Parallelising a Reservoir Geophysics Code on Cayenne

Alex Hunt

August 2011
# Table of Contents

1. Introduction..................................................................................................................2
   1.1 Project Objectives.................................................................................................2
   1.2 Organisation of the Dissertation...........................................................................3
2. Project Overview..........................................................................................................5
   2.1 Background ...........................................................................................................5
   2.2 Aims.......................................................................................................................6
   2.3 Discovering How the Existing SVD Code Worked...............................................9
   2.4 Lessons learned: Assessment of Project Planning...............................................9
   2.5 Project Risk Assessment......................................................................................10
3. Porting and Upgrading of SVD Sequential Code to Cayenne.................................12
   3.1 Compilers and Development Tools on Cayenne.................................................12
   3.2 Development Environment Set-up and Issues Encountered.............................13
      3.2.1 Installing Eclipse...........................................................................................13
      3.2.2 Setting up Photran on Eclipse........................................................................15
      3.2.3 Setting up code repository on NESS.............................................................16
      3.2.4 Setting up SVN and Linking to the Code Repository on NESS..................16
      3.2.5 Creation and Execution of C and Fortran code in Eclipse.........................17
      3.2.6 Portability of Eclipse Development Environment.......................................19
   3.3 Code Changes.......................................................................................................19
      3.3.1 Changes to Serial C Code.............................................................................20
      3.3.2 Changes to Serial Fortran Code....................................................................24
4. Parallelisation, Optimisation and Performance Testing of SVD code on Cayenne .....30
   4.1 Development Environment Issues Encountered.................................................30
      4.1.1 Parallelisation and Execution of C and Fortran code in Eclipse...............30
      4.1.2 Portability of Eclipse Development Environment for Parallelisation........30
   4.2 Code Parallelisation..............................................................................................31
      4.2.1 Investigate Parallel I/O and the Use of MPI-IO..........................................31
      4.2.2 Changes to Parallel Fortran Code.................................................................32
   4.3 Testing of Code.....................................................................................................32
5. Benchmarking Cayenne.............................................................................................33
   5.1 Sourcing Test Packages.......................................................................................33
List of Tables

Table 1: Example Monitor Trace Subtraction from Reference Trace .................9
Table 2: Original Gantt Diagram for Project.........................................................10
Table 3: MIB-MPII All Test Results........................................................................35
Table 4: Full Set of Run-times For Cayenne with the Small Data Set.................42
Table 5: Results for Cayenne Run-time Tests on the Medium Data Volume.........44
Table 6: Results for Large Seismic (6GB) Seismic Volume Showing linear Speed-up ...........................................................................................................46
Table 7: Comparison of Run-times for Code Versions Serial, MPI on Font-end and MPI on Back-end ..............................................................46
Table 8: Comparison of Run-times For Cayenne and Ness...................................50
Table 9: Comparison of Speed-up For Cayenne and Ness.................................51
list of Figures

Figure 1: Location of Nelson Oil and Gas Field [4].................................2
Figure 2: Typical Beowulf Cluster Layout [17]........................................8
Figure 3: Eclipse Project Set-up...................................................................16
Figure 4: SVN Repository Set-up in Eclipse...............................................17
Figure 5: Fully set up SVD Code Project, Showing Project Makefile for Cayenne Compilation.................................................................18
Figure 6: Comparison Memory Assignment arralloc() or array[][]...............21
Figure 7: New code to deal with stack overflow with stack arrays[][]...........22
Figure 8: View of Four Refactoring Options Applied..................................24
Figure 9: Semi-automatic Refactoring Tools in Photran Eclipse..................26
Figure 10: IMB Ext Benchmark Non-Aggregated MPI Message Calls.........36
Figure 11: Example of Optimal Fortran 95 Code Layout............................38
Figure 12: Synthetic Seismic Dataset of 32 Traces, baseline_synthetic.segy...40
Figure 13: Speed-up for Small Data Volume on Cayenne............................43
Figure 14: Comparison of IO and SVD Run-times for Medium Data Set on Cayenne.......................................................................................45
Figure 15: Comparison of Small, Medium and Large Seismic Volume Run-time...47
Figure 16: Comparison of Run-times of Serial and MPI Code....................48
Figure 17: Comparison of IO and SVD Run-times for the Large Data Set on Cayenne.......................................................................................49
Figure 18: Comparison Speed-up on Cayenne and Ness.............................50
Figure 19: Relationship Between Ness Speed-up, IO and SVD Run-time........51
Acknowledgements

I would like to thank my supervisor Dr Fiona Reid for her support, advice and most of all her encouragement during this dissertation work. Her understanding of the subject area and first-hand knowledge of the original SVD code is invaluable to this dissertation project. Her inputs have been key in helping me keep on the right path and because of them I have been able to work with a clear direction on this report.

Dr Fiona Reid is a researcher at the EPCC, Edinburgh University, Edinburgh, UK.

Email: fiona@epcc.ed.ac.uk

I am grateful to Professor Colin MacBeth, for his help, advice and suggestions during this dissertation work. I am also grateful for his help in gaining access to the resources at Heriot-Watt University library¹, the Reservoir Geophysics Group's library in particular.

Colin MacBeth is Professor at the Reservoir Geophysics Group of the Institute of Petroleum Engineering at Heriot-Watt University in Edinburgh UK [1].

Email: colin.macbeth@pet.hw.ac.uk

---

¹ Heriot-Watt, has an extensive geotechnical library, a collection of papers relating to 4-D time-lapse seismic techniques and reservoir modelling. These have proved essential to my understanding of the motivations behind this project.
Abstract

This document concerns the process, results and lessons learned during a project to update, optimise and parallelise Singular Value Decomposition [2] code, supplied by ETLP and to benchmark the Cayenne Beowulf cluster. The SVD code generates 4-D time-lapse seismic volumes for use in oil and gas reservoir modelling [3]. The benchmarking of the Cayenne Beowulf cluster was undertaken with off the shelf libraries such as HPL and the INTEL IMB cluster benchmarking suite. Cayenne is run by the Institute of Petroleum Engineering at Heriot-Watt University in Edinburgh UK and is used for Earth science data analyses. The process of updating and parallelisation of the SVD code was successfully applied to real data volumes of 6GB on the Cayenne cluster where the resulting speed-up was linear. The aims of the project were successfully addressed giving a deliverable of upgraded code that was capable of processing very large seismic data volumes on Cayenne.


1. Introduction

1.1 Project Objectives

What motivated this project was the need to update and parallelise an existing Singular Value Decomposition (SVD) code, which will be used to generate 4-D seismic time-lapse volumes quickly and efficiently. 4-D seismic volumes are made from multiple 3-D seismic volumes, usually shot a few years or decades apart, hence the expression “time-lapse”. In simple terms, two 3-D volumes are subtracted from each other giving a new volume that only shows what has changed during the intervening years.

The specific 3-D volume used for this project was from the Nelson field. The Nelson field, discovered in 1987 has been in production since 1994 and is operated by Shell UK Ltd. It is 202km north east of Aberdeen in the UK sector of the Northern North Sea [4].

![Figure 1: Location of Nelson Oil and Gas Field [4]](image)

The Nelson field data was collected “shot” at sea using an air-gun\(^2\) to create sound waves and recorded using a towed array of hydrophones, called a streamer\(^3\) to collect the returned sound wave energy. The data gathering procedure for marine 3-

---

\(^2\) An air-gun is a long tube, closed at one end with the open end under water, which releases high pressure air into the sea effectively creating an explosion.

\(^3\) A streamer is a neutrally buoyant tube, usually filled with oil, containing hydrophones, which are like microphones but work under water. It is used to gather sonic data from marine environments and is used for sonar as well as seismic applications.
D seismic volumes involves using a vessel following a grid pattern of “in-lines” [5]. Whilst navigating along a predetermined course seismologists aboard the vessel set off the air-gun at regular intervals. Using a streamer, connected to a real-time data processing system, they record and process the returned sound waves. These sound or seismic waves, reflected off layers beneath the seabed, represent layers up to hundreds to thousands of metres beneath the surface, see [6]. Each time the vessel reaches the end of a line, it turns and records the next line. In this way a 3-D volume is built up line by line on a grid. For a good overview of the subject refer to [7].

The second objective was the benchmarking of the Cayenne cluster upon which the modified and parallelised code is designed to run. This was a secondary aim but some useful information about the basic behaviour of the Cayenne cluster has been presented.

1.2 Organisation of the Dissertation

This dissertation is divided into eight sections covering the work carried out during the project and the subsequent period of interpretation and analysis that followed.

Section two, Project Overview, explains the project proposal and some of the motivations behind the project, the bigger picture and the project background. The initial plan and how that plan was actually implemented in reality is discussed in some detail. In that explanation, the justification for the project, its aims and an overview of the criteria established for project completion, the acceptance testing, are reviewed. A brief overview of the expectations for the benchmarking of Cayenne and a comparison of how the benchmarking of Cayenne actually happened during the project. Following that, a reappraisal of the original project time-scales and critical tasks with respect to how these were actually implemented during the real project work schedule.

Lessons learned will be outlined briefly to discover if any improvements could be implemented during a project. Next, a summary of how the original SVD code worked before any changes were applied. Followed by a discussion about the expectations before the code was upgraded compared with the actual process of upgrading and modifying the code during the project. Following on from that, an outline of the parallelisation process and what aspects of parallel I/O were investigated. To round off the section an investigation of how optimisation and performance testing were used to determine the effectiveness of the parallelisation process.

---

4 An in-line is a collection of traces shot in the sailing direction of the vessel towing the streamers. There is a closer relationship between data traces within the same in-line than between traces in a cross-line. The cross-line direction is orthogonal to the in-line direction. Modern acquisition techniques have somewhat blurred this relationship. However, it is still valid to think in terms of in-line traces as being closely coupled and cross-line traces as being loosely coupled. This has implications for the SVD code parallelisation process.

5 Although not strictly true, reflections occur where there is an abrupt contrast in transmission properties such as the velocity of sound. The relationship is close enough to accept that these are layers. However, seismologists can use this to distinguish between oil and water in rock layers, since the change in density between oil and water is sufficient to allow a reflection to occur.
Section three is concerned with the detailed analysis of the process of porting and upgrading the original SVD sequential code to Cayenne, with special attention given to the issues encountered with compilers and development tools used on Cayenne. The areas of analysis breakdown into the following. Setting up the development environment, the issues and the problems encountered during that set up process. Finally, an in-depth discussion of the changes made to the serial C and Fortran code and the implications, side effects and improvements provided by these changes.

Section four, deals with the process of parallelisation and optimisation used on the SVD code, after the code had been upgraded and ported to Cayenne. Followed by a brief coverage of the design and execution of the performance testing on the parallelised code on Cayenne and Ness. Following on from the ideas presented in sections two and three, development issues, problems encountered with the development environment and the machine environments with respect to the parallelisation of the C and Fortran codes.

Section five is about the process of profiling and testing the Cayenne cluster with standard, off-the-shelf benchmarking packages. Two packages were investigated first the INTEL IMB tools and then the HPL open source package HPL version 2.0. This section also covers the process of sourcing the benchmarking packages the INTEL performance testing packages in particular. Finally, a critique of the installation and testing processes with examples of running the profiling tests. The results of the benchmarking and performance testing are discussed with an emphasis on Bandwidth, Latency, Memory and I/O performance of Cayenne.

Section six was written as a stand-alone document that was used as a description of the acceptance testing specifications for the parallelised C and Fortran codes. This section describes how, during the development and parallelisation process, the code was to be tested against synthetic and real data volumes of varying size and shape. It also acts as a source code layout and content rules specification. There is a brief description of how synthetic test data volumes are created and the use of code written to make 3D volumes from existing 2D seismic lines. The requirement for testing SVD code with the Nelson data, after conversion to a 3D volume, is explained. Followed by a paragraph on the speed improvement specification is added for the serial version of the code and a similar specification for the parallelised version of the code. The section ends with a specification for code usability and maintainability, which is necessary to facilitate maintenance of the code during future development work.

Section seven, is a thorough description and discussion of the performance results obtained from the parallelised SVD code on Cayenne. The section ends with a look at how the results compare for a small 64 line synthetic data volume run on both Ness and Cayenne.

Section eight, conclusion and overview, summarises the results and outcomes of benchmarking and discusses any improvements to the methods and procedures used during the project. Followed by a short discussion of future work that could be done to improve the code, the performance testing and benchmarking on Cayenne.
2. Project Overview

In this section the project goals and the rational for the methods used to achieve those goals are discussed. The background theory and the bigger picture are briefly examined. The machine specifications and software limitations encountered are documented in each section as they occur but any major systemic problems are also mentioned here. A brief discussion of the main themes of the following sections is also undertaken here to bring the document together. To narrate progress through the project work this section chronicles the project work, describing step by step how expectations of the software and hardware, obtained before work started, materialised. Briefly, the issues and benefits discovered when setting up this software and using it are over-viewed.

2.1 Background

The business case for this kind of project is compelling but there is also a cogent scientific rationale for the use of 4-D seismic time-lapse methods in dynamic, “real-time” reservoir management [8]. The Edinburgh Time-lapse Project (ETLP) [9] provided the SVD code for this project. It is part of a group of programs used to study the Earth's subsurface [11]. ETLP is a consortium of universities and energy companies interested in exploiting 4-D seismic data to improve their understanding of reservoir behaviour. The hope was to achieve faster and more immediate assistance with hydrocarbon exploration and secondary recovery in existing “in production” wells. The adoption of reservoir modelling techniques will have long-term environmental, strategic [10] and social impact through its effect on the availability and price of energy. What is achieved now, with enhanced remote sensing techniques, will help to shape the future of hydrocarbon exploration and production for the rest of this century.

Projects like this are becoming more important as hydrocarbon resources dwindle and oil field management becomes ever more vital. The economics of oil reserve and energy security are now issues that must be answered quickly. As reserves further decline, and the size and lifetime of reservoirs become shorter, then software and machines that can deliver rapid results will attract more funding from the energy industry.

Many industrialised countries have already depleted their own reservoirs and yet their demand for hydrocarbons continues to increase. Any developed country with oil and gas reserves must take advantage of these techniques and use secondary recovery on their own depleted wells [12]. Secondary recovery is a long established technique, that can work extremely well provided the CO2 or water are injected into the appropriate place. Therefore, an understanding of how the fluid dynamics of the reservoir operate over its lifetime has become a necessity. If injection is applied in

---

6 Secondary recovery is the process of pumping down either water, CO2 and even polymers into a reservoir to push more hydrocarbons out of the main well.
the wrong place then unwanted side effects will occur: reserves may be lost and very large sums of money wasted.

Seismic 4-D time-lapse analysis, in conjunction with other remote sensing methods, and modelling will provide data to aid secondary recovery and monitor its activity over long periods of time. A series of 4-D time-lapse surveys could be used to track the movement of injected gases or water and can detect pressure variations caused by reservoir compartmentalisation. Often this leads to untapped oil pockets that cannot be accessed from existing wells. Extraction of these pockets requires expensive infill drilling and sometimes even horizontal drilling to gain access to these small but economically important reserves.

Another use of this technique is for monitoring well condition, well performance and forecasting future reservoir reserves. The ability to forecast future reservoir performance and generate well depletion plans will greatly extend the life of existing oil and gas deposits and aid energy planning for governments and corporations around the world.

2.2 Aims

The main aim of this project was to optimise and parallelise an existing SVD serial code. This code was written with a mixture of C and Fortran 77 in the late 1990's. The Fortran code was contained in a single file etlpsvds. The code used Singular Value Decomposition (SVD) to generate three or eleven output volumes, depending on command-line options, from two input 3-D volumes. The industry standard SEGY format is used for both data input and data output. One of the main design considerations was to preserve this SEGY format after parallelisation. The original code was designed to run on Linux or Sun SPARC platforms, however, the newly parallelised code was only designed to run on Linux machines such as Cayenne and Ness. On modern UNIX machines Linux code should run without modification except for the compilation settings used in the ancillary Makefiles. The major barrier to portability are the differing locations of libraries and variations in the versions of MPI compilers.

A subsidiary aim was to continue using the existing functionality but to modernise the code to Fortran 95 then to parallelise it for use on Cayenne and Ness. This has worked very well but with some surprising lessons learned about the ordering of the development work. The idea of doing all of the modernisation in one step followed by the parallelisation was a good idea in theory. However, the process of serial code update hid bugs that did not become apparent until the parallelisation step was started. A good example of this was the failure of the code to read data on any process except the master process (rank 0), see section.

---

7 Controlled Source Electro-Magnetics (CSEM).
8 Compartmentalisation is caused by variation in porosity, mineralisation and changes in permeability.
9 Makefiles are part of the make utility. Make automates the compilation of all files that have been updated. Only the most recent versions of code files are linked to the main program. This is done without requiring the user to perform each task separately. The user just types make <target>. 
The method of parallelisation chosen was Message Passing Interface [13] (MPI); optimising it to match the characteristics of both Cayenne and Ness. In practice the implementation was minimal. The only MPI functionality used was the initialisation, getting the number of processes, getting the current rank ID and finalising MPI at the end of the job. Data are not passed between processes and each process handled its own data IO independent of the other processes during the entire run.

