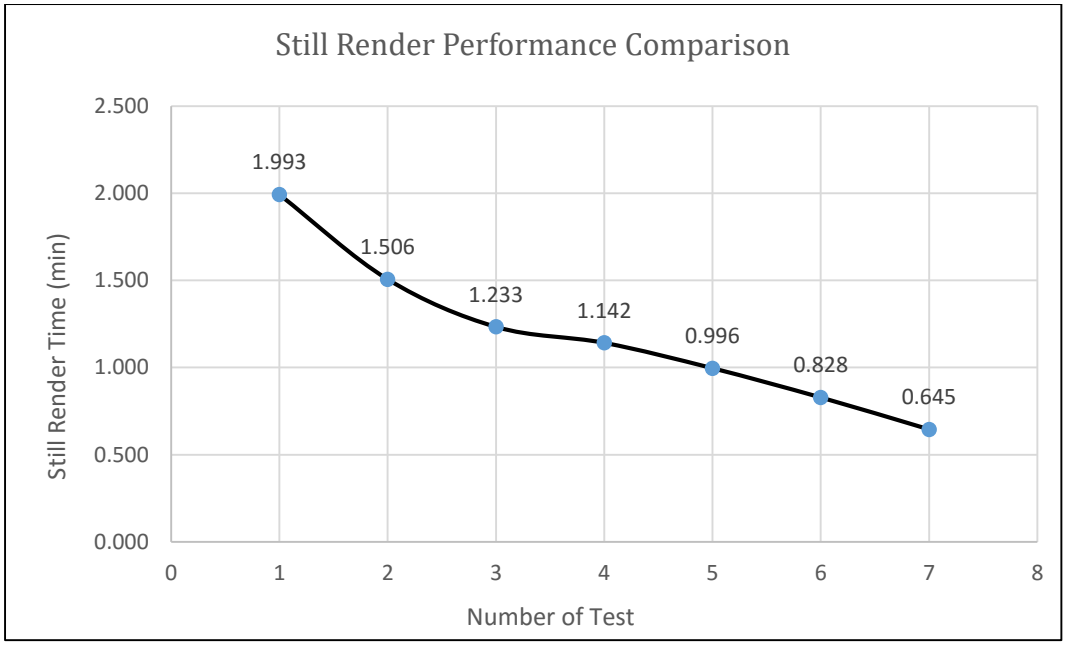

Figure 9. Still Render Performance Comparison

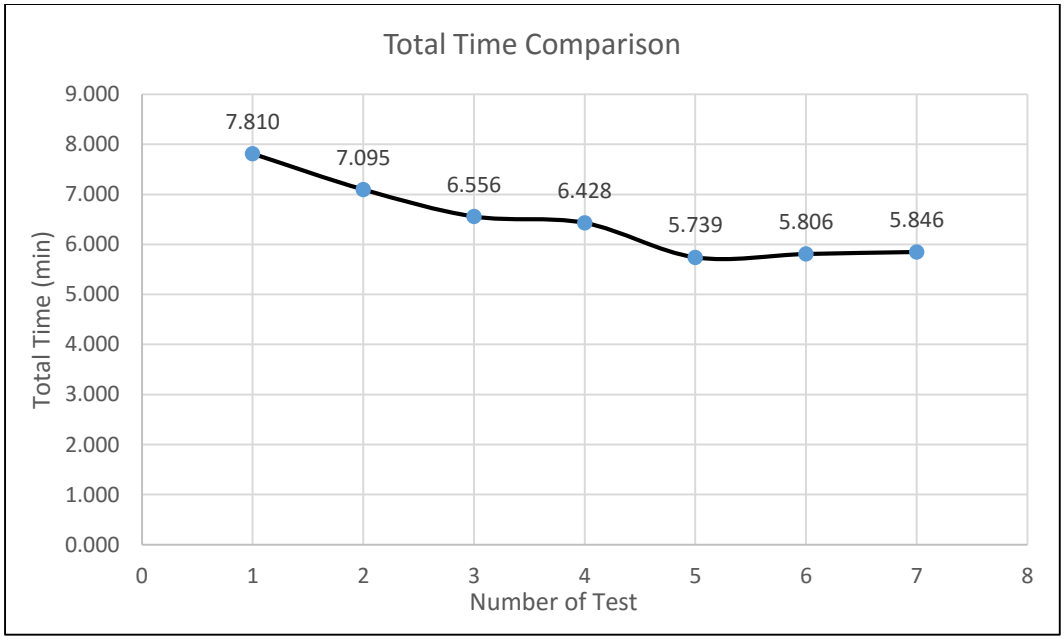

Figure 10. Total Time Comparison

From the above charts, there is an evident improvement in still render performance as the number of cores increases. However, the horizontal coordinate in the charts do not represent the number of cores but the testing number instead. That means the actual improvement is not linear and the speedup is limited in consideration of the resources used. In the I/O (reader) performance chart, there is nearly no obvious improvement and clearly I/O bottlenecks exist to restrict I/O performance.

8.2 Animation

An AVI format video was created and saced in a USB drive. PNG image files were exported from ParaView with a resolution of 1920 by 1080. Then frames were constructed using these images to finish the visualization video. Two figures are shown in Figure 11 and Figure 12. Figure 11 is a render view of velocity u gradient with colour scales. The render view of dataset had good continuity but the headlight and the colour scale applied did not present a clear detailed vision of the flow surface. So several rendering were finished with headlight and colour scales changed to find a relatively desirable view. Figure 12 is a render view of a frame in the final video saved in the USB drive.

