Science & Technology Facilities Council

Technical Report
DL-TR-2015-001

O{ y |m~m¡u£q pqn¢ssuzs ¢ uzs `{¡mxbuq¤ o~u|¡uzs

S O{~nq¡¡ Y M t¤{~¡t

January 2015

Enquiries concerning this report should be addressed to:

Chadwick Library
STFC Daresbury Laboratory
Sci-Tech Daresbury
Keckwick Lane
Warrington
WA4 4AD

Tel: +44(0)1925 603397
Fax: +44(0)1925 603779
email: librarydl@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1362-0207

Comparative Debugging Using TotalView Scripting

Greg Corbett[1] and Mike Ashworth[2]

STFC Scientific Computing Department

[1]STFC Rutherford Appleton Laboratory,
Harwell Oxford,
Didcot OX11 0QX
United Kingdom

[2]STFC Daresbury Laboratory,
Sci-Tech Daresbury,
Warrington WA4 4AD
United Kingdom

Abstract

Comparative debugging makes use of direct comparisons between two different runs of the same or similar programs to diagnose unexpected and often erroneous behaviour in the application or applications. Comparative debugging is an important technique for STFC and for computational scientists at other labs around the world. As a six month project at STFC, in collaboration with Rogue Wave Software Inc., an automated comparative debugging script (CDTV) has been created that makes use of the command interface of RogueWave's TotalView debugger. To demonstrate this script, example scenarios are shown and discussed within the report, highlighting the key features, current limitations and possible improvements that could guide any future work. The CDTV script has been shown to be capable of locating differences caused by an artificial bug introduces into a serial program written in C or in Fortran as well as parallel programs using MPI and OpenMP. As well as simple test codes, the script has been demonstrated using a real application, the UK Turbulence Consortium's Shock/Boundary Layer Interaction code. We believe that similar large scale applications could benefit from comparative debugging without making major changes to the underlying CDTV script.

1 Introduction

This report describes the design and implementation of a prototype automated comparative debugging TotalView script (CDTV) which focuses on array comparisons. First, the project background and the programs used, such as the UK Turbulence Consortium's Shock/Boundary Layer Interaction code (SBLI), are discussed, followed by the design process. The design section starts with how the original comparative_debug_script .cli (CDS) was extended to function with any relatively simple program before moving onto how the speed of the script can be improved with the use of options and how support for parallel programs was incorporated.

The demonstration section shows multiple debugging scenarios to showcase the capability and versatility of the CDTV script. These demonstrations take the form of run scripts, with names of the form run*.cli, which can be run with the command *totalviewcli –s <run script name>*. These run scripts convey to the CDTV script information that includes:

- What logical arrays can be recomposed (the *interestingArray* variable)
- The arrays logical dimensions (the *dimensionVariables* variable)
- How the data is decomposed (the *transformIndices* function) if applicable.

The final CDTV script is able to locate differences in the simple test beds, such as the serial to serial case using the simple_finite_difference_code programs (*runSFDFinal.cli*), as well as the serial to parallel case (*runSBLIFinal4.cli*), parallel to parallel case with the same domain distribution and the parallel to parallel case with different domain decompositions (*runSBLIFinal8.cli*).

The report finishes with possibilities for further work and conclusions gained from the project.

1.1 Project background

Comparative debugging, also sometimes called relative debugging, is the practice of using direct comparisons between two different runs of the same (or similar) codes to diagnose unexpected and erroneous behaviour in the application (or applications). Comparative debugging is typically used between two runs of the same code – working on the same input data but with the work distributed across different numbers of processes or threads. It can also be used between two versions of the same code (e.g. yesterday's version and today's version) with the same data and at the same scale. Comparative debugging is especially useful for detecting small deviations from expected behaviour, such as those introduced by bugs, in dynamical simulations which should, if they worked correctly, follow deterministically from input and boundary conditions.

Comparative debugging is an important technique for STFC and for computational scientists at other labs around the world. Successful automation and simplification of the process of comparative debugging has the potential greatly to improve the scientific productivity of computational scientists, the stability of the software that they develop and the correctness of the computational results generated. Such improvements will have a consequent benefit for the scientific communities which rely on simulation software.