Another aim of was to study the use of parallel I/O or MPI-IO [14] on parallelised SVD code. These were studied and various MPI-IO libraries were found to be available, for example “An Open-Source MPI-IO Library” (OPAL) [15], which is a portable Linux library. One difficulty investigated was how to output SEGY\(^{10}\) format with MPI-IO code. The MPI-IO library has two preferred formats machine native and binary format. The main problem with using MPI-IO would have been the necessity to convert, whatever MPI-IO format was used internally, to either IBM\(^{11}\) or IEEE\(^{12}\) floating point for the SEGY format. After much experimentation and studying how IO worked on Cayenne, using its lustre\(^{13}\) virtual disc technology, it was decided not to use MPI-IO or any other parallel I/O library. The solution was to allow each process simply read the input data direct from the SEGY input files with an offset calculated from the process rank number. Each process works on its own exclusive set of data that it writes out to its own set of SEGY formatted output files. This is actually a most elegant solution thanks to the organisation of traces\(^{14}\) as sets of in-lines in a SEGY volume. The smallest unit of data each processor handled was a complete in-line\(^{15}\) set of traces, which made data IO a simple in-line read-write operation. Simply put, because the in-lines do not need to be reconstructed, the IO overhead is considerably reduced.

The final aim was to benchmark the Cayenne\(^{16}\) cluster using standard benchmarks. The architecture of this cluster is ideally suited to the way the parallel code,

---

\(^{10}\) SEG Y format was developed by the Society of Exploration Geophysicists in 1973 to store geophysical data on tape.

\(^{11}\) IBM is International Business Machines Inc.

\(^{12}\) IEEE is the Institute of Electrical and Electronics Engineers

\(^{13}\) Lustre is a high performance, parallel, virtual, distributed file system used in large scale clusters. It is available under a GNU GPL license.

\(^{14}\) The most basic definition of a seismic trace is an array of amplitudes stored as a set of floating point numbers. However, a trace in terms of SEGY format consists of a 240 byte binary header that stores meta-data about the trace. Immediately followed by trace data stored as a discrete set of bytes representing either IBM or IEEE Floating point values. These represent the original amplitude of the waveform recorded by the hydrophones. More commonly though, the trace in a SEGY volume is the product of several steps of seismic data processing and actually represents a Common Depth Point (CDP). A CDP is the sum of the trace data corresponding to the same common depth point but originating from different seismic shot-points at different offsets [16].

\(^{15}\) In-lines are collections of traces that are grouped in the shooting direction along a seismic line. In terms of land seismic surveys this is literally a line on a map. In marine terms this is the direction of sailing.

\(^{16}\) A beowulf cluster of 448 cores, on 56 dual socket processors, each processor being an AMD quad core Opteron processor.
developed as part of this project, is constructed and used. The simple Beowulf layout, see Figure 2, is an excellent configuration to run parallel code that has little or no communication between processes. This naturally lead to the idea of developing a data parallel\textsuperscript{17} decomposition for the code used in this project. Another, more compelling, reason, to choose a data parallel decomposition, was because of the SEGY data format. SEGY volumes are collections of seismic in-lines that have little or no dependency on the orthogonal, “cross-line\textsuperscript{18}” traces on either side of them. This makes communication between processes unnecessary, which is ideal for a large scale data parallel decomposition on a Beowulf cluster like Cayenne.

![Figure 2: Typical Beowulf Cluster Layout [17]](image)

As part of the process of understanding how the Cayenne cluster works and how the code written for this project could best be deployed. It was thought necessary to understand the memory and communications performance of the cluster before parallelisation began. In practice, two standard testing tools were chosen HPL (High Performance Linpack) version 2 [18] and the INTEL MPI Benchmarks (IMB) version 3.2.2 [19]. For various reasons, mostly lack of documentation and poor error reporting the HPL product failed to install and run. The product was tried on both Ness and Cayenne but on both machines the same error occurred. Although the INTEL IMB software installed correctly it was difficult to run on the Cayenne system. These issues will be fully discussed in Section 5.

\textsuperscript{17} A data parallel decomposition is similar to a task farm but it has no source or master process that generates a pool of worker jobs. There is also no sink process that consumes the results. One or more processes are allocated a set of seismic in-lines to process. The processes do not communicate with each other or with a master process. They handle their own IO and do not share data or variables, see [20] for details on data parallel decomposition.

\textsuperscript{18} See footnote 15, Cross-lines are orthogonal to in-lines. Often the physical distance between cross-lines is different from the distance between the in-lines in the same volume. They are in effect imaginary lines.
2.3 Discovering How the Existing SVD Code Worked

This was not a main aim of the project but still an essential part of the discovery process both before the project started and over the full term of the project. The original SVD code was written in Fortran 77 and standard C. The code used Singular Value Decomposition to generate a time-lapse image of an oil reservoir. One seismic volume is literally subtracted from the another leaving a residual signal, which is taken to be changes in position of layers between surveys. SVD was used because just subtracting matching samples from each other, for a pair of corresponding traces just amplifies random noise, see Table 1. Basically the SVD technique is used to reduce random noise in these data and extract the differences between two 3-D volumes as a set of “singular values” and vectors that represents the desired signal. The process is very CPU and memory intensive and during the testing phase, see Section 7., it was shown that 99% of the run-time was taken up with the process of applying the SVD calculation. Studying the existing sequential SVD code, compiling and running it on Cayenne highlighted the following facts. The original code works on Cayenne without modification and produces credible results. The serial code is fast enough to process 2-D seismic lines and very small 3-D data volumes up to a few megabytes or work with a small SVD window in a reasonably fast time without the need for parallelisation. However, if time is not pressing very large volumes can be processed for periods up to 24 hours\(^\text{19}\) on Cayenne.

2.4 Lessons learned: Assessment of Project Planning

The original work plan [21] was divided into three logical stages with a list of dependencies that required completion at each stage. There was a very good correspondence between the plan and how work actually progressed during the project. The Gantt chart shown next in Table 2, reveals those dependencies graphically. It was intended that the three tasks shown in steps one to three could be run in parallel with each other. Some parts were run in parallel, except the acceptance tests that were completed ahead of the “Sequential Code Update” and “Cayenne Benchmarking”. Nonetheless, there were some dependencies on the critical path, which were predicted in a critical path analysis before the project started.

It was thought that the parallelisation should not start until the serial code modernisation was completed, however, this was a mistake and the plan did not work as well as expected. Some critical defects were discovered during the

\(^{19}\) The Cayenne cluster batch system has a limit on the length of time jobs may run. This is currently set to 24 hours.
parallelisation stage, which could have been identified faster and more easily had some parallelisation been done earlier in the project.

The writing of the dissertation was expected to start as soon as the parallelisation process commenced. In practice the dissertation has been written in stages throughout the project as and when notable outcomes were achieved or problems discovered.

<table>
<thead>
<tr>
<th>Start Of Project Milestone</th>
<th>23/05/11 08:00</th>
<th>30/08/11 17:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upgrade Sequential Code to FORTRAN 90</td>
<td>23/05/11 08:00</td>
<td>16/06/11 17:00</td>
</tr>
<tr>
<td>Contingency for machine down-time</td>
<td>16/06/11 08:00</td>
<td>24/06/11 17:00</td>
</tr>
<tr>
<td>Agree Acceptance Tests with Team at Herts</td>
<td>23/05/11 08:00</td>
<td>30/06/11 17:00</td>
</tr>
<tr>
<td>Benchmark Cayenne</td>
<td>23/05/11 08:00</td>
<td>01/07/11 17:00</td>
</tr>
<tr>
<td>Write weekly notes for Dissertation</td>
<td>23/05/11 08:00</td>
<td>05/08/11 17:00</td>
</tr>
<tr>
<td>Code parallelisation</td>
<td>24/06/11 12:00</td>
<td>08/08/11 13:00</td>
</tr>
<tr>
<td>Ensure Data is Correctly Transformed</td>
<td>11/07/11 08:00</td>
<td>05/08/11 17:00</td>
</tr>
<tr>
<td>Optimisation and Performance Testing</td>
<td>24/06/11 12:00</td>
<td>08/08/11 13:00</td>
</tr>
<tr>
<td>Contingency for machine down-time</td>
<td>05/08/11 08:00</td>
<td>12/08/11 17:00</td>
</tr>
<tr>
<td>Write dissertation</td>
<td>01/07/11 08:00</td>
<td>19/08/11 17:00</td>
</tr>
<tr>
<td>Write dissertation</td>
<td>01/07/11 08:00</td>
<td>19/08/11 17:00</td>
</tr>
<tr>
<td>Prepare for Presentation</td>
<td>20/08/11 08:00</td>
<td>30/08/11 17:00</td>
</tr>
<tr>
<td>Presentation</td>
<td>20/08/11 08:00</td>
<td>30/08/11 17:00</td>
</tr>
</tbody>
</table>

Table 2: Original Gantt Diagram for Project

Some contingency time was allowed to cope with unexpected delays, particularly in the “Upgrade Sequential Code in Fortran 95” stage, in case the modernisation process overran. In practice, all the contingency time was taken up with other problems not related to the project. Despite these problems the project was completed in reasonable time and all the major aims were satisfied. The least successful was the Cayenne Benchmarking, which has only been carried out to a basic level, see section 5.

2.5 Project Risk Assessment

At the start of the project five risks to the project time scale were identified. Each risk was named, had an estimate of severity assigned to it and, if obtainable, a contingency that could be used should the risk be realised.

The most severe risk was the “loss of access to Cayenne”, which was assessed as very severe. With hindsight, that should have been a fatal severity as the total loss of Cayenne would have made the results of the project of doubtful value. Even if the contingency of using Ness was acted upon. One of the main aims of the project was running performance testing on the parallelised SVD code on the Cayenne cluster. As the primary aim of the project this would have been at least a sever impediment to the project outcome. The SVD code, even if made to work on Ness, is really intended to be used on Cayenne, so this contingency does not satisfy the aims of this project. Fortunately this did not occur. However, working on Ness in conjunction with Cayenne brought the benefit of faster debugging, especially with issues concerning compiler switches, library locations and versions.
The next most severe risk was if the Fortran 95 code update caused a major failure. This was thought to be a medium risk and the contingency was to revert to an older copy of the code with a backup contingency. This was achieved by storing versions of the code under SVN\textsuperscript{20} to act as a fall-back. During the project there were several occasions where the serial code update process did lead to code that did not work or had unexpected side effects. Reverting back with SVN, or just doing a comparison with an earlier version of the code, lead to almost instant recovery from problems of this nature. Using SVN with Eclipse was especially useful for this sort of development situation. Eclipse and SVN aided versioning overcame many of the potential pre-project risks identified saving many days, even weeks, of development work.

Another medium risk was that the parallelisation process would cause code failure. The contingency was to explore other methods of parallelisation that would eliminate the problem. In practice this kind of problem happened on more than one occasion, however, an alternative method of parallelisation was found to be unnecessary. The root cause of most bugs were due to bugs inherent in the original code or through lack of variable sharing between individual processes. All of which were resolved by standard debugging and programming methods available in Eclipse. In other words the MPI, data parallel decomposition method of parallelisation did not need to be replaced and caused no problems in itself. The final code created has proved to be very robust, flexible and fast.

The next risk was “Inconsistent results generated on Cayenne”, which was considered to be a medium risk. The solution suggested for this was to run the test on Ness to make a comparison and to rerun standard tests on Cayenne. This risk never materialised, however, one data volume of 64 in-lines was tested on both Ness and Cayenne to make a comparison between the performances of the two machines. This gave an interesting result, which is discussed in section 7.2.

Finally, the risk that MIP-IO may fail to work was identified as a low risk with the recovery option of investigating other methods of parallel I/O or by code redesign to avoid the problem. In the actual project this situation did not arise as MPI-IO was not used at any point. Although it was looked at as a possible method of writing to a single output file it was not thought necessary because of the unique characteristics of the SEGY file format discussed earlier in this section.

\textsuperscript{20} SVN or Subversion is an open source version control system now under the Apache Foundation Umbrella. It is free, multi-platform but still mostly used under Linux and UNIX. It is a client – server application. The server has a database that stores information about the versions in the repository. It is widely used and gives very good control over source code versioning and security.
3. Porting and Upgrading of SVD Sequential Code to Cayenne

This section is a full description of the methods used to modernise the existing serial SVD code supplied at the start of this project. It is about the problems encountered and the solutions found in modernising this code. The code includes any ancillary scripts and the compilation and development tools used with it. Also any methods used to decompose the original code before modification and any automated refactoring solutions provided.

The Fortran code was mostly modernised by hand but as part of Photran an automated refactoring feature was made available in Eclipse, this will be explored later in this section. The core of the Fortran code is very efficient already and uses the LAPACK [22] library functions to do the SVD calculation, see section 3.3.2.

Studying the “man pages” on Cayenne revealed a basic library of hardware and software profiling tools already present. This included gprof for performance analysis at the function level but the Eclipse IDE made this and most of the other external tools unnecessary. However, to maintain portability and to prevent over reliance on Eclipse, the make files on Cayenne and Ness were designed to compile the code stand-alone as well. Hence, users not wishing or unable to use Eclipse could use any text editor they preferred and compile the code by running “make all” on the command-line.

3.1 Compilers and Development Tools on Cayenne

There were several MPI libraries available on Cayenne for C, C++, F77\(^{21}\) and F90 in the system directory /usr/mpi/intel/openmpi-1.2.6/bin/mpi90. During the project it was necessary to find and evaluate these MPI libraries and check compatibility with standard Fortran 95 compilers. Investigations showed that Cayenne had two working compilers available, which were gfortran and mpi90. Both compilers worked with Fortran 95 code and were readily available on the front-end and back-end\(^{23}\) of Cayenne. The compiler's were both tested by running some simple code containing statements only supported by Fortran 95. The tests proved that both gfortran and mpi90 worked correctly with these Fortran 95 statements.

---

\(^{21}\) F in F77, F90, F95 and F2003 stands for Fortran

\(^{22}\) At the very end of the project the gnu compiler was updated to version 4.6.1 and is now found in /usr/local/openmpi/1.4.2/gnu/4.6.1/. Note: It is unwise to rely on any compiler remaining the same after the close of a project. These paths usually end in a wrapper application, in this case one called opal_wrapper, which actually points to a library file in /usr/mpi/intel/openmpi-1.2.6/lib64.

\(^{23}\) Front-end and Back-end refers to the front-end, which is used to run non-parallel interactive applications and the back-end where users submit a batch job to run on multiple processors on the “back-end”.

12
There were some changes in the compiler and libraries made at various times throughout the lifetime of this project. There are several different library locations, compiler versions and compiler options to consider when compiling these codes on Cayenne and Ness. Ness was sufficiently different from Cayenne that it became necessary to separate the original Makefile into two make files one for Cayenne (Makefile) and one for Ness (Makefile.ness). This has simplified the make files considerably but control of what is compiled is still carried out by editing the appropriate symbolic links in the appropriate make file and recompiling the code. An example of this is the conditional compilation of either serial or MPI executable versions of the code. To “compile out” MPI code and its header files “mpi.h” the conditional compiler symbol “-DMPI_Parallel” is commented out, in the make file, by adding a “#” to the start of the line. This preserves the modified serial code so it can be run, without the need to find MPI libraries, should the user wish to run it on a non parallel enabled machine.

CFLAGSMP    =-DMPI_Parallel  #Run code in parallel
#CFLAGMP    =-DMPI_Parallel  #Run code in serial

However, on a parallel enabled machine this is not necessary as running on one processor adequately simulates a serial run, see Figure 16.

3.2 Development Environment Set-up and Issues Encountered

The chosen development environment was Eclipse with additional “perspectives”\textsuperscript{24}, such as Phrotan 6.0.6, C++ Development Perspective (CDC), Subversion 1.6.17 and PTP for parallel MPI development. These perspectives allowed the development of C and Fortran code to be tightly integrated in a simplified environment with the bonus of cross-language testing and debugging. There was also the tremendous asset of being able to control version management within Eclipse using the subversion perspective. With its synchronisation options it saved code loss on a number of occasions and allowed simple version comparison to help solve bugs. The essential feature of the environment was the close integration of tools, which enabled faster and more reliable development. All, development took place on Cayenne, with regular testing with synthetic seismic data. After SVN repository update the code was always recompiled and tested with a standard synthetic seismic data volume. The process of keeping the code synchronised on Ness and cayenne was of enormous benefit especially when there were problems on Cayenne. If there were bugs with no obvious cause then Ness gave a different perspective on them, which accelerated their resolution.

3.2.1 Installing Eclipse

The first job was to install Eclipse 3.6 Helios, however, Cayenne already had Eclipse installed. Nevertheless, the installation procedure was run through to check the correct version of Eclipse had been properly installed.