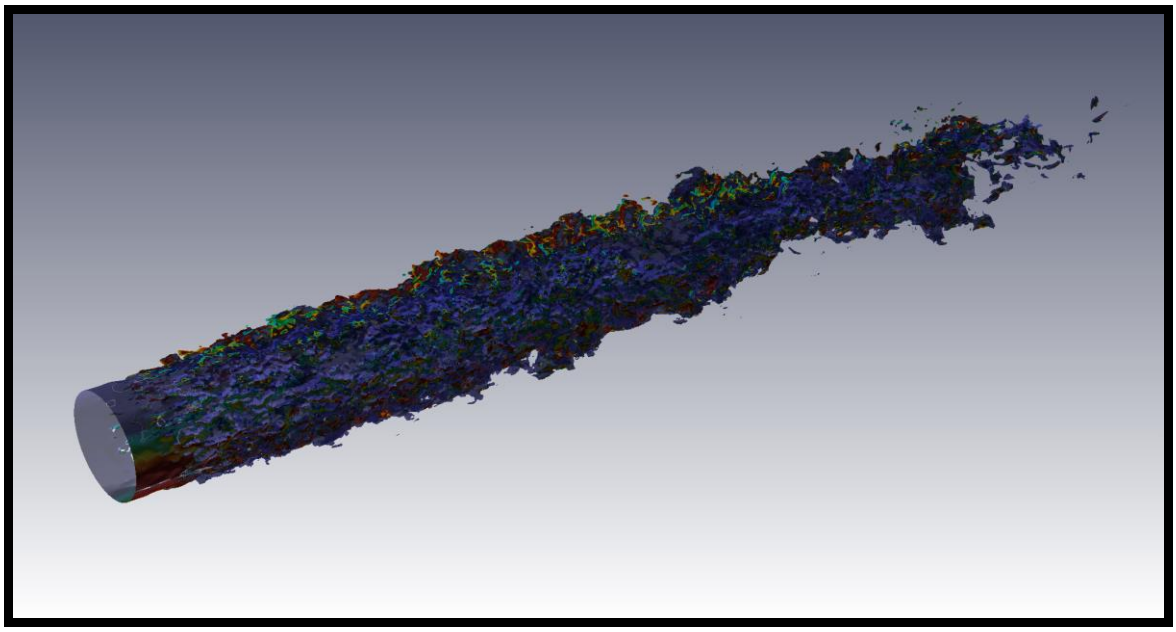

Figure 11. Isosurface of Velocity U Gradient Render View

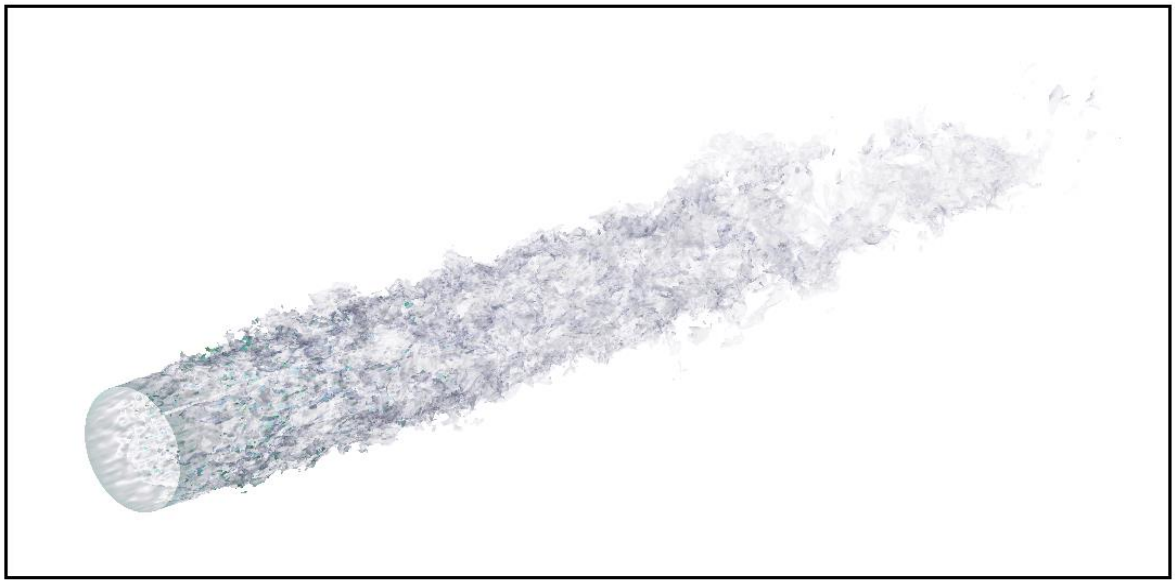

Figure 12. Isosurface of Velocity U Gradient Render View (in final video)

Time used in the render was not recorded. Compared with the scaling test, there were far more computation of still render in the final detailed so that the timer log files were too long to be assessed. By counting time on RDF, the whole process of a single frame render (Figure 12) took around 15 mins.

9 ANALYSIS and DISCUSSION

9.1 Strong Scaling Test

Overall the experimental render performance was proportional to the number of cores used. Average of two minutes were needed to finish a render process of one frame while 2 nodes with 64 cores will decrease the render time to approximately half a minute. The result for rendering speedup and scaling efficiency are show in Figure 13 and Figure 14 respectively. It was found out that multicores had limited rendering speedup.