STFC has entered into a collaborative partnership with RogueWave Software Inc.to look at a number of R&D topics around the TotalView debugger. TotalView provides users with a very powerful command line interface which is based on the TCL language. This can be used interactively for those users who like to work with command line (rather than graphical) debugger interfaces. In addition TCL is a mature and well documented scripting language, similar to Perl or Python, upon which very sophisticated extensions can be built. TotalView users have built new custom graphical interfaces, plugged TotalView into Interactive Development Environments, implemented testing frameworks, automated repetitive activities, and described complex data type transformations.

This report describes the results from a six-month project to use TCL scripts to implement the comparison of arrays between two processes which might have, for example, different numbers of threads or processes.

1.2 Scope of the project

This project served as a proof of concept for an automated comparative debugging TotalView (CDTV) script. The CDTV script was designed and implemented with the intention of being a general purpose debugging tool. However, testing of the current capability of the script has been limited to three application test beds for the purposes of this project. These applications are described in detail in Section 2.

1.3 Brief description of TotalView

TotalView is a GUI-based source code defect analysis tool that gives the programmer control over processes and thread execution and visibility into program state and variables.

It allows the programmer to debug one or many processes and/or threads in a single window with complete control over program execution. This allows the programmer to set breakpoints, stepping line by line through the code on a single thread, or with coordinated groups of processes or threads, and run or halt arbitrary sets of processes or threads. The programmer can reproduce and troubleshoot difficult problems that can occur in concurrent programs that take advantage of threads, OpenMP, MPI, GPUs or other co-processors (Rogue Wave Software, 2013).

TotalView's ReplayEngine records the execution history of your program and makes that history available for diagnosis. This approach—working back from a failure, error, or crash to its root cause—eliminates the need to restart your program repeatedly with different breakpoint locations.

1.4 Brief description of applications

The project used three applications as test beds for the CDTV script.

1.4.1 simple_finite_diff_code

The simple_finite_diff_code (SFD) pair of programs consists of buggy and non-buggy versions, written in C. Both feature a 2D data array that is "evolved" forwards over a series of timestamps. The mathematics is intentionally straightforward. The value in each grid cell is the average of the previous timestamps' values for the 3x3 stencil centred on that grid cell. Boundary conditions are fixed at 0 except for the origin grid cell which simply increases over time.

This code was written by Chris Gottbrath of Rogue Wave Software.

1.4.2 recomp

The recomp family of code is written in C++ and uses OpenMP to test the parallel element of the CDTV debugger. The serial, and non-buggy, version of recomp[1] populates an array of size *n* with the numbers *[0…n-1]* in ascending order. Then for each element, the index is added to the value at that element, effectively doubling the value. The parallel versions attempt to do the same, although some have errors.

This code was written by Greg Corbett for the purpose of this project.

[1] recomp_test_serial.cpp

1.4.3 SBLI

SBLI (Yao et al, 2000) is a fully parallel simulation code to solve problems associated with Shock/Boundary Layer Interaction. There is an ever-increasing need to understand turbulence and, more importantly, to be able to model turbulent flows with improved predictive capabilities. As computing technology continues to improve, it is becoming more feasible to solve the governing equations of fluid flow, the Navier-Stokes equations, from first principles. The direct solution of the equations of motion for a fluid, however, remains a formidable task and simulations are only possible for flows with small to modest Reynolds numbers. SBLI is a sophisticated Direct Numerical Simulation (DNS), written using standard Fortran 90 code together with MPI in order to be efficient, scalable and portable across a wide range of high-performance platforms.

In the new version, a “bug” has been introduced in one of the functions in the cent2.f file, which affects an array if and only if the processes MPI rank is equal to 1.

SBLI was developed by the UK Turbulence Consortium (UKTC[2]).

1.5 Technical details of the iDataplex system

The Hartree Centre’s IBM iDataPlex, known as "Blue Wonder", comprises 512 nodes each with two 8-core 2.6 GHz Intel SandyBridge processors making 8,192 cores in total. Of these 224 nodes have 32 GB of memory. An additional 24 nodes with the same specification plus two nVidia M2090 GPUs will be available. Four high memory nodes have 256 GB memory each. 256 nodes with 128 GB memory are intended for data intensive computing. The system has a common interconnect across all nodes with high-speed Infiniband. Blue Wonder started operation at Daresbury in 2012 as the 114th most powerful computer in the world (Top500[3] list number 39).

A rich range of software is installed on the Blue Wonder system. The software used for this work comprised:

- Intel C compiler icc version 12.1.0
- Intel Fortran compiler ifort version 12.1.0
- Intel MPI 4.0.3 (part of Intel Cluster Studio 2012)
- TotalView version 8.12.0

2 Applications

This section gives a detailed description of the three test beds originally introduced in Section 1.4.