\textsuperscript{24} Eclipse “plug-ins” are called perspectives.
The version of Java\textsuperscript{25} that was running on the machine was tested:

```
java -version
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07)
Java HotSpot(TM) Server VM (build 19.1-b02, mixed mode)
```

Using this method, it was verified that Cayenne was running Java 1.6, which satisfied the requirement of Java 1.5+ to run Eclipse Helios. However, it was necessary to check that Cayenne was also running the correct Java SDK\textsuperscript{26} version as well and that it was set up in the PATH, LD_LIBRARY_PATH environment variables and in the correct order. A test was performed to make sure that “make” and “gfortran” (GNU Fortran compiler) were runnable on the system. Ness and Cayenne already had these set up, the following test was run to make sure they were present:

```
Make:
[s1061391@ness src]$ make -version
GNU Make 3.81
Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
This program built for x86_64-redhat-linux-gnu

gfortran:
[s1061391@ness src]$ gfortran -v
Using built-in specs.
Target: x86_64-redhat-linux
Configured with: ./configure --prefix=/usr --mandir=/usr/share/man
--infodir=/usr/share/info --enable-shared --enable-threads=posix --enable-checking=release --with-system-zlib --enable-cxa_atexit --disable-libunwind-exceptions --enable-libgcj-multifile --enable-languages=c,c++,objc,obj-c+
+java,Fortran,ada --enable-java-awt=gtk --disable-dssi --disable-plugin --with-java-home=/usr/lib/jvm/java-1.4.2-gcj-1.4.2.0/jre --with-cpu=generic --host=x86_64-redhat-linux
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-50)
```

Noting the version of “gfortran” (4.1.2) helped to find the correct libraries on Cayenne and Ness minimising the set up time. One of the early project design considerations was that the compilation would be available without having to run Eclipse. Hence the correct version and directory location of these libraries a vital part of the the development process. These tests were also important when Eclipse was not being used especially on Ness, which did not run eclipse at all. It was also worthwhile running to be sure that all compilers and tools were running in 64 bit mode\textsuperscript{27}. These prerequisites were met on both Ness and Cayenne ultimately making the installation process for Eclipse trivial. Eclipse was only installed on the Cayenne cluster but it was accessed remotely across the network from various client

\textsuperscript{25} Java is necessary to run Eclipse and Eclipse requires a specific minimum version of Java (1.5).

\textsuperscript{26} Java may run on any machine without the Software Development Kit SDK, because it is run from the JRE, but Eclipse needs the SDK to be present to run. Normally Eclipse is used to develop Java code but it has perspectives for C/C++, Fortran and even Visual Basic.

\textsuperscript{27} Just because an application runs on a 64 bit machine, does not mean that it is running in 64 bit mode. In fact many applications still run in 32 bit mode without any appreciable problems on a 64 bit machines
machines, which gave outstanding flexibility and little or no maintenance overhead.
Installation instructions for Eclipse Helios 3.6 are available from http://www.eclipse.org/downloads/index.php and should be used should re-installation be required.

3.2.2 Setting up Photran on Eclipse

Photran was not present on the Cayenne copy of Eclipse so installation was necessary. It was a simple installation but required further set-up to work with the correct gfortran and mpi90 Fortran compilers. This task was eased by setting up the pSVD project with the “standalone make file” option in Eclipse. This allowed the supplied make file to be run whenever the project was built in Eclipse, as well as on the command-line on Cayenne or Ness. It is worth noting this because being certain of libraries linked in is vital for correct executable operation. This was particularly true when running in batch mode on the Cayenne back-end. A summary of the steps required to install Photran are shown below:

1. Eclipse was started
2. Next Help was clicked and the “Install New Software” option selected
3. Then the Add button was clicked
4. Next the URL http://download.eclipse.org/tools/ptp/releases/helios was typed in to the Location field
5. OK button was clicked
6. “Fortran Development Tools (Photran)” was chosen, the check-box beside “Photran End-User Runtime” was checked
7. "Fortran Compiler Support" was opened and the desired compiler selected, (this was the gfortran compiler identified section 3.2.1)
8. CLICK Next
9. Finally, the Finish button was clicked after ticking the agree to the license box
10. That was the end of the installation process

The installation was then verified by compiling a simple Fortran “hello world” application, using a standard template in Eclipse and adding:

    WRITE (*,*) “Hello, world”

If this ran then the gfortran compiler was working correctly in the Eclipse environment.

After setting up the code repository on Ness, the code in the SVN “pp_repository” was connected to Eclipse using the “SVN perspective”, section 3.2.4 describes this process in detail. This connection is necessary to keep the code on Ness and Cayenne synchronised.
3.2.3 Setting up code repository on NESS

For security, to share the load on hard disk storage and to keep the code in-house\textsuperscript{28} the SVN repository was set up on Ness. This was part of the original project plan and made sense at the planning stage. However, this did cause some problems, especially later in the development cycle. As Ness uses a disk quota, for each user, the amount of disk space available is limited. This proved troublesome towards the end of the project when the repository had grown to 300Mb. It restricted other work that could have been done on Ness and meant that very large test volumes could not be run on Ness at all. This was not too much of a problem, however, as Cayenne was the intended target for testing. Ness was only used as a back-up and to run small comparison tests matching those already done on Cayenne.

3.2.4 Setting up SVN and Linking to the Code Repository on NESS

Ness had a suitable SVN server and using the command “svnadmin create pp_repository” it was set up. This created an empty repository to which the existing SVD code, as supplied in a tar-ball\textsuperscript{29}, was added. After logging on to Cayenne a test was performed to make sure that Ness was available on Cayenne via SSH\textsuperscript{30}:

\begin{verbatim}
(1)svn checkout svn+ssh://s1061391@ness.epece.ed.ac.uk/home/s02/s1061391/pp_repository pSVD
\end{verbatim}

This “checked out” a version of the empty repository in the pSVD directory. The contents of the tar-ball were then unpacked in to the pSVD directory. One mistake made here was not first editing the directory structure to remove unnecessary files and directories before adding the repository contents. In the topmost directory “~/pSVD”, the command “svn add *” was executed. This added all the original code files to the repository. Finally, the command “svn commit” was executed committing all the added files to the repository. A \texttt{vi}\textsuperscript{31} session opens on the command-line and the comments section was filled in with some useful comments describing the changes to the version being committed. This was done with this

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure3_Eclipse_Project_Set-up.png}
\caption{Eclipse Project Set-up}
\end{figure}

\textsuperscript{28} If there were access problems or loss of internet links, it was more likely to happen when connecting to an external site.

\textsuperscript{29} A tar-ball is a type of archive file that uses TAR (Tape Archive) and is “zipped up” with a compression tool like GNU gzip. The files often have the suffix .gz or sometimes .tgz.

\textsuperscript{30} ssh \texttt{username@ness.ed.ac.uk} if this works then access to Ness is available.

\textsuperscript{31} vi is a standard editor in Linux and can be guaranteed to be found on Linux machines.
entry, the information stored can still be viewed with the command “SVN log”. The log shows that the original code was added to revision 4 “r4” on February 20th 2011.

```
r4 | s1061391 | 2011-02-20 21:37:14 +0000 (Sun, 20 Feb 2011) | 2 lines
Repository for original source code
```

At any time during the project, the pristine original code could be extracted and used or modified. Any intermediate version between revision r4 and the current version could be extracted as required.

The procedure for getting the SVN repository in Eclipse was handled with dialogues but the process was essentially the same, this will be explained in the next subsection (sub-section 3.2.5).

### 3.2.5 Creation and Execution of C and Fortran code in Eclipse

Created a new project to contain the SVD code. It is important to get this right so command-line “make files” can be used and to make sure the C and Fortran debugging work smoothly together. The process begins by setting up the project as a “C/C++ make file project” and then getting the code from the repository, see Figure 3, which shows the set up of a “C/C++ make file project” called pSVD. Notice the option selected in the project type text area is “Makefile project”, this is the essential step. Language is set by the file suffix, the toolchain32 and in the manually written make files. Do not set the language using one of the other Eclipse project type options or Eclipse will make its own internal “make file” based on Ant. Ant files are difficult to export to a Linux environment and it uses inbuilt Eclipse tools that may not be available on the target Linux platform.

32 The toolchain is how Eclipse builds in sets of external tools like compilers into its environment.
The next step was to open the SVN perspective and checkout, to the new project file on Cayenne, a copy of the repository on Ness, containing the SVD code, see Figure 4. Notice, that the same SVN URL, which was used in the command-line set up, is also used here (1). After completion of this, the SVN code was available for editing. The first new code to add were the two make files Makefile and Makefile.ness that controlled all the compilation. Once the make files were created and the original sequential SVD code was compiled and tested, changes to the code were started, see section 3.3 for details.

Eclipse is an excellent tool to run serial codes especially for testing and debugging. It also provides a useful profiling tool, giving an overview of where time is spent during execution. There is also a choice of run and debug modes that can be chosen simply by clicking on the appropriate icon. Unfortunately, the PTP perspective, intended to work in a, parallel MPI development environment, failed to work and the manuals were very poor. In a short project, such as this, it is not worthwhile trying to fix poorly written code with poor documentation. This perspective could be explored a longer term project as a piece of future work.

Although, the built-in debugging tool in Eclipse is excellent it is worth debugging the code on the command-line with “gdb” as well. It was necessary to use gdb on Ness anyway because Eclipse was not installed on Ness. It is also worth using write statement based debugging with strategically placed write statements to write

---
33 gdb is a GNU open source debugger, which comes with gcc and gfortran. It is an integral part of the GNU compiler's library of functions and tools.
information to the console at various points in the code execution. With the use of compiler switches it is simple to set the debug level desired, which controls the detail of the debug results shown. Three levels of debugging were created, controlled by the compiler switches -DDebug, -DDebug_Full and -DDebug_Complete. Notice how the commands are nested to make all debug code, below the chosen level, visible on the console when the code is run.

```c
#define Debug Compilation options
#define NO_DEBUG =
#define SHORT_DEBUG = -DDebug
#define LONG_DEBUG = -DDebug_full $(SHORT_DEBUG)
#define COMPLETE_DEBUG = -DDebug_complete $(LONG_DEBUG)
```

### 3.2.6 Portability of Eclipse Development Environment

The portability issues with Eclipse were avoided by using hand-made make files and setting up all the library locations and compiler versions in the make file with raw machine library paths. Major portability issues were avoided by not using Eclipse built-in paths at all. Compiling and running the code on Ness and Cayenne generated some dependencies that were solved by having different make files for each machine.

Eclipse has syntax checking tools to help write better code, however, they did not work well with Fortran code under Phortran and did not reformat previously written C code. New code was formatted, as set out in the Eclipse format specification and old code was manually modified to match the new code.

### 3.3 Code Changes

Stage one was principally about upgrading the existing C code to C99 Standard [23] and the Fortran 77 code to Fortran 95 standard [24]. The intention was to make the code more reliable, portable and, within the scope of scientific programming, easier to maintain. Improvements to Fortran's syntax since the code was originally written have been employed to make the code more efficient and more human-readable. This has improved the robustness and stability of the code and optimised for portability, which enabled it to run more efficiently on both Cayenne and Ness.

The original intention was to add JUnit style test modules, similar to those used in Java but as time was short and no suitable, stable and well tested product could be found, testing was limited to standard debugging. The use of synthetic and real data volumes made this kind of testing immaterial. Modules, see [25], were added to make the Fortran code more robust and to enable the protection of local variables in critical subroutines. The use of Fortran “Internal Files” was explored as a way of obviating the need for parallel I/O. However, this became unnecessary due to the use of fseek() in the C code and the availability of a virtual, distributed file system called Lustre. Lustre enables a single file to be accessed in parallel with little or no contention.
3.3.1 Changes to Serial C Code

The first step was modernising the code to use fully labelled function prototypes in header files. The proper use of header files ensures that accidental redefinition\(^{34}\) does not take place, which on some compilers would give spurious errors. Adding header files was needed to aid the creation of new specialist functions for SEGY IO handling and a function for testing and allocating stack space in gen_utils.h, see Figure 7. The division of functions by usage and using header “.h” and .c source files gave the code enhanced maintainability and made function sharing possible. Notably, the SEGY_util.h functions are shared by the etp_svd_64bit code and the build3DdataSet\(^{35}\) code. Later, during the parallelisation process, more functions containing specialist MPI calls were added to this code.

Another problem with the original C code was a long history of updates by various programmers who used different layout styles and varying degrees of commentary. Eclipse was used to refactor the code to make it more compliant with C99 and standardise the layout. Non-tabbed indentation was set up, replacing tab characters with 4 spaces throughout the C and Fortran code. This ensured that when the code was moved between editors or edited with vi that the indentation and comments stay in the correct position. The format of comments, file headers, subroutine headers, function and block headers was standardised. Comments had boxes put round them if they were used to comment on major sections of the code. A consistent method of laying out function calls was adopted to make them more readable and have related variables grouped according to their function. Where code was new, a header was added with the name of the author, when it was created and what function it performed.

3.3.1.1 Additional Code

The first new code added was the timing statements, which were added to the original code before any major alterations were applied. The `<time.h>` functions were used to time the “total run-time” of the SVD code, the time spent doing “I/O operations” and the time spent doing the actual “SVD calculations”:

```c
/* Actually start timing now */
time(&start_time);

/* Here Full time and IO time start at the same time */
start_io_time = start_time;

/* Stop IO Timer Here and accumulate the time */
time(&end_io_time);

/* Accumulate IO time */
accum_io_time += (double)difftime (end_io_time, start_io_time);
```

Every time a section of IO code fread() or fwrite() was encountered the IO timer was started and at the end of that code section it was stopped immediately. The time

\(^{34}\) Modern compilers will give warnings or, in some cases, an error but this should not be relied upon. Older compilers may or may not even warn about redefinition.

\(^{35}\) Added by the author to generate the artificial 3D volumes from 2D lines.
accumulated was then added to a holding variable “accum io time”. At the end of the job that accumulated total run-time was output. The same was done for the SVD code, which is called every time the trace read loop is run. However, the start and stop timers for SVD run-time were conveniently placed around the call to the Fortran SVD subroutine etlpsvdsepar:

```c
/* Start SVD Timer */
time(&start_SVD_time);
etlpsvdsepar(
    &n, &nSamples_r, &equal, &M, &EN, &LWORK, 
    rchunk_r, rchunk_m, 
    dataR_E1, dataR_E2, dataR_E3, dataR_E4, 
    dataM_E1, dataM_E2, dataM_E3, dataM_E4, 
    dataR_geol, dataM_geol, 
    data_t1
);
/* End SVD Timer Here */
time(&end_SVD_time);
/* Accumulate SVD time */
accum_SVD_time += difftime(end_SVD_time, start_SVD_time);
```

One change that was thwarted by the inability of C to transfer dynamically allocated arrays using arralloc() (via the Fortran etlpsvdsepar subroutine call. Fortunately, using the SVN version control option in Eclipse, it was easy to back out of the unsuccessful code changes. The change was an attempt to remove the static arrays, rchunk_r[nChunk][nSamps],..., data_t1[nChunk][nSamps], which would have simplified changing the number of traces and number of samples in a dataset. The code to use dynamic arrays in Fortran was successful and worked well with a Fortran harness that sent data similar to data sent by the C application:

```c
REAL(KIND=sp), INTENT(IN), DIMENSION(nSamples, nTraces) :: refData
REAL(KIND=sp), INTENT(IN), DIMENSION(nSamples, nTraces) :: monData
```

The code, shown above has been retained in the Fortran 95 code but the complementary C code was removed and the static arrays were reinstated. Figure 6, shows the difference, generated by Eclipse and SVN, between revision 36, with arralloc(), on the left and revision 55, reverted back to static arrays, on the right.

<table>
<thead>
<tr>
<th>/<em>Arrays to store output data</em>/</th>
<th>/<em>Arrays to store output data</em>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataA_E1=arralloc(sizeof(float),2,nchunk,nSamps);</td>
<td>float dataR_E1[nchunk][NAXSAM];</td>
</tr>
<tr>
<td>dataA_E2=arralloc(sizeof(float),2,nchunk,nSamps);</td>
<td>float dataR_E2[nchunk][NAXSAM];</td>
</tr>
<tr>
<td>dataA_E3=arralloc(sizeof(float),2,nchunk,nSamps);</td>
<td>float dataR_E3[nchunk][NAXSAM];</td>
</tr>
<tr>
<td>dataA_E4=arralloc(sizeof(float),2,nchunk,nSamps);</td>
<td>float dataR_E4[nchunk][NAXSAM];</td>
</tr>
</tbody>
</table>

Figure 6: Comparison Memory Assignment arralloc() or array[]

The Fortran Allocated arrays work because they have the dimensions passed to the etlpsvdsepar() subroutine as nSamples and nTraces in the argument list. However, the values sent to nSamples and nTraces from the C code, must be defined in the C code at compile time, to allocate space in the static arrays and when passed to etlpsvdsepar().

---

36 arralloc() is a function provided by EPCC. It is used to allow multidimensional C arrays to retain a contiguous memory layout, similar to a static array[][]...[] with multiple dimensions.
Code to allow job control parameters and input files to be specified on the command-line was added in preparation for parallelisation. The code was added in a way that did not prevent the existing input methods from working, which meant that a check for command line parameters had to be made when the code was run:

```c
if (argc < 4)
   
else {
   printf("%s", argv[3]);
}
```

If argc is less than four, i.e. if there are not four or more arguments on the command-line, then the printf statement is executed and the user must interactively input the missing value(s). There is an advantage to being this precise because partial command-line lists can be set up allowing the user to change some parameters at will.