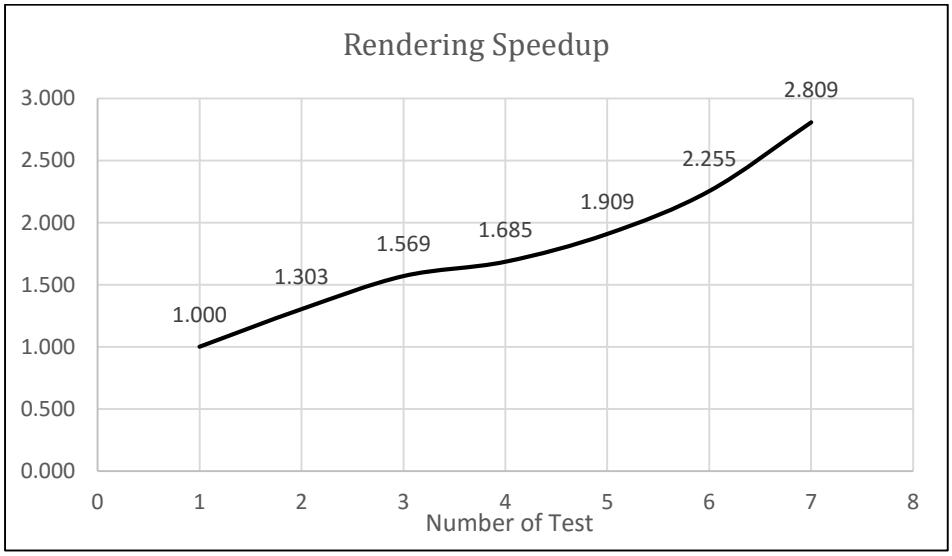

Figure 13. Rendering Speedup

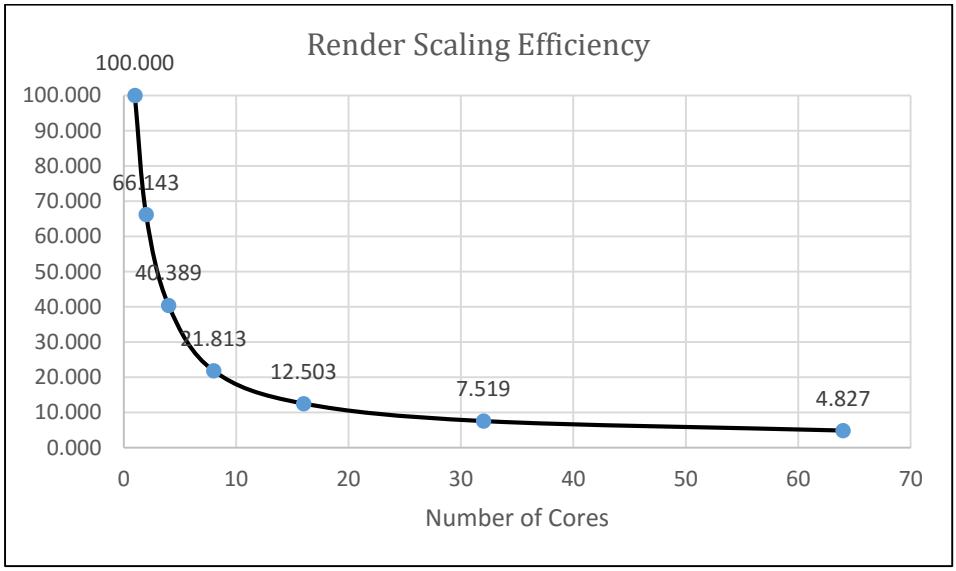

Figure 14. Scaling Efficiency

After careful consideration, the limited rendering speedup and low scaling efficiency happened due to the following reasons. The render server takes charge of generating images (frames) based on the data transferred from the data server, which is then sent back to the client for display [13]. For parallel render servers, each server renders a subset of data and create sub-images which are then composited by the client to form the final image. The communications of data server- render server and server-client rely on TCP/IP socket connections. Network on

RDF Data Analytic Cluster is extremely fast, however limited by the wireless network connection, the connection between server and client is impossible to be as fast as intranet on RDF, which will lead to communication latency [25]. Similarly, time to apply the filter in ParaView also has the same communication latency. So in the interactive (Client/Server) mode, communication latency can cause a bottleneck in performance.

For this reason, tests measuring the time to apply filter on standalone ParaView were performed. Actually the time used to apply the contour filter is very short and by measurements with timer log function, it took around 10 seconds to finish the contour filter per frame on standalone ParaView. As shown in Table 14, the filter time for each test did not indicate clear variations, which can be explained that the main parts of filter time were not actually consumed in the filtering but in the communications. Therefore, an estimated communication time was determined and taken out of the measured still time. The results are displayed in Figure 15 and Figure 16.

ParaView scales in terms of data size instead of speed [26]. That means if the data size is as small as one processor can be competent, increasing the number of processors will probably slow it down. If the data is very large, because the processing exceeds memory entirely, increasing the number of processors will speed the process up. However, the speedup is impossible to increase exponentially as the number of cores increases since the communication overhead for most algorithms in proportion to the number of cores used [].

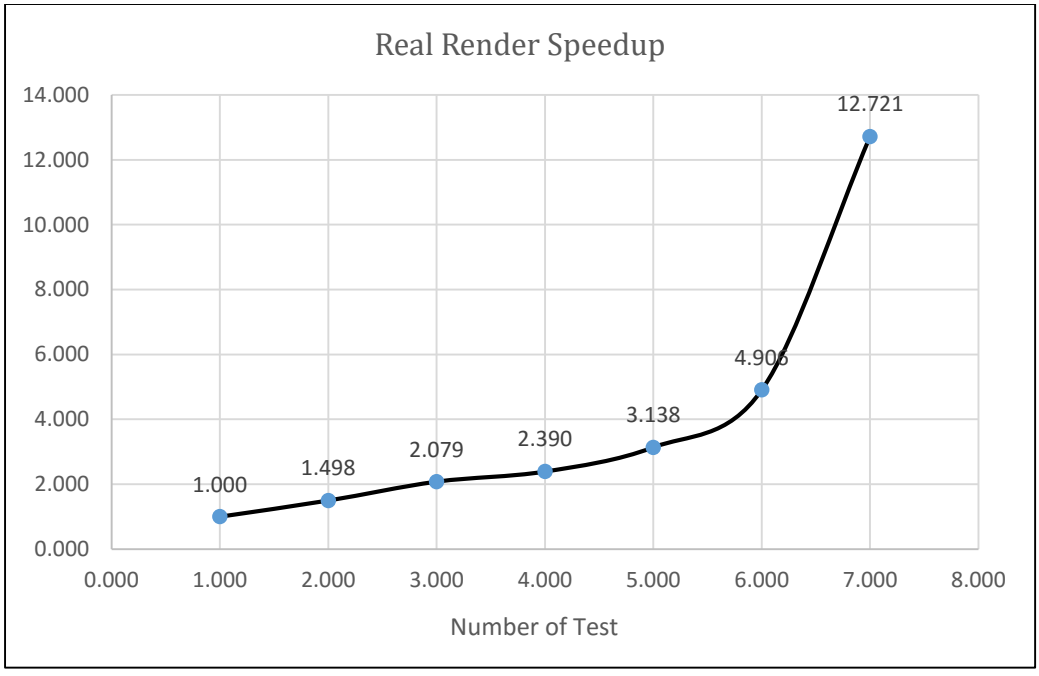

Figure 15. Real Still Rendering Speedup

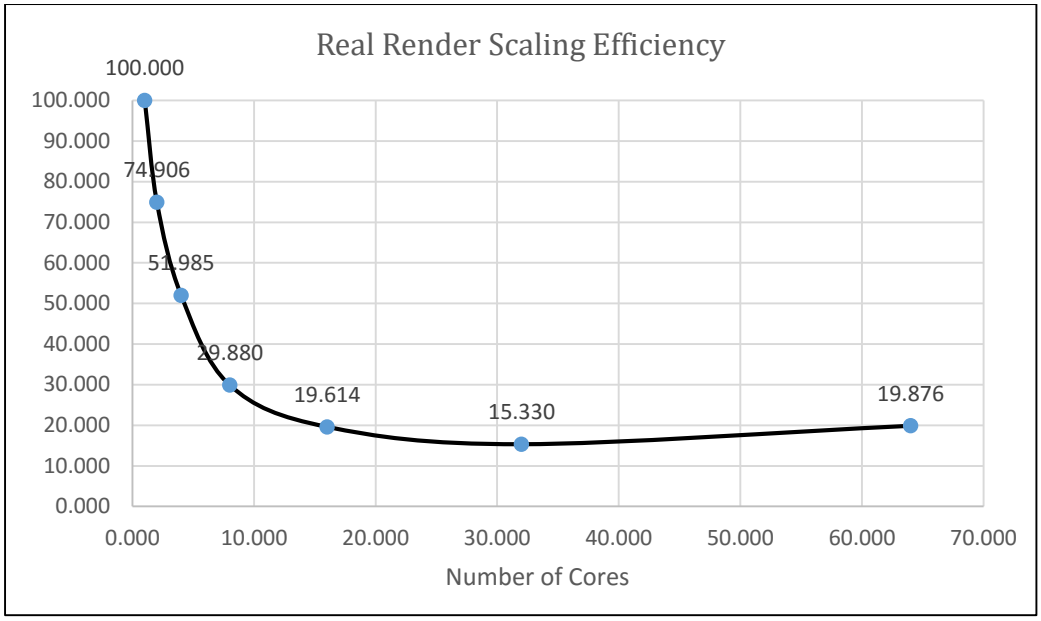

Figure 15. Real Render Scaling Efficiency

Thus it was concluded that the server was run properly as a parallel application although the render speedup and render scaling efficiency still had much space for improvements.

CPU time on ARCHER and RDF was measured in kAU [27]. Each job run on the service consumed kAUs from the budget. Usage of resources and corresponding cost is shown in Table 15 with the reference to ARCHER Resource Management chapter [27]. In the testing, the maximum number of nodes used was two. That made little effect on the final compute resource consumed and notional cost. However, when a larger scale of data is processed and thousands of cores are requested, the budget will be considered to be a key factor to balance the compute efficiency and the cost.