The last major added code was included, in the C code, very late in the project. After the C code was reverted back, by SVN, to the static array version, see Figure 6. As a consequence of reverting to static arrays the memory available for the stack size became an issue. Rather than ask the user to set the stack size on the command line it was thought neater and more elegant to create a new function to do that whilst the code was running. So setStackSize() was created, see Figure 7. Some of the test volumes, synthetic and real data, were causing the code to “segmentation fault” immediately the application started but thanks to the use of the compilation parameter -fbounds-check, the bug was quickly located. The solution was to create a new function, that was called right at the beginning of the main () function, that checks the memory size of the stack. To do this it needs to know the amount of memory37 required by all the stack arrays used in the application:

```
int setStackSize (int reqStackSize) {
    struct rlimit rl;
    rlim_t rStack = reqStackSize;
    
    if (0 == (getrlimit(RLIMIT_STACK, &rl))) {
        if (rl.rlim_cur < rStack) {
            return (setrlimit(RLIMIT_STACK, &rl));
        }
    }
    return 0;
}
```

Figure 7: New code to deal with stack overflow with stack arrays[][], [37] In bytes.
stackSize = (MAXSAMPS * MAXTRACE * floatSize * numWorkArrays) +
    (fileNameLen * numFiles * charSize); /* Need at least this stack size */

if((setStackSize (stackSize + margin)) != 0) {
    printf("Unable to process this volume, stacksize is too small\n");
    printf("Ask your systems administrator to increase the sack size to");
    printf(" at least %d bytes\n", stackSize);
    exit(-1);
}

If the application cannot get the memory required the job aborts with a message to
the user. If a 3D volume with a large number of traces per line and a large number of
samples per trace is run then memory can become a problem. It does not matter how
many in-lines the 3D volume contains because the application processes just one in-
line at a time. The application will not run if the number of samples and the number
of traces per in-line cause it to exceed the maximum stack size that can be allocated.

3.3.1.2 Code changes

There were no specific changes required for pre-parallelisation. The only
requirement was for clean and comment code. The MPI calls themselves were
already written by the author in C for a previous project and reused with slight
modification in this code. The rational was to make it easy to conditionally compile
the MPI code in or out of the application; essentially keeping the code serial by
effectively removing the MPI code.

For this #ifdef/#endif blocks were used in conjunction with the make file symbol $(CFLAGSMP), which contained the compiler switch -DMPI_Parallel to turn the
parallel code “on and off”.

#include MPI_Parallel
    if((mpiError = init_mpi(argc, argv, &mpiParm.size)) != SUCCESS) {
        printf("Unable to initialise parallel MPI (mpiError = %d)\n", mpiError);
    }
#endif // End of MPI_Parallel section

Other make file symbols used include the debugging flags used to control “write to
console” style debugging.

#include Debug_complete
    printf("(ctr < nChunk\n");
#endif

To enable the processing of 3D volumes with multiple in-lines. The outer while loop
code was changed to handle the number of in-lines to process, along with its
ancillary code to count the number of lines processed.

/* Find traces independent of lines but process in line chunks */
while(fread(headR.uc, trHeadLen, charSize, fps_in_r) &&
    fread(headM.uc, trHeadLen, charSize, fps_in_m) &&
    (lineCount < stripWidth)) {
    ...

The line count is incremented every time an in-line of traces (a “chunk”) has been
processed and the stripWidth is calculated at MPI initialisation to hold the number of
in-lines processed by each processor, which non-parallel runs is set to the number of
in-lines in the seismic volume (MAXLINES).
3.3.2 Changes to Serial Fortran Code

The core of the etlpsvds.f95 code is the call to the LAPACK function DGESVD, which applies the SVD method to the reference\(^{38}\) and monitor\(^{39}\) seismic data.

\( \text{(2)} \text{CALL DGESVD('A', 'A', M, N, A, LDA, lambda, U, LDU, VT, LDVT, WORK, LWORK, INFO)} \)

The subroutine, in (2) is used to compute "singular value decomposition". The subroutine is called with a set of double float M*N arrays of data from the reference and monitor seismic volumes. The mathematical behaviour of subroutine (2) is shown in equation (3):

\( A = U \ast \sum \ast \text{transpose}(V) \)

Where:

- \( A \) is a matrix for computing the left and right singular vectors.
- \( \sum \) is a matrix of \( M, N \) values that are all zero except for the diagonal, which contains the value of \( \min(m,n) \)
- \( U \) is an orthogonal matrix of \( M, M \) values
- \( V \) is an orthogonal matrix of \( N, N \) values

The "singular values" of "A" are the diagonal values of the \( \sum \) matrix. These are a set of real, non-negative double values in descending numerical order. The first columns of the \( U \) and \( V \) matrices are the left and right singular vectors of "A" respectively. The function returns \( V^\text{T} \) as \( VT \) (\( V \) transposed), which is another \( N, N \) matrix.

See the following references for detailed information about the SVD methodology. For LAPACK information use the LAPACK user guide [22], for the specific mathematical background see [26] and for a basic introduction to the subject [27] and [28] for more details. This code has been largely left in its original form, just some minor cosmetic changes to the layout and enhanced commentary. The rest of the Fortran code has been radically changed by the author to modernise it to Fortran 95 standard. However, there have also been some new subroutines and functions,

\(^{38}\) The reference data set refers to the oldest seismic volume; shot earlier than the monitor data set and the monitor data set refers to the younger data set shot after the reference data set.

\(^{39}\) See previous footnote 38
added to simplify the code, remove repeated sections of code and to make the code more durable and easier to maintain.

3.3.2.1 Refactoring Required to Update Code to Fortran 95

Initially the Fortran code etlpsvdsdpalr.f was refactored using the partially automated Eclipse refactoring feature, see Figure 9 and Figure 8, which shows the four options used. The first option, “Introduce Implicit None...” added the statement to the start of all program, function and subroutines in the Fortran code immediately exposing problems poor coding. The “Remove Unused Local Variables” option proved to be most useful there were more than a few unused variables to be removed. Finally “Standardise Statements” was run, which made all statements upper-case (lower case was also possible) and correctly formatted them for Fortran 95. This was good because it showed clearly which statements were built-in Fortran statements.

This was followed by a manual refactoring of the code, which included re-indentation to make the code more readable. Modernisation of comments from 'C' style to the new '!' delimiter.

Frequent retesting was required as part of this process to ensure that the SVD algorithm was not damaged or if any artefacts had been introduced into the processing. The speed of execution was also monitored, which ensured that the modernisation steps did not make the code run more slowly than the original code.
The following new features were added to the Fortran 95 code.

- Modules: A module called setPrecision was added, which set the size and precision of float and double real variables. This was a small but important subroutine. If the desired precision could not be obtained the calling program exited.

```fortran
MODULE setPrecision
IMPLICIT NONE
SAVE
INTEGER, PARAMETER :: sp = SELECTED_REAL_KIND(6, 37)
INTEGER, PARAMETER :: dp = SELECTED_REAL_KIND(15, 307)
END MODULE setPrecision
```

- Kind: Kind is a new feature that essentially sets the precision of variables, usually FLOAT and DOUBLE precision sizes. REAL(KIND=sp) sets the precision to FLOAT or Single Precision.

- Allocatable arrays: REAL, DIMENSION(:, :), ALLOCATABLE :: array. At run time, the actual bounds for the array 'A' may be determined by the statements sent either in an argument list or on the command-line at run-time. Or use ALLOCATE (array(N, N), STAT = AllocateStatus) then the status can be checked to prove that allocation has been successful and if not an error message is sent back to the user after exiting the program, for

SELECTED_REAL_KIND(p,r) returns a real data type with decimal precision P (or better) digits and an exponent range r or greater.
example.

**IF (AllocateStatus /= 0) STOP "Allocation failure in array"** Although these were used in the arralloc() version of the code, see sub-section 3.2.4. Ultimately these were removed because the calling C code could not send the correct pointers for a 2 dimensional array using arralloc(). The Fortran code failed to interpret the pointers correctly.

- **INTENT**: Added to specify if an argument was receiving data in only, outputting data only or both in and out:
  
  `REAL(KIND=sp), INTENT(OUT), DIMENSION(ns, nT) :: outA_E1`

- Elemental types: These were explored but they failed to work correctly with allocatable arrays. Arrays with an allocated size at compile time will work correctly but if the size is not known until run-time, as in the arralloc() version of the code, the elemental operations fail. It is thought that the compiler gfortran fails to recognisable the problem. So, it is not until the code is run that the problem occurs. The code executes forever probably because number of elements to process is unknown. There is some evidence in the literature that similar bugs have been discovered in gfortran with allocatable arrays but not this specific bug, so, following up on this would make interesting future work.

- Built-in types: Built-in types like SQRT(), SUM() were tried in an attempt to reduce the complexity of the mathematical statements used in the code. However, there were very few places in the code where they would have been useful. The SUM() intrinsic function had problems with elemental operations on allocatable arrays and so was replaced with a do loop to sum array elements instead.

- New subroutines: equaliseData() was added to improve the reuse of code that calculated RMS averages. This code is now reusable and, since it is applied to both the reference and monitor volumes, is now guaranteed to be applied to both input volumes identically.

- More comments: The original code lacked important comments that made clear what was being processed and more importantly why it was being processed. That has now been rectified.

- Updating the syntax: see Figure 11, for the new layout style, which is essentially to use upper case standard Fortran statement names and variables. Using a mixture of upper and lower case characters to generate more meaningful names that reflect their use.

### 3.3.2.2 Improvements to Code

A new feature, setPrecision() was added to the code, which checked that variables of the correct precision can be allocated. If the requested variable “kind” with the required accuracy cannot be allocated then the job is aborted. The values of sp and dp are set up in the module setPrecision, page 26, which has replaced the concept of real and double in this code.
! Check single/double precision accuracy requested
! If not exit, Incorrect floating point accuracy.  

IF (sp == -1) STOP &
  & 'Accuracy of single precision numbers is too low'
IF (dp == -1) STOP &
  & 'Accuracy of double precision numbers is too low'

Some lines of code that were no longer used were removed most had previously been commented out or in defunct sections of code or were place-markers added by previous developers.

The code had the number and detail of comments increased. With better, fuller explanations and the reason for an action the code became easier to maintain. An explanation of why code is there is as important as what it does, for example:

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!* Before we can start work on the SVD calculations *
!* We must reorder the input C data into Fortran array order *
!* For this we have a set of internal Fortran arrays *
!* that we populate with the C data and simultaneously *
!* convert it to double precision required by the *
!* library function DGESVD *
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

### 3.3.2.3 Testing of Code

This was an essential part of any development process especially when altering an existing code like the SVD code, see section 6. Fortunately the original code was kept to act as a reference to test against with the new code. The new Fortran 95 code had to give exactly the same results as the original code and was not allowed to run any slower than the original code. So, in combination with error checking and performance testing, the optimisation process continuously fed back into the upgrade process.

The Fortran code had no special timing code added to it but relied on the timing added to the C code. The overall run-time for the application as a whole was the critical factor, not just the time spent in the Fortran part of the application. However, the Eclipse profiling tool was used to understand how long the code took to run in each function and subroutine. The longest time was spent in the main etlpsvdsepar() subroutine, which does the actual SVD calculation.

Accuracy was unaltered from the original code. However, based on using the new Fortran 95 functions SELECTED_REAL_KIND(6, 37) for a single precision float and SELECTED_REAL_KIND(15, 307) for double precision float, the accuracy could be guaranteed to be implementation independent\(^{41}\). That means no matter how float or double is implemented on a different machine or with a different compiler,\(^{41}\)

---

\(^{41}\) Not strictly true, some compilers on some machines do not implement this feature as well as others. Fortunately both Ness and Cayenne do implement this correctly.
this explicit definition of accuracy ensures correct data. If this accuracy is not attainable then the run is terminated.

The fidelity of seismic dataset output was tested in a number of ways. First the traces in both input datasets have an RMS equalisation applied. The first test is to run the same data with the original code and then with the new code, checking that the RMS value is the same\[42\] in both cases. Both old and new SVD code contains the following statement:

\[
\text{WRITE}(*,*)'\text{Value of rmsmean = ',rmsmean}
\]

The synthetic data sets are run on the new code and compared with the same data sets run on the old code. The synthetic data sets are cleaner with simpler idealised waveforms compared to real data, which has noise, harmonics, convolution effects and processing artefacts already present. So SVD processing artefacts are easier to see in synthetic data than they are in real data.

If the synthetic data shown no unexpected effects the real Nelson data set was run through the new code to make sure the code could handle the original data provided with the original code. For 3D volume testing the Nelson 2-D in-line supplied was run through build3DdataSet application to build a 3D volume of as many in-lines as required.

Some of these tests can be run concurrently the RMS testing with real data or synthetic data and the timing of jobs. The run-times should be no slower than the original code running the original data under the same conditions on the same machine.

\[42\] The same within the limits of the data type. Double precision in this case.
4. Parallelisation, Optimisation and Performance Testing of SVD code on Cayenne

This covers the whole process of parallelisation from identifying blocks of code that will benefit from parallelisation to devising methods to test the performance and the accuracy of the parallelised code. Stage two, of the project, was intended to be concerned with the parallelisation part of the project, however, some of the bugs discovered by the parallelisation process were already present in the serial code.

4.1 Development Environment Issues Encountered

The main issue was the failure of the “PTP perspective”, which was designed to work with an MPI parallel environment in Eclipse. There was not project time available to explore this issue in depth but getting this to work on Cayenne could be an interesting and useful future project.

4.1.1 Parallelisation and Execution of C and Fortran code in Eclipse

The code was branched before parallelisation to retain the original sequential code in a pure sequential state, at this point the name of the project file became pMPI_SVD. In the end, the methodology used to parallelise the code was a simple process of dividing the input volume into sets of in-lines so that they were evenly shared out between the all processors. The use of MPI was restricted to getting the number of processors and the current rank. The “size” or number of processors was used to calculate the number of in-lines per processor and stored in a variable called “stripWidth”. The initial read position on the SEGY input files was calculated using the processor rank ID, which gave a unique start pointer, used by fseek(), for each processor. This code was easily written and tested in Eclipse without having to work in a parallel environment. It was, however, necessary to leave the Eclipse environment to test the code in parallel. In that instance the standard etlp_svd_64bit.sge script files were used to good effect.

4.1.2 Portability of Eclipse Development Environment for Parallelisation

What is produced by Eclipse is completely portable because the make files created for the project were designed to work outside of Eclipse. It would have been difficult if not impossible to export the code from a standard Eclipse project. The code itself is portable because it follows the standards laid down for the C and Fortran languages. The constant testing in Eclipse, on the Cayenne command-line and on Ness guaranteed that this code would be portable.

The other barrier to portability, the MPI library, had been effectively “ring fenced” by putting all the raw library calls inside a set of MPI wrapper functions. Also, by using a C structure to contain all the necessary information passed between these MPI functions the wrapper function could be completely isolated from the serial
code. This was confirmed by testing the code with the MPI functions compiled out, using conditional compilation statements, see section 3.3.1.1.

4.2 Code Parallelisation

The method of choice was data parallel decomposition, which was implemented using a standard GNU openMPI library suitable for both Cayenne and Ness. MPI calls were added using function wrappers and conditional compilation flags to take advantage of portability whenever possible. Some testing on Ness was also carried out to ensure portability and to ensure that work could still progress even if Cayenne became unavailable. Ness was always intended as the fall-back option and in practice was used as a verification machine to test the code when unexpected errors occurred during development. One of the risks outlined in the risk section of the feasibility study [21], was if Cayenne should become inaccessible then Ness was to be used as the back up machine. However, in practice the verification process was more important than the back up requirement. As code was developed, the ability to also test it on Ness lead to quicker bug fixing and the avoidance of machine dependencies in the solutions obtained. So the MPI parallelisation was designed to work on both machines with no code modification. The only differences were in the make files that were used to compile the code and in the location of the LAPACK and MPI libraries linked with the code. To ease this situation two makes files were written one for Ness ans one for Cayenne.

4.2.1 Investigate Parallel I/O and the Use of MPI-IO

MPI-IO is a set of methods for handling parallel I/O. It requires defined data types that have data order dependencies and it has portability and scalability issues. Eventually a simpler and more reliable method of making I/O run in parallel was devised avoiding MPI-IO completely. MPI-IO, performance is sensitive to buffer sizes, blocking factors, the number of nodes used for I/O, MPI implementation and system parameters. There were also issues with differences between node topologies on Ness and Cayenne to consider. The biggest concern was how MPI-IO would deal with the SEGY file format used by the seismic volumes.

4.2.1.1 Additional Code for Parallelisation

The MPI calls themselves were already written by the author in C for a previous project and reused with slight modification for this code. The idea was to make it possible to conditionally compile the MPI code into the application or to leave the application essentially serial by not compiling the MPI code into it.

For this #define/#endif blocks were used in conjunction with the make file symbol $(CFLAGSMP), which contained the compiler switch -DMPI_Parallel to turn the parallel code "on and off".

```c
#ifdef MPI_Parallel
    if((mpiError = init_mpi(argc, argv, &mpiParm.size)) != SUCCESS) {
        printf("Unable to initialise parallel MPI (mpiError = %d)\n", mpiError);
    }
#endif
```

............

31
4.2.2 Changes to Parallel Fortran Code

There was no code changes required explicitly for parallelisation and there was no parallel code added to the Fortran code. There were no side effects from the Fortran code when running in parallel.

4.3 Testing of Code

This was an essential part of the parallelisation process and done combination with the error checking and performance testing gave powerful feedback into the parallelisation process. The testing methods were exactly the same as the serial methods. Each volume was compared with a volume run in serial mode and the RMS values checked for a match with the serial version.
5. Benchmarking Cayenne

This section is about the profiling and testing with standard packages on Cayenne and a discussion of the tools used, how they performed and what difficulties were encountered in their use.

5.1 Sourcing Test Packages

Both test packages are readily available by download from the internet, see [18] [19].

5.1.1 INTEL Performance Testing Packages

There is a complete set of packages available from the INTEL web site [19], which contains a confusing array of documentation and products. The correct one to choose is the “Intel Cluster Toolkit 2011 for Linux OS.” To download it requires a 30 day trial licence that must be signed up to. The download give a tar-ball containing the code, license, a collection of manuals and about a gigabyte of extraneous files and documents. Confusingly the tar-ball downloaded is called ics_2011.0.013.tgz. However, buried in that is the IMB benchmarking tools.

5.1.2 HPL Open-Source Package

HPL provides timing and testing code to determine accuracy and records the time it takes to solve a random dense linear system with double precision numbers on computers with distributed memory. It is freely available, portable and is an implementation of the High Performance Computing Linpack Benchmark. It must have an implementation of MPI 1.1 or later to link with and, for this project, a link to the BLAS\textsuperscript{43} library. It is a more basic product than the INTEL IMB tools but does not require a 30 day trial agreement.

5.2 Installation and Testing of Performance Test Packages

Both packages we easy to download and the installation was fairly easy. Again documentation was in the case of HPL sparse and had no contingency if problems were encountered. The download file chosen for HPL was hpl-2.0.tar.gz.

5.2.1 Installation Process

After unpacking HPL, find the INSTALL file, which gives very basic installation instructions. First create the Make.<arch> file with settings to match the hardware, processors and OS used on Cayenne. This file was found in the subdirectory “set-up”. For Cayenne the best choice of architecture file was “Make.Linux_ATHELON_CBLAS”. A copy of this file was put in the HPL root directory. Now the “Make.Linux_ATHELON_CBLAS” file was edited to change the

$\textsuperscript{43}$ Basic Linear Algebra Sub-programs
location the MPI library in the $Mpdir symbol, i.e. MPdir =
/usr/local/openmpi/1.4.3/gnu/4.6.1 replacing /usr/local/mpi..

Install the hpl product by running “make arch=Linux_ATHLON_CBLAS”, which
immediately fails with a “Make error 2”, because it cannot find libmpich.a in
/usr/local/mpi. This was because the compiler makes another copy of the installation
directory under the users $(HOME) directory, calling it hpl and, ignoring the new
path added during the set-up proceeds to fail again. Next attempt, moved to the ~/hpl
directory and re-edited the Make.Linux_ATHLON_CBLAS” file and tried again.
This time the architecture file in ~/hpl is run again
“make arch=Linux_ATHLON_CBLAS”
This time it fails because $(HOME)/netlib.ARCHIVES/Linux_ATHLON does not exist.
There is no explanation about why this is needed and how to get a copy.
This chain of events continued for some time but the installation was eventually
abandoned as there was so many prerequisites missing from Cayenne and the
instructions were so poor that progress stopped. Instead the more complete and
professionally produced INTEL IMB product was installed. So, the HPL product
was abandoned in flavour of the INTEL tool.

Intel installation was a simple self-contained script with multiple-choice questions.
The basic requirement was to install as root for all users or as any user for own use.
The rest was about library file locations. After this simple installation a directory
named “intel” was found in the user $(home) directory. Full installation process is in
the appendix, see Appendix 1.

5.2.2 Testing the Packages for correct installation

No tests were done for HPL as it did not install and so work on it was abandoned.
Testing intel IMB required another “make” procedure to make the IMB executables.
First step was to go to $(HOME)/intel/imb/3.2.2/src and run “make all”, see
Appendix 1. for details.

5.2.3 Setting up the Initial Test Routines

None were carried out for HPL.

INTEL IMB provides three procedures, which were run as automated benchmarks
by the following commands

    mpiexec -np 8 IMB-IO
    mpiexec -np 8 IMB-EXT
    mpiexec -np 8 IMB-MPII

The first IMB-IO tests the IO Bandwidth of the cluster, the next, IMB-EXT test the
message passing overheads of two sided MPI functions and IMB-MPII test the
single sided functions.
5.3 Running Profiling Tests

The most complete results were obtained with MPI-MPI1 for single sided performance testing. This gave interesting results for the all the benchmarks tested. The IMB-IO tests failed to run correctly because they could not get a file lock with the version of NFS being used on Cayenne. This problem could be explored in future work but may require an upgrade of the NFS system software.

5.4 Profiling Test Results

The IMB-MP1 results were as follows, see Table 3:

<table>
<thead>
<tr>
<th>Message Size (bytes)</th>
<th>Broadcast</th>
<th>AllToAll</th>
<th>ScatterV</th>
<th>Scatter</th>
<th>GatherV</th>
<th>Gather</th>
<th>AllGatherV</th>
<th>AllGather</th>
<th>Reduce Scatter</th>
<th>Reduce</th>
<th>AllReduce</th>
<th>Exchange</th>
<th>SendRecv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.09</td>
<td>0.39</td>
<td>0.18</td>
<td>0.09</td>
<td>0.18</td>
<td>0.09</td>
<td>0.13</td>
<td>0.15</td>
<td>0.16</td>
<td>0.17</td>
<td>0.19</td>
<td>0.36</td>
<td>0.33</td>
</tr>
<tr>
<td>1</td>
<td>0.09</td>
<td>0.39</td>
<td>0.20</td>
<td>0.18</td>
<td>0.20</td>
<td>0.18</td>
<td>0.21</td>
<td>0.19</td>
<td>0.21</td>
<td>0.22</td>
<td>0.22</td>
<td>0.40</td>
<td>0.39</td>
</tr>
<tr>
<td>4</td>
<td>0.09</td>
<td>0.39</td>
<td>0.19</td>
<td>0.19</td>
<td>0.25</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.24</td>
<td>0.19</td>
<td>0.19</td>
<td>0.40</td>
<td>0.40</td>
</tr>
<tr>
<td>8</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.23</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>16</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>32</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>64</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>128</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>256</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>512</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>1024</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>2048</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>4096</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>8192</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>16384</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>32768</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>65536</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>131072</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>262144</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>524288</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>1048576</td>
<td>0.09</td>
<td>0.39</td>
<td>0.21</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>2097152</td>
<td>0.11</td>
<td>0.46</td>
<td>0.22</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>4194304</td>
<td>0.14</td>
<td>0.50</td>
<td>0.23</td>
<td>0.19</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.19</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
<td>0.42</td>
<td>0.42</td>
</tr>
</tbody>
</table>

Table 3: MIB-MP1 All Test Results

The MPI-IO runs gave a bandwidth for IO of:

# For nonblocking benchmarks:
# Function CPU_Exploit obtains an undisturbed
# performance of 4103.33 Mflops
# performance of 4111.13 Mflops
# performance of 4127.71 Mflops
# performance of 4103.33 Mflops
# performance of 4087.66 Mflops
# performance of 4105.01 Mflops

This gave an average I/O bandwidth of 4106.36 Mflops. The rest of the IMB-IO test failed because of the following error:

35
File locking failed in ADIGI_Set_lock(fd 3,cmd F_SETLKW/7,type F_WRLCK/1,whence 0) with return value FFFFFFFF and errno 26.
If the file system is NFS, you need to use NFS version 3, ensure that the lockd daemon is running on all the machines, and mount the directory with the 'noac' option (no attribute caching).

The IMB-EXT benchmarks produced a lot of output but the labelling is extremely poor. Therefore it is difficult to determine what these figures actually mean and which test they belong to. The one set of data that has some meaning is the NON-AGGREGATE Mode Benchmarking for an Accumulate benchmark, see Figure 10

![NON-Aggregated MPI Message Calls](image_url)

**Figure 10**: IMB Ext Benchmark Non-Aggregated MPI Message Calls

### 5.4.1 Bandwidth
The average IO Bandwidth was determined to be 4106.36 Mflops based on the results form IMB-IO.

### 5.4.2 Latency
Shortest latency time was found in the broadcast benchmark in the IMB-MPI11 benchmarking code and was 0.06 Micro-seconds.

### 5.4.3 I/O
The test for IMB-IO were incomplete due to problems with the NFS file system on Cayenne, however, a value for the IO bandwidth was obtained see subsection 5.4.1
6. Acceptance Testing of Parallelised Code with Synthetic and Real Data

Here the acceptance criteria have been laid out. What tests were performed, how the test data was obtained and any observations about the results of testing.

6.1 Source Code Layout and Content

The requirement is for clean, clear easy to maintain code that has meaningful and relevant comments about the purpose and methodology of the algorithm used by the code. Where possible the code shall include a description that can, if required, become separate documentation using doxygen or a similar documentation tool.

6.1.1 C Code

The C code shall conform to the standards laid down in the Eclipse Standard Style Preferences for C and C++ [29]. The use of block delimiters (curly braces) may be used either under-line or end-of-line but shall not be mixed in the same C file.

At all times obfuscation shall be avoided. The use of semicolons ';' to separate lines of code shall always be followed immediately by a new line. Multiple code statements shall not be put on a line, for example.

\[
A = B + C; \quad D = A + C; \quad C++; 
\]

Spaces and newlines shall be used to enhance the readability and understandability of code. If a line of code must be continued over more than one line, the code shall be indented in such a way that the relationship between each line is clear. If possible, the code should be divided between lines to emphasize the meaning of variables. For example, all input variables could be on one line and returned (call by reference) variables on another line. If variables are called in pairs or triplets they shall be divided up with this relationship mind.

\[
\text{MyFunction ( input1, input2, input3, input4 } \\
\quad \text{ output1, output2, } \\
\quad \text{ xValue1, yValue1, zValue1, } \\
\quad \text{ xValue2, yValue2, zValue2 } \\
\quad \};
\]

Also, whenever the argument list of a function is longer than a line the round brackets should be used like block delimiters to make the code easier to understand and look neater.
6.1.2 Fortran code

Fortran 95 layouts shall be those set out in the Phrotan free-form editor as shown in

```fortran
/*******************************************************************************/
/* equaliseData() */
/* dataIn  INTENT(IN) (Kind=dp) Input Array seismic trace */
/* dataOut INTENT(OUT)(Kind=dp) Output Array seismic trace */
/* nTraces INTENT(IN) INTEGER number of seismic traces in data */
/* nSamples INTENT(IN) INTEGER number of samples per trace */
/* */
/* Subroutine to compute the data equalisation array */
/* Author : Alex Hunt */
/* Date : 19/06/2011 */
*******************************************************************************/
SUBROUTINE equaliseData(dataIn, dataOut, nTraces, nSamples)

USE setPrecision

IMPLICIT NONE

! Subroutine dummy arguments
INTEGER, INTENT(IN) :: nTraces
INTEGER, INTENT(IN) :: nSamples
REAL(KIND=dp), INTENT(IN), DIMENSION(nTraces, nSamples) :: dataIn
REAL(KIND=dp), INTENT(OUT),DIMENSION(nTraces) :: dataOut

!*******************************
/* Local Variables */
!*******************************

! Temp array to hold rms values during calculation
REAL(KIND=dp), DIMENSION(nSamples) :: amplitudes

! Temp double to hold the calculated rms value
REAL(KIND=dp) :: rmsout

! Loop counters
INTEGER :: i
INTEGER :: j

! Init temp sample array before use
DO i=1,nTraces
    amplitudes(i) = 0.0_dp
END DO

!*******************************
/* Load amplitudes of the input data */
/* for each trace in turn */
!*******************************
DO i=1,nTraces
    ! Take amplitude data out of reference data set
    DO j=1,nSamples
        amplitudes(j)= dataIn(i,j)
    END DO
    ! Now do the RMS Equalisation
    CALL getRMS(ampitudes,nSamples,rmsout)
    dataOut(i) = rmsout
END DO
RETURN
END
```

Figure 11: Example of Optimal Fortran 95 Code Layout

the Phrotan 5.0 Users Guide [30] and in [25], which exemplifies the rules laid out in
[30] highlights the major layout features required. All subroutines and functions
shall have a main comment header that shall contain at least the following items:

- Authors name

38
• date of creation or modification,
• purpose
• a list of arguments and return types

This shall be kept updated whenever code changes are made.

Each dummy argument shall be fully defined with an INTENT() and all arrays
DIMENSIONED and fully shaped.

REAL(KIND=dp), INTENT(IN), DIMENSION(nTraces, nSamples) :: dataIn

The INTENT keyword shall always be used to protect variables in the argument list
of the subroutine or function call from being accidentally overwritten unless they are
expected to be updated. In that case an INTENT(OUT) shall be used that clearly
shows the argument is to be updated when the subroutine or function exits. For more
details, please see, [25].

6.2 Synthetic Test Data Creation

A series of synthetic seismic datasets shall be used to test code operation and the
varsity of the SVD algorithm. This shall be carried out after all changes to the code
as part of the test regime. It is not enough that the code runs, it must produce the
correct results within tolerance. The most basic test is to check the “Value of
rmsmean” printed out when the program terminates. This should be the same to
within 1.0x10-6. The original code outputs this as a floating point number
0.961773919 for the Nelson data set, with RMS setting set to 3. The same setting on
the new code, which outputs the value as a double, gives 0.961773918678851,
which is well within the 1.0 x10-6 tolerance.

There are three synthetic data sets provided as part of the test package that
accompanies the code. They must all generate acceptable results. The most obvious
result is running the same synthetic data set as both the reference data set and the
monitor data set. The result should be a “blank” time-lapse output file. There will be
some random noise, but the signal must be completely cancelled. If there are any
remaining coherent data present then a problem has occurred.

If there are problems with the code, the most likely outcome will be failure to output
any data. There are other criteria to consider as well. The code must not run slower
than the original serial code supplied. So, timing code has been added to the main C
section of the application. After, every change to the code the time for a full run shall
be compared to the time taken for the original serial code. Due to difference in
machine loading, this test shall be run on the original serial code and the modified
code within one hour of each other. In both cases the same data set shall be run with
the same settings.
6.3 Acceptance Testing With Real Nelson Data Set

This must be carried out after the synthetic data tests have been run. The synthetic data shall expose any processing artefact if they are present. These are much harder to see in real data like the Nelson volume. The requirement is to QC (Quality Control) the resulting output data. It should be an exact match with the data generated with the original code.

6.4 Serial Code Speed Improvement Specification

Code run-times shall be compared after each modification, the speed of execution shall be no longer than the original code under the same conditions. The original code has a run-time of 3.00 Seconds, the new code should be no slower than that. Currently, the updated code is giving a run-time of 2.00 Seconds.

6.5 Parallel Code Speed Improvement Specification

A significant increase in speed is expected when this code is parallelised To test the amount of speed-up. The original code shall be run with the same data set as the parallelised code. This is to enable a full comparison of speed up and to make sure the parallelisation process has not introduced artefacts. This shall be carried out with both synthetic and Nelson data sets with an appropriate number of lines to make parallel runs efficient within the bounds of Gustafson's Law [31]
6.6 Code Usability and Maintainability Specifications

The code shall be kept in a suitable repository and shall be available to edit from the command-line or in a suitable IDE. (Integrated Development Environment). The code shall not have any dependencies introduced by an IDE that would prevent it from being edited, complies and run outside the IDE.

Wherever possible, the code shall be kept in a state that allows it to be run in a multi-platform environment. Where this is not possible the code shall have such conditional compilation directives as required to make the code portable without undue obfuscation of the code structure and without change to the expected results of testing.
7. SVD Parallelised Code Performance Results

Performance testing was carried out on both Cayenne and Ness but mainly on Cayenne. One test with the small data volume\(^{44}\) was run on both Ness and Cayenne to compare the speed-up and run-times of each machine. These results will be analysed and the performance of each machine compared and contrasted later.

A more thorough set of performance tests were run on Cayenne. The small data volume, the medium data volume\(^{45}\) and the large data volume\(^{46}\) were run for up to 35 processors. Each run was repeated five times and the median value taken. If the variation in run-times was very small (1%) then the median value was used otherwise the average of the five values was used.