Table 15. Usage of Resources and Cost in the Tests

No. Test	No. Core	No. Node	Compute Resource (kAU)	Notional Cost (£)
1	1	1	2.520	1.410
2	2	1	2.520	1.410
3	4	1	2.520	1.410
4	8	1	2.520	1.410
5	16	1	2.520	1.410
6	32	1	2.520	1.410
7	64	2	4.320	2.420

From Table 14, it was clear that reader time (I/O) took up the most of total visualization time and had few changes as the number of processors increases. There was a suggestion that the bottleneck was caused because the dataset was stored on RDF as serial files [28]. It can be assumed that there are a number of branches connected to ParaView servers (MPI enabled) but only one entrance is available to get access to these branches. That means for the large data, when parallel servers were launched all the processes had to get access to the same file and perform logically the same operations on the data. Then servers will operate on their own distributed subset of the data. One solution is to use lustre file systems on ARCHER or another proper I/O [29]. By setting stripes of the data storage, ARCHER lustre file systems can store the data in a parallel way, which means the data will have multi channels each with branches

connected to servers. However, ARCHER lustre file system is applied in the work disk and cannot load the data from RDF. In consideration that currently there is not a MPI-enabled ParaView supporting interactive visualization on ARCHER, the test needs to have a comprise to be run on RDF DAC. Moreover, since the client and server are connected via communication sockets, it is a relatively slow mode of connection. In contrast, off-screen rendering can be controlled and achieved by a pvpython script attached to the job submission file. By doing this, there will not be communication between client and server and the time consumed can thus be reduced by minimizing the data transfer over the socket.

9.2 Animation

The last part of the project was to produce an animation of 3D combustion flow. Output video was saved in a USB drive and the display rate was 5 frames per second. During the render of frames, several problems were met. Render processes usually crashed after rendering nine frames and the output images were not rendered thoroughly like Figure 16.

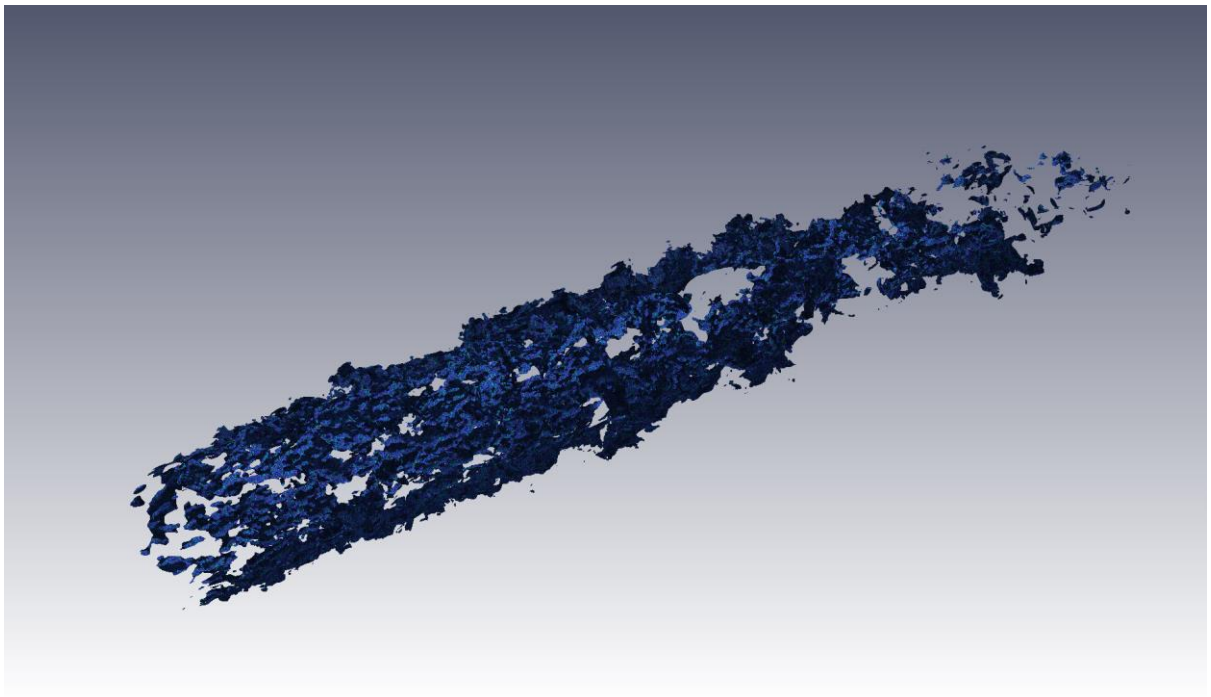

Figure 16. Insufficient Rendered Image

After repeated testing, the insufficient render problem was solved by adjusting key render parameters. Remote rendering had to be turn on to use the power of entire server to render and send images to the client. Remote render threshold was set as 0 MB, which meant the geometry would not be sent to the client at any time. Usually by default, this value is 2MB and the geometry instead of images will be moved to the client when the geometry is below the threshold, which will increase the burden on the socket communication. When the data size is large, it is always quicker to send images than ship data. In addition, immediate mode rendering has to be turn on, which can prevent the graphics systems from building special rendering structures. Overall any delay in the render process will increase the probability of making Client/Server grow enter the “waiting for response” mode. Under that condition, ParaView server will be disconnected and the client will crash to output an insufficient rendered image. In addition, a research [24] suggests that the data is not stored as images but values in a buffer

on the memory. For each new image displayed, all the information about the previous images need to be cached in the buffer. When producing animations, a number of frames will be cached in the buffer. As frames produced increase, the size of data cached also grows larger, which can cause a bottleneck on the performance [31]. This is another reason making the process crash. To solve it, a hard way is adapted. The visualization process will be divided into several repeated operations, each of which will be directed by the Python script and produce nine frames to avoid crashing [30]. The script has the similar format as the script in Appendix 13.2. The different parts are attached in Appendix 13.3. This is not a very smart solution but it actually works well.

10 FURTHER WORK

The further work can be carried out mainly in three categories, parallel I/O, off-screen rendering and render server/data server separation in ParaView.

File I/O often becomes a severe bottleneck when a large scale of data is processed on parallel application. MPI IO can improve performance by orders of magnitude over serial I/O. However, the realization of parallel IO is very complicated, requiring tuning of parameters. First, parallel Io must be configured to use collective routines; second, the lustre file system must be configured to use appropriate parallel striping [30]. Due to the time limitation of the project, parallel I/O is not achieved in the test. But it is worthwhile to compile a MPI enabled ParaView on ARCHER and configure to perform parallel I/O on ARCHER.

Compared with Client/Server mode, off-screen rendering has overwhelming advantages in improving visualization efficiency. Pvbatch takes a Python script and can be run by a simple command without any interaction with users. Therefore, time consumed in the communication of client-server socket can be saved and the whole process can be automated. The challenge of off-screen rendering is to ensure the Python script to be run is effective and contains all the information needed. Python tracing function is useful to create a python template by recording users' operations but to achieve some advanced animations, users need to be familiar with ParaView servermanager module and have a strong programming ability.

By default, servers consist of data servers and the same number of render servers. Each data server has one, and only one connection with render server. This mode is simple and robust. However, in practice, data processing takes up more resources in data server, which is responsible for data reading, filtering and writing. So it will be efficient to allocate the number of data servers and render servers by users themselves. Since the separation of data/render server is customized, further investigation and benchmarking can be conducted as an independent project.

11 CONCLUSION

ParaView is a powerful scientific visualization tool widely installed on high performance computers for large scale data rendering. The literature survey indicates that parallel computing is crucial in the big data processing. ParaView provides two modes for large data visualization, Client/Server and off-screen rendering. Parallel computing in ParaView mainly consists of parallel I/O and parallel rendering. In the project, Client/Server was used and a parallel rendering performance test was conducted. With the size of testing data fixed, a series of tests with different number of cores were finished. From the section 9, it was indicated that the still render performance would be effectively improved by increasing the number of cores. But due to some limitation factors, the total performance was not improved. The reasons were investigated and explained in mainly two aspects. One was the restriction of I/O performance. Although data servers support MPI, there was a severe bottle neck in the data transfer between file system and data servers, especially for the large dataset. Moreover, there was a socket communication between client and server, which was thought as a slow connection. Despite the overall optimal visualization being incomplete, it is still believed to be a good starting point to optimize and investigate the render performance on parallel application. All the progresses and changes are documented in the report so it can be convenient for someone to continue and completer other aspects in optimal visualization (as discussed in section 10).

Another area of the project is to produce a 3D flow animation. Crashing problems were met during the process and the reasons and solutions were introduced in section 9. Animations were successfully produced in the end. However, due to the time limitation of the project, some solutions seem not very smart. It would be really interesting to have more time to figure out the problem that data cached in the buffer grow larger as frames produced increase.

Overall the project was finished and expectations and the goals were fulfilled. This project has successfully contributed to the optimization of parallel visualization on RDF DAC.

12 BIBLIOGRAPHY

[1] David Ellsworth, Bryan Green, Chris Henze, Patrick Moran, & Timothy Sandstorms. (2006). Concurrent Visualization in a Production Supercomputing Environment. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*. 12 (5), p1-7.

[2] H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Maxi. A contract based system for large data visualization. In Visualization, 2005. VIS 05. IEEE, 2005.

[3] Prabhu ramachandran. (2011). Mayavi: 3D Visualization of Scientific Data. *Scientific Python*. 3 (2), p40-49.

[4] Sandia National Laboratory. (2014). *Raw Files*. [online] Available at: http://www.paraview.org/Wiki/ParaView/Data_formats.

[5] hipstar-code.org. (2016). *HiPSTAR/Code/subspaces*. [online] Available at: http://HiPSTAR/Code/subspaces - hipstar-code.org.

[6] J. Nievergelt, H. Hinterberger *The Grid File: An Adaptable, Symmetric Multikey File Structure*. Institut fur Informatik, ETH and K. C. Sevcik, 1984. Abstract, pp.1.

[7] Xdmf.org. (2016). *XDMF Model and Format - XdmfWeb*. [online] Available at: http://www.xdmf.org/index.php/XDMF_Model_and_Format.

[8] Kitware. (2016). *Big Data Analysis with ParaView*. [online] Available at: http://extremecomputingtraining.anl.gov/files/2015/03/DOEPV11-demarle-aug6-330.pdf.

[9] Cedilnik, A., Geveci, B., Moreland, K., Ahrens, J. and Favre, J. (2006). Remote Large Data Visualization in the ParaView Framework. Parallel Graphics and Visualization. *The Eurographics Association*, 13(1).

[10] Snir, Marc; Otto, Steve W.; Huss-Lederman, Steven; Walker, David W.; Dongarra, Jack J. (1995) *MPI: The Complete Reference*. MIT Press Cambridge, MA, USA.ISBN 0-262-69215-5

[11] Paraview.org. (2016). *Setting up a ParaView Server - KitwarePublic*. [online] Available at: http://www.paraview.org/Wiki/Setting_up_a_ParaView_Server.