7.1 Results for Cayenne

Table 4, shows the number of processors verses IO, SVD and Total run-time. The final column shows the speed-up compared with running on 1 processor. The most marked feature is the low IO run-times; these are a small percentage of the total run-time for all numbers of processors. The IO run-time varies between almost 0% and 14 %. The zero values are a function of the resolution of the time.h timing functions, which are only accurate to the nearest second. However, this accuracy was sufficient, for these tests, especially on the larger volumes where run-times were around 6 hours on 1 processor. The most obvious property of these speed-up values is the consistently poor performance of this size of volume on Cayenne. The reason for this is that the size of N is simply not big enough to keep

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>IO Run-time (seconds)</th>
<th>SVD Calculation Run-time (seconds)</th>
<th>Total Run-time (seconds)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>87</td>
<td>88</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>77</td>
<td>77</td>
<td>1.14</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>50</td>
<td>51</td>
<td>1.73</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>44</td>
<td>44</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>32</td>
<td>32</td>
<td>2.75</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>27</td>
<td>27</td>
<td>3.26</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>21</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>20</td>
<td>21</td>
<td>4.10</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>17</td>
<td>18</td>
<td>4.89</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>16</td>
<td>16</td>
<td>5.2</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>15</td>
<td>15</td>
<td>5.87</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>6.29</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>7.33</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9.75</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9.75</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>8.8</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9.75</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>12.51</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>12.51</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>14.67</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>14.67</td>
</tr>
</tbody>
</table>

Table 4: Full Set of Run-times For Cayenne with the Small Data Set

---

\(^{44}\) The small data volume, 3d_l64_synthetic_medium.segy, is a synthetic seismic volume with 64 in-lines, 524 traces per line and each trace contains 623 samples; the combined size including all headers being 91 MB.

\(^{45}\) The medium data volume, 3d_l112_synthetic_medium.segy, is a synthetic volume with 112 in-lines, 524 traces per line and each trace contains 623 samples; the combined size, including all headers, being 160,MB.

\(^{46}\) The large data volume, 3d_l1048_baseline.segy, is the Nelson data volume, expanded to a 3D volume of 1048 in-lines, 1048 traces per in-line and 1351 samples per trace; the combined size, including all headers, being 6.2GB.
all the processors busy. The speed-up for 35 is 14.5, which is apparently sub-linear. The explanation for this is simply the distribution of in-lines shared between processors is dependent on the following calculation:

(4) \text{ceil} \left( \frac{\text{Total Number of In-lines}}{\text{Number of Processes}} \right)

The rounding up with \text{ceil}() causes the majority of processors to have the same number of in-lines except for the last few processors, which only get what is left over. Using the small data volume with 64 in-lines and using 35 processors, gives:

(5) \text{ceil}(64/35) = 2 \ (1.82857)

Using 32 processors gives 2 in-lines to each processor and leaves 3 processors idle

(6) \text{ceil}(64/30) = 3 \ (2.13)

Using 21 processors gives 3 in-lines to each processor, 1 processor gets 1 in-line and 8 processors remain idle

(7) \text{ceil}(64/25) = 3 \ (2.56)

Using 21 processors gives 3 in-lines to each processor, 1 processor gets 1 in-line and 3 processors remain idle.

With a smaller number of in-lines per processor the chance of some processors not having enough or no work increases. The cause of this is the unavoidable use of \text{ceil}(), which rounds up to the next whole in-line and the low number of in-lines available to process. This leads to the jitter effect seen in the speed-up graph, Figure 13.

![Speed-up For Small Data Volume on Cayenne](Image)

Figure 13: Speed-up for Small Data Volume on Cayenne
Exploring the processor starvation problem further and finding worthwhile
optimisations would be an interesting topic for future work.

The medium sized data volume results are similar to the small data volume but as,
Table 5 and Figure 15, clearly show, that there is still insufficient data to work

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Run-time (seconds)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IO</td>
<td>SVD</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>151</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>76</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5: Results for Cayenne Run-time Tests on the Medium Data Volume

efficiency on the Cayenne cluster. The amount of time spent in IO is again
insignificant when compared to the SVD run-times and rarely gets greater than 10%
and is usually about 1% or less. Also, when the IO run-time value gets to more than
1% it is probably due to the inaccuracy of the timers rather than a real increase in IO
run-time, see Figure 14. Although the medium volume had 112 in-lines compared to
64 in-lines in the small volume the results showed that performance is still less than
optimum. The lack of in-lines to share between processors makes a less efficient
data parallelisation. In fact as N / P approaches 1 and especially when N%P ≠ 0, the
efficiency drops off rapidly. To work well on Cayenne with data parallel decomposition, the value of N should be very much greater than P. So if N/P = “large number of in-lines” or if N%/p=0 Cayenne will give better results. So, the worst case scenario, a small volume with a prime number\(^{47}\) of in-lines, a large number of samples per trace and a large number of traces per in-line will probably scale very badly. Again, this would make an interesting topic for future work.

---

\(^{47}\)There are 169 prime number from 1 to 1000.
So, as N increases in size, relative to the number of processors, the chances of some processors being idle is considerably reduced and eventually the speed-up becomes linear, see the trend for the large volume in Figure 15. Both the small and the medium data volumes show the same judder at 16, 20, 23 and 25 processors. However, the large volume shows no judder at those processor numbers and the speed-up is linear. Table 6 shows that the large data volume with 35 processors has a speed-up is 33.39. This is expected to continue up to and beyond the 56 processors available on Cayenne. Applying the algorithms in equation (5) to the 1048 in-lines available in the large seismic volume, the calculation indicates that the same judder should not occur until 350 processors are used. This would be a good topic for future work on an exascale machine with a very large 3D volume.

The “speed-up between P” column was added to Table 6 to show the variation in speed-up between individual processors, which shows that for some processors the increase in speed up was greater, especially between 1 and 5 processors. Speed-up is considerably better than that obtained for smaller volumes. However, The initial run-time used to derive the speed-up figures is based on the same code; with MPI running on 1 processor.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Run-time (seconds)</th>
<th>Speed up</th>
<th>Speed up between P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IO</td>
<td>SVD</td>
<td>Total</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>145</td>
<td>18721</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>3934</td>
<td>3956</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>1978</td>
<td>1986</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>1317</td>
<td>1229</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>992</td>
<td>998</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>787</td>
<td>796</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td>666</td>
<td>671</td>
</tr>
<tr>
<td>35</td>
<td>3</td>
<td>562</td>
<td>566</td>
</tr>
</tbody>
</table>

Table 6: Results for Large Seismic (6GB) Seismic Volume Showing linear Speed-up

<table>
<thead>
<tr>
<th>Run-time Kind</th>
<th>Serial Code</th>
<th>MPI Code On Front-end</th>
<th>MPI Code Back-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>SVD</td>
<td>89</td>
<td>88</td>
<td>85</td>
</tr>
<tr>
<td>Total</td>
<td>91</td>
<td>90</td>
<td>88</td>
</tr>
</tbody>
</table>

Table 7: Comparison of Run-times for Code Versions Serial, MPI on Front-end and MPI on Back-end

So as a further test, to see if this speed-up was reduced by the MPI library overheads or differing compiler settings, the small data volume was used to test the SVD code. The code was first run as a purely serial code with all MPI library calls removed but leaving the rest of the code unaltered and with the same compilation flags. The code was tested on Cayenne using the small 64 in-line seismic data volume. The serial version was run first run on the Cayenne front-end then the MPI version was also run on the front-end and finally the same MPI version was run on the back-end in full parallel mode with 1 processor. The trends shown in Table 7, clearly show little difference in run-time.48

48 These figures are based on a set of five runs for each code type, so minor fluctuations should be evened out.

46
Speed-up Comparison: Large (6GB), Medium (160MB) and Small (100MB) Volumes

An Illustration of Gustafson's Law

Figure 15: Comparison of Small, Medium and Large Seismic Volume Run-time
Comparison of Run-times for Code

Serial, MPI Front-end and Back-end Runs

Run-time Type

- Serial Code
- MPI Code On Front-end
- MPI Code Back-end

Figure 16: Comparison of Run-times of Serial and MPI Code

The increasing trend towards faster runs on the back-end with MPI, despite the difference being small, is very clearly shown in the bar chart in Figure 16.
MPI start-up and library calls can be eliminated as a cause of added overhead but the

**Comparison of I/O Run-time and SVD Run-time**

6.2 Gb Volume on Cayenne

![Graph showing comparison of I/O and SVD run-times for the large data set on Cayenne.](image)

Figure 17: Comparison of I/O and SVD Run-times for the Large Data Set on Cayenne

difference is too small to suggest using the serial code as the base run-time for speed-up calculations.
7.2 Comparison Results for Ness and Cayenne

![Comparison Speed-up on Cayenne and Ness](image)

Figure 18: Comparison Speed-up on Cayenne and Ness

One set of tests were run to compare the speed-up and raw run-times on Cayenne and Ness, see Table 8. The graph, Figure 18, shows a similar trend for both machines although the speed-up does begin to diverge at 8 processors with Ness being less efficient with more than 8 processors.

<table>
<thead>
<tr>
<th>Processors</th>
<th>CAYENNE Runtime (Seconds)</th>
<th>NESS Runtime (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IO SVD Total</td>
<td>IO SVD Total</td>
</tr>
<tr>
<td>1</td>
<td>1 87 88</td>
<td>5 214 219</td>
</tr>
<tr>
<td>2</td>
<td>0 77 77</td>
<td>4 168 172</td>
</tr>
<tr>
<td>3</td>
<td>1 50 51</td>
<td>2 119 121</td>
</tr>
<tr>
<td>4</td>
<td>0 44 44</td>
<td>0 80 80</td>
</tr>
<tr>
<td>5</td>
<td>0 32 32</td>
<td>1 78 79</td>
</tr>
<tr>
<td>6</td>
<td>0 27 27</td>
<td>1 65 66</td>
</tr>
<tr>
<td>7</td>
<td>1 21 22</td>
<td>0 58 58</td>
</tr>
<tr>
<td>8</td>
<td>1 20 21</td>
<td>0 52 52</td>
</tr>
<tr>
<td>9</td>
<td>1 17 18</td>
<td>0 52 52</td>
</tr>
<tr>
<td>10</td>
<td>0 16 16</td>
<td>1 48 48</td>
</tr>
<tr>
<td>11</td>
<td>0 15 15</td>
<td>1 40 41</td>
</tr>
<tr>
<td>12</td>
<td>0 14 14</td>
<td>1 38 37</td>
</tr>
<tr>
<td>13</td>
<td>0 12 12</td>
<td>1 36 36</td>
</tr>
<tr>
<td>14</td>
<td>0 11 11</td>
<td>1 34 35</td>
</tr>
<tr>
<td>15</td>
<td>0 11 11</td>
<td>0 36 36</td>
</tr>
<tr>
<td>16</td>
<td>0 9 9</td>
<td>1 33 34</td>
</tr>
</tbody>
</table>

Table 8: Comparison of Run-times For Cayenne and Ness
Table 9, shows in detail what is happening to the speed up at 8 processors and above. The cause will not be communication overheads because there was no communication between processors during the runs. The IO was not responsible either, see Figure 19, there is no relationship between IO runtime and speed-up, however, there is clear inverse relationship with SVD run-time.

For future work, this curious difference beginning at 8 processors could be explored and compared with benchmarks that have characterised the basic behaviour of Ness with 8 or more processors.

The behaviour of the speed-up on both machines seemed to be independent of the machine topology This is probably because of the overwhelming effect of the data parallel decomposition, which effectively makes communication and IO insignificant. Leaving

<table>
<thead>
<tr>
<th>Processors</th>
<th>Ness Speed up</th>
<th>Cayenne Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>1.17</td>
<td>1.14</td>
</tr>
<tr>
<td>3</td>
<td>1.81</td>
<td>1.74</td>
</tr>
<tr>
<td>4</td>
<td>2.74</td>
<td>2.42</td>
</tr>
<tr>
<td>5</td>
<td>2.77</td>
<td>2.76</td>
</tr>
<tr>
<td>6</td>
<td>3.32</td>
<td>3.25</td>
</tr>
<tr>
<td>7</td>
<td>3.70</td>
<td>4.0</td>
</tr>
<tr>
<td>8</td>
<td>4.21</td>
<td>4.15</td>
</tr>
<tr>
<td>9</td>
<td>4.21</td>
<td>4.88</td>
</tr>
<tr>
<td>10</td>
<td>4.66</td>
<td>5.7</td>
</tr>
<tr>
<td>11</td>
<td>5.34</td>
<td>5.87</td>
</tr>
<tr>
<td>12</td>
<td>5.92</td>
<td>6.25</td>
</tr>
<tr>
<td>13</td>
<td>6.26</td>
<td>7.32</td>
</tr>
<tr>
<td>14</td>
<td>6.26</td>
<td>8.0</td>
</tr>
<tr>
<td>15</td>
<td>6.08</td>
<td>8.3</td>
</tr>
<tr>
<td>16</td>
<td>6.44</td>
<td>9.75</td>
</tr>
</tbody>
</table>

Table 9: Comparison of Speed-up For Cayenne and Ness

![Graph: Relationship Between Speed-up, IO and SVD Run-time](image)

Figure 19: Relationship Between Ness Speed-up, IO and SVD Run-time

the amount of work (N) and its division by the number of processors the controlling factor in determining the amount of speed-up.
8. Conclusion and Future Work

The parallel speed-up needed to cope with expected increases in 4-D time-lapse data throughput was the major consideration for this project. More and more new volumes are shot every day and more legacy data are being retrospectively examined. The turnaround time of 4-D time-lapse reservoir analysis requires code that can run large volumes of data efficiently and accurately. This code is now capable of answering those requirements. The operation and efficiency of the code has been improved by several factors and it is now ready for any new data it is required to process.

Standard software and performance optimisation were also a part of this project. The benchmarking set expectations for the maximum speed-up that could be obtained using MPI. The processor topology found on Cayenne is based on a standard beowulf set up. This has limited the amount of speed-up available to an application. Cayenne has good speed-up using MPI, between cores within a processor and between processors on the same network node or physical machine. Probably, due to the slowness of the network interconnects between network nodes (1Gb/s) the speed-up under these circumstances will be considerably slower. This would make an interesting topic for future work but with a dataset that requires MPI communication across nodes. A good example of a large dataset that needs 3D communication is wavelet modelling. This would require several processors to work in 3D domains with swap halos communicating wave-front data between the 3D domains.

The software development in this project was divided in three logical stages; Firstly, the optimising and modernisation of the existing SVD code. Followed by the parallelisation of the existing, C based, SEGY manipulation code by adding basic MPI and simple parallel I/O techniques. In conjunction with that, performance testing and optimisation was carried out on the newly parallelised code. Finally, through a previously designed set of acceptance tests using synthetic and real data the code was rigorously tested before being signed off as completed.

The MPI code, added to the upgraded serial code, is a set of stand-alone subroutines and functions created by the author for general code reuse. This was done to protect the integrity of the new SVD serial code and to facilitate code update when it becomes necessary. A by-product of this is greater portability. The MPI functions are designed to keep the details of exact MPI implementation out of the main C code of the SVD application. This makes changing either the MPI or the C code a simpler task than if the code had been intermixed.

This project has gone a long way in discovering Cayenne's capabilities and hopefully will enable developers to take full advantage of Cayenne's strengths and avoid its obvious weaknesses; the possibly slow network interconnects between network nodes. However, the growing need for hydrocarbons is becoming more acute every year. So, there is still scope for yet more speed-up. In the analysis of oil and gas reserves for immediate or future exploration it is worthwhile remembering that parallel speed-up is only part of the solution and better algorithms and theory
may in the longer term be necessary. However, this update of the existing SVD software, was a vital part of the process of producing faster and better reservoir estimation and modelling packages. This project has proved useful, cost effective and has added to the knowledge base in both the geoscientific and computer science domains.

A future task, that would greatly benefit the development process is the exploration of how the Eclipse “PTP perspective” would work on Cayenne. On paper it looked like a good tool to debug and optimise code in a parallel environment and all inside Eclipse. It would be very useful to get that product working and could be great time saver when debugging parallel code.

Another problem encountered during the project would also benefit from some future work. To find the reason for the failure of Fortran allocatable arrays to work correctly with C arrays with arralloc() allocated memory when passed as arguments to an external Fortran subroutine.

Exploring the processor starvation problem, when the number of in-lines per processor is low and when N is not neatly divisible by P would be an interesting topic for future work. As would running a very large seismic volume on an exascale machine using the data parallel decomposition technique.
References

[1] Institute of Petroleum Engineering, Heriot-Watt University
http://www.pet.hw.ac.uk/index.cfm


[14] Parallel IO using MPI-IO, Lecture 3: Basic MPI-IO Calls, Dr David Henty EPCC MSc PSMA Module 2011

[27] Seymour Lipschutz, 1997 Beginning Linear Algebra, McGraw Hill

55
Appendices

Appendix 1. INTEL IMB Installation Procedure

1. Install as a root for system wide access for all users [default]
2. Install to root for system wide access for all users using sudo privileges and password
3. Install as current user to limit access to user level

h. Help
q. Quit

Please type a selection [1]: 3
Proceeding with non-root installation.

-------------------------------------------------------------------------------------------------
Initializing, please wait...
-------------------------------------------------------------------------------------------------
Step no: 1 of 6 | Welcome
-------------------------------------------------------------------------------------------------
Welcome to the Intel(R) Cluster Tools 2011 for Linux* installation program.

-------------------------------------------------------------------------------------------------
You will complete the steps below during this installation:
Step 1 : Welcome
Step 2 : License
Step 3 : Activation
Step 4 : Options
Step 5 : Installation
Step 6 : Complete
-------------------------------------------------------------------------------------------------
Press "Enter" key to continue or "q" to quit:
Checking the prerequisites. It can take several minutes. Please wait...
-------------------------------------------------------------------------------------------------
Step no: 1 of 6 | Options > Missing Optional Pre-requisite(s)
-------------------------------------------------------------------------------------------------
There is one or more optional unresolved issues. It is highly recommended to fix it all before you continue the installation. You can fix it without exiting from the installation and re-check. Or you can quit from the installation, fix it and run the installation again.

-------------------------------------------------------------------------------------------------
Missing optional pre-requisite
-- unsupported OS
-------------------------------------------------------------------------------------------------
1. Skip missing optional pre-requisites [default]
2. Show the detailed info about issue(s)
3. Re-check the pre-requisites

h. Help
b. Back to the previous menu
q. Quit

Please type a selection or press "Enter" to accept default choice [1]:1
Step no: 2 of 6 | License
-------------------------------------------------------------------------------------------------
To continue with the installation of this product you are required to accept the terms and conditions of the End User License Agreement (EULA). The EULA is displayed using the "more" utility. Press the spacebar to advance to the next page or enter "q" to skip to the end. After reading the EULA, you must enter "accept" to continue the installation or "decline" to return to the previous menu.

-------------------------------------------------------------------------------------------------
IMPORTANT - READ BEFORE COPYING, INSTALLING OR USING.

57
Do not copy, install, or use the Materials provided under this license agreement ("Agreement"), until you have carefully read the following terms and conditions.

By copying, installing, or otherwise using the Materials, you agree to be bound by the terms of this Agreement. If you do not agree to the terms of this Agreement, do not copy, install, or use the Materials.

End User License Agreement for the Intel(R) Software Development Products

1. LICENSE DEFINITIONS:

Do you agree to be bound by the terms and conditions of this license agreement? Type "accept" to continue or "decline" to back to the previous menu: accept

Step no: 3 of 6 | Activation

If you have purchased this product and have the serial number and a connection to the internet you can choose to activate the product at this time. Activation is a secure and anonymous one-time process that verifies your software licensing rights to use the product. Alternatively, you can choose to evaluate the product or defer activation by choosing the evaluate option. Evaluation software will time out in 30 days. Also you can use license file, license manager, or the system you are installing on does not have internet access activation options.

1. I want to activate my product using a serial number [default]
2. I want to evaluate my product or activate later
3. I want to activate either remotely, or by using a license file, or by using a license manager

h. Help
b. Back to the previous menu
q. Quit

Please type a selection or press "Enter" to accept default choice [1]: 1

Please type a selection or press "Enter" to accept default choice [1]:
Note: Press "Enter" key to back to the previous menu.
Please type your serial number (the format is XXXX-XXXXXXX): V4VD-FDLZ4BTJ

Activation completed successfully.

Press "Enter" key to continue:

Step no: 4 of 6 | Options

You are now ready to begin installation. You can use all default installation settings by simply choosing the "Start installation Now" option or you can customize these settings by selecting any of the change options given below first. You can view a summary of the settings by selecting "Show pre-install summary".

1. Start installation Now

2. Change install directory  [ /lustre/home/alex/intel ]
3. Change components to install  [ All ]
4. Change advanced options  [ Current node ]
5. Show pre-install summary

h. Help
b. Back to the previous menu
q. Quit

Please type a selection or press "Enter" to accept default choice [1]: 1

Step no: 5 of 6 | Installation

Each component will be installed individually. If you cancel the installation, components that have been completely installed will remain on your system. This
installation may take several minutes, depending on your system and the options you selected.

Installing Intel(R) MPI Library component... done

Installing Intel(R) Trace Analyzer and Collector component... done

Installing Intel(R) Math Kernel Library 10.3 for Linux* (for applications running on IA-32) component... done

Installing Intel(R) Math Kernel Library 10.3 for Linux* (for applications running on Intel(R) 64) component... done

Installing Intel(R) MPI Benchmarks component... done

Installing Intel(R) Cluster Toolkit Common Files component... done

Step no: 6 of 6 | Complete

Thank you for installing and for using the Intel(R) Cluster Toolkit 2011 for Linux* OS.

Support services start from the time you install or activate your product, so please create your support account now in order to take full advantage of your product purchase. Your subscription service support account provides access to free product updates interactive issue management, technical support, sample code, and documentation.

To create your support account, please visit the Subscription Services web site https://registrationcenter.intel.com/RegCenter/registerexpress.aspx?clientsn=V4V-

D-FDLZ4PTJ

q. Quit [default]

Please type a selection or press "Enter" to accept default choice [g]:
[alex@cayenne0 1_ics_2011.0.013]$ cd /lustre/home/alex/intel/imb/3.2.2/src
[alex@cayenne0 src]$ make all

gmake -f make_ict IMB-EXT
gmake[1]: Entering directory `'/lustre/home/alex/intel/imb/3.2.2/src'
sleep 1; touch exe_ext *.c; rm -rf exe_io exe_mpi

gmake: f Makefile_base EXT CPP=EXT

gmake[2]: Entering directory `'/lustre/home/alex/intel/imb/3.2.2/src'

mpiicc -DEXT -c IMB.c

mpiicc -DEXT -c IMB_declar.c

mpiicc -DEXT -c IMB_init.c

mpiicc -DEXT -c IMB_mem_manager.c

mpiicc -DEXT -c IMB_benchlist.c

IMB_benchlist.c(215): warning #167: argument of type "const char *" is incompatible with parameter of type "char *"

Bmark->name = IMB_str(bname);

^{

mpiicc -DEXT -c IMB_parse_name_ext.c

mpiicc -DEXT -c IMB_strs.c

mpiicc -DEXT -c IMB_err_handler.c

mpiicc -DEXT -c IMB_g_info.c

mpiicc -DEXT -c IMB_warm_up.c

mpiicc -DEXT -c IMB_output.c

mpiicc -DEXT -c IMB_window.c

mpiicc -DEXT -c IMB Ones_unidir.c

mpiicc -DEXT -c IMB Ones_bidir.c

mpiicc -DEXT -c IMB Ones_accu.c

mpiicc -DEXT -c IMB_init_transfer.c

mpiicc -DEXT -c IMB_user_set_info.c

mpiicc -DEXT -c IMB_chk_diff.c

mpiicc -o IMB-EXT IMB.o IMB_declare.o IMB_init.o IMB_mem_manager.o

IMB_benchlist.o IMB_parse_name_ext.o IMB_strs.o IMB_err_handler.o IMB_g_info.o

59
IMB warm_up.o IMB output.o IMB window.o IMB ones unidir.o IMB ones bidir.o IMB ones accu.o IMB init transfer.o IMB user set info.o IMB chk diff.o

gmake[2]: Leaving directory `/lustre/home/alex/intel/imb/3.2.2/src'
gmake[1]: Leaving directory `/lustre/home/alex/intel/imb/3.2.2/src'
gmake -f make upt IMB-IO

gmake[1]: Entering directory `/lustre/home/alex/intel/imb/3.2.2/src'
touch exe io *.c: rm -rf exe io exe_ext exe_mpi1

gmake -f Makefile.base IO CPP=MPIO

gmake[2]: Entering directory `/lustre/home/alex/intel/imb/3.2.2/src'

mpiicc -DMPIIO -c IMB.o
mpiicc -DMPIIO -c IMB declare.c
mpiicc -DMPIIO -c IMB_init.c
mpiicc -DMPIIO -c IMB_mem_manager.c
mpiicc -DMPIIO -c IMB_init_file.c
mpiicc -DMPIIO -c IMB_benchlist.c

IMB_benchlist.c(215): warning #167: argument of type "const char *" is incompatible with parameter of type "char *"
Bmark->name = IMB_str(bname);
^

mpiicc -DMPIIO -c IMB_parse_name_io.c
mpiicc -DMPIIO -c IMB_strgs.c
mpiicc -DMPIIO -c IMB_user set info.c
mpiicc -DMPIIO -c IMB_cpu exploit.c
mpiicc -DMPIIO -c IMB_err handler.c
mpiicc -DMPIIO -c IMB_g info.c
mpiicc -DMPIIO -c IMB warm up.c
mpiicc -DMPIIO -c IMB output.c
mpiicc -DMPIIO -c IMB open close.c
mpiicc -DMPIIO -c IMB write.c
mpiicc -DMPIIO -c IMB read.c
mpiicc -DMPIIO -c IMB init transfer.c
mpiicc -DMPIIO -c IMB chk diff.c

mpiicc -o IMB-IO IMB.o IMB declare.o IMB_init.o IMB_mem_manager.o IMB init file.o IMB_benchlist.o IMB_parse_name_io.o IMB_strgs.o IMB_user set info.o IMB_cpu exploit.o IMB_err handler.o IMB_g info.o IMB warm up.o IMB output.o IMB open close.o IMB write.o IMB read.o IMB init transfer.o IMB chk diff.o

gmake[2]: Leaving directory `/lustre/home/alex/intel/imb/3.2.2/src'
gmake[1]: Leaving directory `/lustre/home/alex/intel/imb/3.2.2/src'

This Makefile uses make upt to build IMB based on "ICT" (Intel(R) Cluster Toolkit)

Targets: MP1 (default), EXT, IO, all, clean
MP1 builds the MPI-1 standard benchmarks
EXT builds the one sided communications benchmarks
IO builds the MPI-File I/O benchmarks
all = MP1+EXT+IO

When an Intel(r) MPI Library install and mpiicc path exists, this should work immediately.

Alternatively, use

gmake -f make_mpich

to install an mpich or similar version; for this,
you normally have to edit at least the MPI_HOME variable provided in make_mpich

gmake -f make upt IMB-MP1

gmake[1]: Entering directory `/lustre/home/alex/intel/imb/3.2.2/src'
sleep 1; touch exe_mpi1 *.c; rm -rf exe io exe_ext

gmake -f Makefile.base MP1 CPP=MP1

gmake[2]: Entering directory `/lustre/home/alex/intel/imb/3.2.2/src'

mpiicc -DMPI1 -c IMB.c
mpiicc -DMPI1 -c IMB declare.c
mpiicc -DMPI1 -c IMB_init.c
mpiicc -DMPI1 -c IMB_mem_manager.c
mpiicc -DMPI1 -c IMB_parse_name_mpi1.c
mpiicc -DMPI1 -c IMB_benchlist.c
IMB_benchlist.c(215): warning #187: argument of type "const char *" is incompatible
with parameter of type "char *"
      Bname->name = IMB_str(bname);

mpiicc -DMPI1 -c IMB_strgs.c
mpiicc -DMPI1 -c IMB_err_handler.c
mpiicc -DMPI1 -c IMB_g_info.c
mpiicc -DMPI1 -c IMB_warn_up.c
mpiicc -DMPI1 -c IMB_output.c
mpiicc -DMPI1 -c IMB_pingpong.c
mpiicc -DMPI1 -c IMB_pingping.c
mpiicc -DMPI1 -c IMB_allreduce.c
mpiicc -DMPI1 -c IMB_reduce_scatter.c
mpiicc -DMPI1 -c IMB_reduce.c
mpiicc -DMPI1 -c IMB_exchange.c
mpiicc -DMPI1 -c IMB_bcast.c
mpiicc -DMPI1 -c IMB_barrier.c
mpiicc -DMPI1 -c IMB_allgather.c
mpiicc -DMPI1 -c IMB_allgatherv.c
mpiicc -DMPI1 -c IMB_gather.c
mpiicc -DMPI1 -c IMB_gatherv.c
mpiicc -DMPI1 -c IMB_scatter.c
mpiicc -DMPI1 -c IMB_scatterv.c
mpiicc -DMPI1 -c IMB_alltoall.c
mpiicc -DMPI1 -c IMB_alltoallv.c
mpiicc -DMPI1 -c IMB_sendrecv.c
mpiicc -DMPI1 -c IMB_init_transfer.c
mpiicc -DMPI1 -c IMB_chk_diff.c
mpiicc -DMPI1 -c IMB_cpu_exploit.c

mpiicc -o IMB-MPI1 IMB.o IMB_declare.o IMB_init.o IMB_mem_manager.o
IMB_parse_name_mpi.o IMB_benchlist.o IMB_strgs.o IMB_err_handler.o IMB_g_info.o
IMB_warn_up.o IMB_output.o IMB_pingpong.o IMB_pingping.o IMB_allreduce.o
IMB_reduce_scatter.o IMB_allreduce.o IMB_exchange.o IMB_bcast.o IMB_barrier.o
IMB_allgather.o IMB_allgatherv.o IMB_gather.o IMB_gatherv.o IMB_scatter.o
IMB_scatterv.o IMB_alltoall.o IMB_alltoallv.o IMB_sendrecv.o IMB_init_transfer.o
IMB_chk_diff.o IMB_cpu_exploit.o
gmake[2]: Leaving directory `/lustre/home/alex/intel/imb/3.2.2/src'
gmake[1]: Leaving directory `/lustre/home/alex/intel/imb/3.2.2/src'
[alex@cayenneo src]$ mpirun -n 6 IMB-IO <----- RUN INTEL BENCHMARK HERE.
[alex@cayenneo src]$ mpirun IMB-EXT > results.txt <----- MORE BENCHMARKING

This generates: See Appendix 3 for details.

#-----------------------------------------------------------
# Intel (R) MPI Benchmark Suite V3.2.2, MPI-2 part
#-----------------------------------------------------------
# Date : Sat Aug 27 15:56:55 2011
# Machine : x86_64
# System : Linux
# Release : 2.6.18-92.1.18.el5.sunhpc1
# Version : #1 SMP Mon Dec 1 15:25:03 EST 2008
# MPI Version : 2.1
# MPI Thread Environment: MPI_THREAD_SINGLE

# New default behavior from Version 3.2 on:
# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# "SECS_PER_SAMPLE" (= IMB_settings.h)
# or through the flag => -time

# Calling sequence was:
# IMB-EXT
Appendix 2. Example Benchmarking Data for Cayenne
INTEL IMB-IO

# Intel (R) MPI Benchmark Suite V3.2.2, MPI-IO part
# Date: Thu Aug 11 12:26:39 2011
# Machine: x86_64
# System: Linux
# Release: 2.6.18-92.1.18.el5.sunhpc1
# Version: #1 SMP Mon Dec 1 15:25:03 EST 2008
# MPI Version: 2.1
# MPI Thread Environment: MPI_THREAD_SINGLE

# New default behavior from Version 3.2 on:
# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# "SECS_PER_SAMPLE" (=> IMB_settings.h)
# or through the flag => -time

# Calling sequence was:
# IMB-IO

# Minimum io portion in bytes: 0
# Maximum io portion in bytes: 16777216
#
# List of Benchmarks to run:
# S_Write_Indv
# S_IWrite_Indv
# S_Write_Expl
# S_IWrite_Expl
# P_Write_Indv
# P_IWrite_Indv
# P_Write_Shared
# P_IWrite_Shared
# P_Write_Priv
# P_IWrite_Priv
# P_Write_Expl
# P_IWrite_Expl
# C_Write_Indv
# C_IWrite_Indv
# C_Write_Shared
# C_IWrite_Shared
# C_Write_Expl
# C_IWrite_Expl
# S_Read_Indv
# S_IRead_Indv
# S_Read_Expl
# S_IRead_Expl
# P_Read_Indv
# P_IRead_Indv
# P_Read_Shared
# P_IRead_Shared
# P_Read_Priv
# P_IRead_Priv
# P_Read_Expl
# P_IRead_Expl
# C_Read_Indv
# C_IRead_Indv
# C_Read_Shared
# C_IRead_Shared
# C_Read_Expl
# C_IRead_Expl
# Open_Close

# For nonblocking benchmarks:

# Function CPU_Exploit obtains an undisturbed
# performance of 3779.96 MFlops
#-------------------------------------------------------------
# Intel (R) MPI Benchmark Suite V3.2.2, MPI-IO part
#-------------------------------------------------------------
# Date       : Thu Aug 11 12:26:39 2011
# Machine    : x86_64
# System     : Linux
# Release    : 2.6.18-92.1.18.el5.sunhpc1
# Version    : #1 SMP Mon Dec 1 15:25:03 EST 2008
# MPI Version: 2.1
# MPI Thread Environment: MPI_THREAD_SINGLE

# New default behavior from Version 3.2 on:

# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# "SECS_PER_SAMPLE" (=> IMB_settings.h)
# or through the flag => -time