[12] ParaView Scripting with Python. (2007). *Servermanager*. [online] Available at: http://www.paraview.org/Wiki/images/2/26/Servermanager.pdf

[13] CINECA -SCAI. (2014). *Post-processing with Paraview*. [online] Available at: http://www.training.prace-ri.eu/uploads/tx_pracetmo/ParaviewLargeData.pdf

[14] Archer.ac.uk. (2016). *ARCHER » Hardware*. [online] Available at: http://www.archer.ac.uk/about-archer/hardware/ [Accessed 5 May 2016].

[15] Archer.ac.uk. (2016). *ARCHER » 1. Introduction to UK-RDF*. [online] Available at: http://www.archer.ac.uk/documentation/rdf-guide/introduction.php

[16] Sharcnet.ca. (2016). *Measuring Parallel Scaling Performance - Documentation*. [online] Available at: https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance

[17] Wiki.scinet.utoronto.ca. (2016). *Introduction to Performance - SciNetWiki*. [online] Available at:
https://wiki.scinet.utoronto.ca/wiki/index.php/Introduction_To_Performance

[18]7.1 Yang, O. (2014). *How to Use Paraview in Client-Server Mode*. [online] School of Engineering and Centre for Scientific Computing. Available at:

http://www2.warwick.ac.uk/fac/sci/eng/study/pg/students/esrmae/openfoam/use_paraview_client-server.pdf

[19] Paraview.org. (2015). *ParaView/Python Scripting*. [online] Available at:
http://www.paraview.org/Wiki/ParaView/Python_Scripting.

[20] HPCx Consortium. (2007). *Parallel Visualisation on HPCx*. [online] Available at: http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0803.pdf

[21] Martin, K. (2016). *ParaView All you need for parallel visualization*. [ebook] Kitware Inc. Available at:
http://graphics.stanford.edu/~mhouston/VisWorkshop04/ParaView.pdf.

[22] Martin Jucker. (2014). Scientific Visualization of Atmospheric Data with ParaView. *Open research software*. 2 (1), p2-4.

[23] Paraview.org. (2016). *paraview.benchmark — ParaView/Python 5.0.1-285-g1828811 documentation*. [online] Available at:
http://www.paraview.org/ParaView/Doc/Nightly/www/py-doc/_modules/paraview/benchmark.html

[24] Klontza, G. (2012). *4D Visualisations of Climate Data with ParaView*. Msc in High Performance Computing. The University of Edinburgh.

[25] Antoni Artigues, Fernando M. Cucchietti, Carlos Tripiana Montes, David Vicente, Hadrien Calmet, Guillermo Marin, Guillaume Houzeaux, Mariano Vazquez. (2014). Scientific Big Data Visualization: A Coupled Tools Approach. *Supercomputing Frontiers and Innovations*. 10 (2), p4-15.

[26] Moreland, K., Greenfield, J. and Scott, W. (2009). *Large Scale Visualization with ParaView*. [ebook] Available at:
http://myweb.clemson.edu/~eduffy/sc09_tutorials/docs/tut102/SC09_ParaView_Tutorial_Slides.pdf.

[27] ARCHER. (2016). *ARCHER kAU Calculator*. [online] Available at:
https://www.archer.ac.uk/access/au-calculator/ [Accessed 5 May 2016].

[28] EPCC ARCHER. (n.d.). *Lustre /work filesystem*. [online] Available at:
Http://www.archer.ac.uk/documentation/data-management/lustre.php#LustreBasics
[Accessed 5 May 2016].
[29] Henty, D., Jackson, A., Moulinec, C. and Szeremi, V. (2015). *Performance of Parallel IO on ARCHER*. [online] EPCC ARCHER. Available at:
https://www.archer.ac.uk/documentation/white-papers/parallelIO/ARCHER_wp_parallelIO.pdf

[30] 6.2 Ponzini, R. (n.d.). *Paraview scripting*. [online] Summer School on Scientific Visualization. Available at: http://www.training.prace-ri.eu/uploads/tx_pracetmo/paraviewBatch.pdf

[31] Eric Weissenstein, Onkar Sahni, Ken Jansen and Adam Todorski, (2016). Running ParaView in Parallel. [online] Available at:
https://wiki.scorec.rpi.edu/wiki/Running_ParaView_In_Parallel.

13 APPENDIX

13.1 Data Extraction Script

```
"""
Created on Thu Apr 28 17:48:29 2016
@author: yuzhe
"""

def read_data(directory1, directory2, itr, nvar):
  import numpy as np
   #%% Process the first file
   filename1 = directory1+'/FLOW_phys_1_var_8_'+str(itr)+'.raw'


   # Open the Stat files
   with open(filename1', 'rb') as fobj:
     grid_size = np.fromfile(fobj, 'i', 3);
     Mach = np.fromfile(fobj, 'f', 1);
     dummy = np.fromfile(fobj, 'f', 1);
     ReyNo = np.fromfile(fobj, 'f', 1);
     Time = np.fromfile(fobj, 'f', 1);


   Nx  = grid_size[0];
   Nr1 = grid_size[1];
   Nth = grid_size[2];
   # Move the file pointer
   fid1.seek(Nx*Nr1*Nth*(nvar-1)*4, 1);   # seek from the current position


   # Read the data & close the data
   q1 = np.fromfile(fid1, 'f', Nx*Nr1*Nth)
   filename2 = file(directory2+'/FLOW_phys_1_var_8_'+str(itr)+'.raw', 'w')
   fid2 = open(filename2, 'wb')
   grid_size.tofile(fid2)
   Mach.tofile(fid2)
```

```
    dummy.tofile(fid2)
    ReyNo.tofile(fid2)
    time.tofile(fid2)
    q1.tofile(fid2)

    fid1.close()
    fid2.close()
##
aa = 1100000
while aa < 1140000:
    aa = aa + 500
    read_data('/media/yuzhe/My Book/Yuzhe2', '/media/yuzhe/My Book/Yuzhe3',  aa, 2)
```