# Calling sequence was:

# IMB-IO

# Minimum io portion in bytes:  0
# Maximum io portion in bytes:  16777216
#
#
#

# List of Benchmarks to run:

# S_Write_Indv
# S_IWrite_Indv
# S_Write_Expl
# S_IWrite_Expl
# P_Write_Indv
# P_IWrite_Indv
# P_Write_Shared
# P_IWrite_Shared
# P_Write_Priv
# P_IWrite_Priv
# P_Write_Expl
# P_IWrite_Expl
# C_Write_Indv
# C_IWrite_Indv
# C_Write_Shared
# C_IWrite_Shared
# C_Write_Expl
# C_IWrite_Expl
# S_Read_Indv
# S_IRead_Indv
# S_Read_Expl
# S_IRead_Expl
# P_Read_Indv
# P_IRead_Indv
# P_Read_Shared
# P_IRead_Shared
# P_Read_Priv
# P_IRead_Priv
# P_Read_Expl
# P_IRead_Expl
# C_Read_Indv
# C_IRead_Indv
# C_Read_Shared
# C_IRead_Shared
# C_Read_Expl
# C_IRead_Expl
# Open_Close

# For nonblocking benchmarks:

# Function CPU_Exploit obtains an undisturbed
# performance of 3768.01 MFlops

."
Appendix 3. Example Benchmarking Data for Cayenne
INTEL IMB-EXT

#----------------------------------------------------------
# Intel (R) MPI Benchmark Suite V3.2.2, MPI-2 part
#----------------------------------------------------------
# Date : Thu Aug 11 12:27:49 2011
# Machine : x86_64
# System : Linux
# Release : 2.6.18-92.1.18.el5.sunhpc1
# Version : #1 SMP Mon Dec 1 15:25:03 EST 2008
# MPI Version : 2.1
# MPI Thread Environment: MPI_THREAD_SINGLE

# New default behavior from Version 3.2 on:
# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# "SECS_PER_SAMPLE" (=> IMB_settings.h)
# or through the flag => -time

# Calling sequence was:
# IMB-EXT

# Minimum message length in bytes: 0
# Maximum message length in bytes: 4194304
# MPI_Datatype : MPI_BYTE
# MPI_Datatype for reductions : MPI_FLOAT
# MPI_Op : MPI_SUM
#
# List of Benchmarks to run:
# Window
# Unidir_Get
# Unidir_Put
# Bidir_Get
# Bidir_Put
# Accumulate

#----------------------------------------------------------
# Benchmarking Window
# #processes = 1
#----------------------------------------------------------
# bytes #repetitions t_min[usec] t_max[usec] t_avg[usec]
<table>
<thead>
<tr>
<th># bytes</th>
<th># repetitions</th>
<th>t_min[usec]</th>
<th>t_max[usec]</th>
<th>t_avg[usec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>0.08</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>0.08</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>0.08</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
</tr>
<tr>
<td>1024</td>
<td>1000</td>
<td>0.19</td>
<td>0.19</td>
<td>0.19</td>
</tr>
<tr>
<td>2048</td>
<td>1000</td>
<td>1.49</td>
<td>1.49</td>
<td>1.49</td>
</tr>
<tr>
<td>4096</td>
<td>1000</td>
<td>1.77</td>
<td>1.77</td>
<td>1.77</td>
</tr>
<tr>
<td>8192</td>
<td>1000</td>
<td>1.79</td>
<td>1.79</td>
<td>1.79</td>
</tr>
<tr>
<td>16384</td>
<td>1000</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
</tr>
<tr>
<td>32768</td>
<td>1000</td>
<td>1.80</td>
<td>1.80</td>
<td>1.80</td>
</tr>
<tr>
<td>65536</td>
<td>1000</td>
<td>1.81</td>
<td>1.81</td>
<td>1.81</td>
</tr>
<tr>
<td>131072</td>
<td>1000</td>
<td>1.82</td>
<td>1.82</td>
<td>1.82</td>
</tr>
<tr>
<td>262144</td>
<td>1000</td>
<td>1.83</td>
<td>1.83</td>
<td>1.83</td>
</tr>
<tr>
<td>524288</td>
<td>80</td>
<td>1.84</td>
<td>1.84</td>
<td>1.84</td>
</tr>
<tr>
<td>1048576</td>
<td>40</td>
<td>1.85</td>
<td>1.85</td>
<td>1.85</td>
</tr>
<tr>
<td>2097152</td>
<td>20</td>
<td>1.86</td>
<td>1.86</td>
<td>1.86</td>
</tr>
<tr>
<td>4194304</td>
<td>10</td>
<td>1.87</td>
<td>1.87</td>
<td>1.87</td>
</tr>
</tbody>
</table>

# Benchmarking Accumulate
# #processes = 1
#
# MODE: AGGREGATE
#
# bytes repetitions t_min[usec] t_max[usec] t_avg[usec]
0    1000      0.06     0.06     0.06
4    1000      0.08     0.08     0.08
8    1000      0.08     0.08     0.08
16   1000      0.09     0.09     0.09
32   1000      0.09     0.09     0.09
64   1000      0.10     0.10     0.10
128  1000      0.12     0.12     0.12
256  1000      0.14     0.14     0.14
512  1000      0.19     0.19     0.19
1024 1000      0.19     0.19     0.19
2048 1000      0.19     0.19     0.19
4096 1000      0.19     0.19     0.19

# Intel (R) MPI Benchmark Suite V3.2.2, MPI-2 part
#--------------------------------------------------
# Date : Thu Aug 11 12:27:49 2011
# Machine : x86_64
# System : Linux
# Release : 2.6.18-92.1.18.el5.sunhpc1
# Version : #1 SMP Mon Dec 1 15:25:03 EST 2008
# MPI Version : 2.1
# MPI Thread Environment: MPI_THREAD_SINGLE

66
# New default behavior from Version 3.2 on:
# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# "SECS_PER_SAMPLE" (=> IMB_settings.h)
# or through the flag => -time

# Calling sequence was:
# IMB-EXT

# Minimum message length in bytes:  0
# Maximum message length in bytes:  4194304
# MPI_Datatype : MPI_BYTE
# MPI_Datatype for reductions : MPI_FLOAT
# MPI_Op : MPI_SUM
#
#
# List of Benchmarks to run:
# Window
# Unidir_Get
# Unidir_Put
# Bidir_Get
# Bidir_Put
# Accumulate

#Benchmarking Window
# processes = 1

<table>
<thead>
<tr>
<th>#bytes</th>
<th>#repetitions</th>
<th>t_min[usec]</th>
<th>t_max[usec]</th>
<th>t_avg[usec]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1.45</td>
<td>1.45</td>
<td>1.45</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>1.77</td>
<td>1.77</td>
<td>1.77</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>1.77</td>
<td>1.77</td>
<td>1.77</td>
</tr>
<tr>
<td>16</td>
<td>100</td>
<td>1.83</td>
<td>1.83</td>
<td>1.83</td>
</tr>
<tr>
<td>32</td>
<td>100</td>
<td>1.81</td>
<td>1.81</td>
<td>1.81</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td>1.77</td>
<td>1.77</td>
<td>1.77</td>
</tr>
<tr>
<td>128</td>
<td>100</td>
<td>1.84</td>
<td>1.84</td>
<td>1.84</td>
</tr>
<tr>
<td>256</td>
<td>100</td>
<td>1.76</td>
<td>1.76</td>
<td>1.76</td>
</tr>
<tr>
<td>512</td>
<td>100</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
</tr>
<tr>
<td>1024</td>
<td>100</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
</tr>
<tr>
<td>2048</td>
<td>100</td>
<td>1.81</td>
<td>1.81</td>
<td>1.81</td>
</tr>
<tr>
<td>4096</td>
<td>100</td>
<td>1.94</td>
<td>1.94</td>
<td>1.94</td>
</tr>
<tr>
<td>8192</td>
<td>100</td>
<td>1.83</td>
<td>1.83</td>
<td>1.83</td>
</tr>
<tr>
<td>16384</td>
<td>100</td>
<td>1.83</td>
<td>1.83</td>
<td>1.83</td>
</tr>
<tr>
<td>32768</td>
<td>100</td>
<td>1.85</td>
<td>1.85</td>
<td>1.85</td>
</tr>
<tr>
<td>65536</td>
<td>100</td>
<td>1.86</td>
<td>1.86</td>
<td>1.86</td>
</tr>
<tr>
<td>131072</td>
<td>100</td>
<td>1.91</td>
<td>1.91</td>
<td>1.91</td>
</tr>
<tr>
<td>262144</td>
<td>100</td>
<td>1.79</td>
<td>1.79</td>
<td>1.79</td>
</tr>
</tbody>
</table>

67
524288  80  1.79  1.79  1.79
1048576  40  1.72  1.72  1.72
2097152  20  1.75  1.75  1.75
4194304  10  1.81  1.81  1.81

# Benchmarking Accumulate
# processes = 1
#
# MODE: AGGREGATE
#
# bytes repetitions t_min[usec] t_max[usec] t_avg[usec]
# 0     1000    0.06    0.06   0.06
# 4     1000    0.08    0.08   0.08
# 8     1000    0.09    0.09   0.09
# 16    1000    0.09    0.09   0.09
# 32    1000    0.09    0.09   0.09
# 64    1000    0.09    0.09   0.09
# 128   1000    0.09    0.09   0.09
# 256   1000    0.10    0.10   0.10
# 512   1000    0.12    0.12   0.12
# 1024  1000    0.14    0.14   0.14
# 2048  1000    0.19    0.19   0.19
# 4096  1000

# New default behavior from Version 3.2 on:
# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# "SECS_PER_SAMPLE" (= IMB_settings.h)
# or through the flag => -time

# Calling sequence was:
#
# IMB-EXT
#
# Minimum message length in bytes:  0
# Maximum message length in bytes:  4194304
#  
# MPI_Datatype            : MPI_BYTE

68
# MPI_Datatype for reductions : MPI_FLOAT
# MPI_Op : MPI_SUM
#

# List of Benchmarks to run:

# Window
# Unidir_Get
# Unidir_Put
# Bidir_Get
# Bidir_Put
# Accumulate

#---------------------------------------------------------------
# Benchmarking Window
# # processes = 1
#---------------------------------------------------------------

<table>
<thead>
<tr>
<th>#bytes</th>
<th>#repetitions</th>
<th>t_min [usec]</th>
<th>t_max [usec]</th>
<th>t_avg [usec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td>1.51</td>
<td>1.51</td>
<td>1.51</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>1.80</td>
<td>1.80</td>
<td>1.80</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>1.80</td>
<td>1.80</td>
<td>1.80</td>
</tr>
<tr>
<td>16</td>
<td>100</td>
<td>1.79</td>
<td>1.79</td>
<td>1.79</td>
</tr>
<tr>
<td>32</td>
<td>100</td>
<td>1.82</td>
<td>1.82</td>
<td>1.82</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td>1.80</td>
<td>1.80</td>
<td>1.80</td>
</tr>
<tr>
<td>128</td>
<td>100</td>
<td>1.79</td>
<td>1.79</td>
<td>1.79</td>
</tr>
<tr>
<td>256</td>
<td>100</td>
<td>1.85</td>
<td>1.85</td>
<td>1.85</td>
</tr>
<tr>
<td>512</td>
<td>100</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
</tr>
<tr>
<td>1024</td>
<td>100</td>
<td>1.80</td>
<td>1.80</td>
<td>1.80</td>
</tr>
<tr>
<td>2048</td>
<td>100</td>
<td>1.85</td>
<td>1.85</td>
<td>1.85</td>
</tr>
<tr>
<td>4096</td>
<td>100</td>
<td>1.81</td>
<td>1.81</td>
<td>1.81</td>
</tr>
<tr>
<td>8192</td>
<td>100</td>
<td>1.84</td>
<td>1.84</td>
<td>1.84</td>
</tr>
<tr>
<td>16384</td>
<td>100</td>
<td>1.85</td>
<td>1.85</td>
<td>1.85</td>
</tr>
<tr>
<td>32768</td>
<td>100</td>
<td>1.88</td>
<td>1.88</td>
<td>1.88</td>
</tr>
<tr>
<td>65536</td>
<td>100</td>
<td>1.87</td>
<td>1.87</td>
<td>1.87</td>
</tr>
<tr>
<td>131072</td>
<td>100</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>262144</td>
<td>100</td>
<td>1.80</td>
<td>1.80</td>
<td>1.80</td>
</tr>
<tr>
<td>524288</td>
<td>80</td>
<td>1.87</td>
<td>1.87</td>
<td>1.87</td>
</tr>
<tr>
<td>1048576</td>
<td>40</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
</tr>
<tr>
<td>2097152</td>
<td>20</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
</tr>
<tr>
<td>4194304</td>
<td>10</td>
<td>1.79</td>
<td>1.79</td>
<td>1.79</td>
</tr>
</tbody>
</table>
Appendix 4. README: How to Run and Compile SVD Code

########################################################################
# RUNNING CODE #
########################################################################
On Cayenne and Ness Back-end
./etlp_svd_64bit.sh

On Cayenne and Ness Fromt-end
./etlp_svd_64bit 1 3 3d_l10_baseline.segy 3d_l10_monitor.segy

Running other volumes.
First go to etlp_segycode.c and edit the file.
Find the 3 lines:
/* NELSON and large synthetic data set Settings */
int MAXSAMPS = 1351; /* Number of samples per traces */
int MAXTRACE = 1048; /* Number of traces in a single in-line (or
crossline) */
int MAXLINES = 10; /* Number of lines in 3D data volume */

Edit them to the correct dimentions of your data set including number of
in-lines.
Recompile the code using the correct make file and the correct settings
see next section compiling code.

########################################################################
# COMPILING CODE #
########################################################################
Choosing Correct make file
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Cayenne Compilation Options
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If the compilation is to be done on Cayenne use the Makefile file and just run
make all
or just run
make

Ness Compilation Options
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If the compilation is to be done on Ness then use the Makefile.ness and run
make -f Makefile.ness

Conditional Compilation Options
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1. MPI/serial Choices

# MPI / SERIAL RUN OPTIONS (1)
# If you want to run this code in parallel used the $(CFLAGMP)
# Otherwise either set it to blank or remove it.
# The easy option is to comment out the default flag and replace it with
#CFLAGSMP = -DMPI_Parallel

# Set with OBJs with MPIOBJ in it for parallel runs and i
# without for serial runs
#OBJs = $(ETLP_OBJs) $(MPIOBJs) # PARALLEL
OBJs = $(ETLP_OBJs) # SERIAL

####### COMPILER FLAGS (2)
# C Compiler :: Set MPICC for parallel runs and gcc for serial runs.
#CC = $(MPICC) # PARALLEL
CC = gcc # SERIAL

# Fortran Compiler
#FF = $(MPIFF) # PARALLEL
FF = gfortran # SERIAL

#########################################################################

####
# FINAL Compiler and linker Flags (3)
# Change for production or debug and parallel or serial settings as required
#########################################################################

###
# C Compiler flags
#CFLAGS = $(C_DEBUG_FLAGS) $(CFLAGSMP) $(USE_STATIC_LIBS) # Debug
CFLAGS = $(C_RUN_FLAGS) $(CFLAGSMP) $(USE_STATIC_LIBS) # Production

# Fortran compiler flags
#FFLAGS = $(F_DEBUG_FLAGS) $(USE_STATIC_LIBS) # Debug
FFLAGS = $(F_RUN_FLAGS) $(USE_STATIC_LIBS) # Production

#########################################################################

# Comment one of these LINKER in for a parallel run #

#########################################################################

# Comment one of these in for a serial run
#LFLAGS = $(L_DEBUG_FLAGS) $(MPIF90_LINK) # Debug PARALLEL
LFLAGS = $(L_RUN_FLAGS) $(MPIF90_LINK) # Production PARALLEL

# Comment one of these in for a serial run
#LFLAGS = $(L_DEBUG_FLAGS) # Debug SERIAL
LFLAGS = $(L_RUN_FLAGS) # Production SERIAL

#########################################################################
# Synthetic Test Data Creation #
#########################################################################

Run build3DdataSet

[alex@cayenne0 src]$ ./build3DdataSet
Enter 2D survey SEGY file name:
baseline.segy        <-- 2D Input file
Enter number of traces per 2D line:
1048        <-- You must specify traces per in-line,
Enter number of 2D lines to insert into new 3d volume:
10        <-- Specify how many in-lines you want to create.

******************************************************************************
THE OUTPUT SHOULD LIKE THIS IF THE RUN HAS WORKED
created 3D output file 3d_i10_baseline.segy
Write contents of the binary header to see if it makes sense
JobID = 0, Line No. = 0, Reel No. = 0
NSO (Near Shot Order) = 1, nSamples = 1351
Traces read (2D line 1) = 1048, Total traces 1048
Traces read (2D line 2) = 1048, Total traces 2096
Traces read (2D line 3) = 1048, Total traces 3144
Traces read (2D line 4) = 1048, Total traces 4192
Traces read (2D line 5) = 1048, Total traces 5240
Traces read (2D line 6) = 1048, Total traces 6288
Traces read (2D line 7) = 1048, Total traces 7336
Traces read (2D line 8) = 1048, Total traces 8384
Traces read (2D line 9) = 1048, Total traces 9432
Traces read (2D line 10) = 1048, Total traces 10480

SUCCESSFUL EXIT FROM PROGRAM
[alex@cayenne0 src]$
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpi_utils.h</td>
<td>MPI Function Header (New code)</td>
</tr>
<tr>
<td>old_code</td>
<td>Contains original SVD code from ETLP</td>
</tr>
</tbody>
</table>