13.2 ParaView Python Script for 8 Cores Test

```
Try: paraview.simple

except: from paraview.simple import *

paraview.simple._DisableFirstRenderCameraReset()


Flow_phys_YZ_xmf = XDMFReader( FileName='/epsrc/e476/e476-
zhou/zhou/data/Flow_phys_YZ.xmf' )


AnimationScene2 = GetAnimationScene()

AnimationScene2.EndTime = 1680.0

AnimationScene2.AnimationTime = 1640.0

AnimationScene2.PlayMode = 'Snap To TimeSteps'

AnimationScene2.StartTime = 1640.0


RenderView2 = GetRenderView()


RenderView2.CameraPosition = [0.0, 0.0, 3.777533164199517]

RenderView2.CameraClippingRange = [1.7497578325575218, 6.33919616166251]

RenderView2.CenterOfRotation = [30.048175811767578, 0.0, 0.0]


Flow_phys_YZ_xmf.Sets = []

Flow_phys_YZ_xmf.Grids = ['Time 1101500', 'Time 1102000', 'Time 1102500', 'Time
1103000', 'Time 1103500', 'Time 1100000[1]', 'Time 1100500[1]', 'Time 1101000[1]', 'Time
1101500[1]', 'Time 1102000[1]']

Flow_phys_YZ_xmf.PointArrays = ['$  ho$', '$ T$', '$ Z0$', '$ Z1$', '$ a$', '$ u$', '$ v$',
'$ w$']


DataRepresentation1 = Show()

DataRepresentation1.EdgeColor = [0.0, 0.0, 0.5000076295109483]

DataRepresentation1.SelectionPointFieldDataArrayName = '$ u$'

DataRepresentation1.ScalarOpacityUnitDistance = 0.08498448592470886
```

```
DataRepresentation1.ExtractedBlockIndex = 1
DataRepresentation1.Representation = 'Outline'
DataRepresentation1.ScaleFactor = 6.009635162353516

RenderView2.CameraPosition = [30.048175811767578, 0.0, 180.12491354366458]
RenderView2.CameraFocalPoint = [30.048175811767578, 0.0, 0.0]
RenderView2.CameraClippingRange = [128.15987309505655, 245.97266525661567]
RenderView2.CameraParallelScale = 46.61975812254537

Flow_phys_YZ_xmf.PointArrays = ['$ u$']

Contour1 = Contour( PointMergeMethod="Uniform Binning" )

Contour1.PointMergeMethod = "Uniform Binning"
Contour1.ContourBy = ['POINTS', '$ u$']
Contour1.Isosurfaces = [0.46619756519794464]

Contour1.GenerateTriangles = 0
Contour1.ComputeGradients = 1

DataRepresentation2 = Show()
DataRepresentation2.ScaleFactor = 4.782422056794167
DataRepresentation2.SelectionPointFieldDataArrayName = 'Gradients'
DataRepresentation2.EdgeColor = [0.0, 0.0, 0.5000076295109483]

DataRepresentation1.Visibility = 0

RenderView2.CenterAxesVisibility = 0
RenderView2.OrientationAxesVisibility = 0
RenderView2.CameraClippingRange = [172.20885355522347, 190.78063274299723]
```

WriteAnimation('/home/yuzhe/Documents/np8.png', Magnification=2, Quality=2, FrameRate=10.000000)

Render()

13.3 Parts of ParaView Python Script for Animations

```
RenderView1.RemoteRenderThreshold = 3.0
RenderView1.CameraClippingRange = [64.46409196824584, 76.1896440306247]
RenderView1.Background = [0.8666666666666667, 0.8862745098039215, 1.0]
RenderView1.CameraPosition = [30.048175811767578, 0.0, 69.44595167583441]

CameraAnimationCue1 = GetCameraTrack()
CameraAnimationCue1.AnimatedProxy = RenderView1
CameraAnimationCue1.Mode = 'Path-based'

TimeAnimationCue1 = GetTimeTrack()

camera = view.GetActiveCamera()
num_of_keyframes = 10
for i in range(0, num_of_keyframes):
    camera.Azimuth(360.0/num_of_keyframes)
            WriteAnimation('/epsrc/e476/e476/zhou/project/test.avi', Magnification=1,
Quality=2, FrameRate=15.000000)
    keyframe = animation.CameraKeyFrame()
    # set the value of the key frame to the current camera location.
    keyframe.KeyTime = i*1.0/num_of_keyframes
    keyframe.Position = camera.GetPosition()
    keyframe.FocalPoint = camera.GetFocalPoint()
    keyframe.ViewUp = camera.GetViewUp()
    keyframe.ViewAngle = camera.GetViewAngle()
    # cue.KeyFrames.append(keyframe)
# CameraAnimationCue1.KeyFrames = [ KeyFrame4661, KeyFrame4662 ]
```