2.1 finite_diff_code

This simple finite difference (SFD) test code is a kind of "hyper-simplified" sketch of a numerical simulation which iterates time forwards on a 10x10 array. The initial condition is with every element set to 0. The boundary region is fixed to 0 except for a single cell that increases by 1 each timestamp. On each timestamp the value in each cell is set to the average of the previous steps values in a 3x3 template centred on the cell.

In the buggy version, a variable called “bug” is set to 1. This assignment results in an if-statement on line 67 evaluating to true rather than false. This evaluation means the value in the cell main_data[7][7] is set to 1 at timestamp 5, rather than the average of a 3x3 grid centred at [7][7].

[2] http://www.turbulence.ac.uk/

[3] http://www.top500.org/

2.2 recomp code

Parallel versions of recomp use OpenMP to parallelise the serial version. OpenMP is, however, used in the same style as MPI. Each tread has a private array called privateData, which it populates from the shared memory array. This design was chosen as it was believed that OpenMP would be easy to quickly build a prototype with, but that this prototype would need to be extended to MPI with little change.

Some parallel versions have the, unintentional, bug that incorrect values are added, due to the data decomposition reducing the size of the array. The bug occurs when a thread is adding the index to the element stored at the index. Because the data is now stored in multiple, smaller, arrays the indexes (in general) are no longer equal to the value stored at that index. For example, in the serial case the number 40 was stored at index 40, but in the 16 thread parallel case the number 40 was stored in thread 1.4 at index 15, as such the number 55 would be written back, rather than the intended 80.

Table 1 shows the parallel recomp versions, indicating the important features.

Version	chunckSize (i.e. how big is each threads private array)	Number of threads	Bug present?
recomp_test_par	10	40	Yes
recomp_test_par_cs10	10	40	No
recomp_test_par_cs10_2	10	40	Yes*
recomp_test_par_25	25	16	Yes

* The bug in this version is different, it is contrived to change the value of index 4 in the 8th thread

Table 1. Parallel recomp versions with chunk size and bugs

2.3 SBLI

The SBLI code was originally developed for the Cray T3E and is a sophisticated DNS code that incorporates a number of advanced features: namely high-order central differencing; a shock-preserving advection scheme from the total variation diminishing (TVD) family; entropy splitting of the Euler terms and the stable boundary scheme. The code has been written using standard Fortran 90 code together with MPI in order to be efficient, scalable and portable across a wide range of high-performance platforms. The test case used in this work is a simple turbulent channel flow benchmark (Ashworth et al, 2001) with a mesh of 30 cubed.

The most important communications structure within SBLI is a halo exchange between adjacent computational sub-domains. Providing the problem size is large enough to give a small surface area to volume ratio for each sub-domain, the communications costs are small relative to computation and should not constitute a bottleneck. The code has thus been shown to scale with sufficiently large problems to over 100,000 processor cores (Sunderland et al, 2010)

3 CDTV

3.1 Design

The CDTV script was designed and implemented with the intention of being a general purpose debugging tool. The CDTV originated from the comparative _debug_script (Section 3.1.1) and was then generalised (Section 3.1.2) to work on different programs with a variety of comparison options

[4] Using the OpenMP thread numbering convention

(Sections 3.1.3 - 3.1.6). Later, parallel support was included (Section 3.1.7 and 3.1.8) which gave insight into how breakpoint placement would affect the execution of CDTV (Section 3.1.10).

3.1.1 comparative_debug_script .cli

CDTV was designed to extend and build upon the comparative_debug_script .cli (CDS) from Chris Gottbrath (Gottbrath, 2013). Many functions from this original script were used as a basis for the initial CDTV script. To run the comparison, we run the following commands as shown in Figure 1 in the cli.

```
Setup
for { set i 0 } { $i < 8 } { incr i } { aa }
#at timestamp 8
Myfindearliestdiff
```

Figure 1. The commands to run the original CDS script

The original CDS demo involves starting from a point downstream of a difference and back tracking to it by comparing the data elements in the array between the reference code and the target code, and moving backwards in time using the ReplayEngine.

```
Output
5  5
5 : 7 7 : 2.0
```

Figure 2. The output of the commands in Figure 1.

The last line of the output in Figure 2 indicates that at a time stamp of 5, at cell [7][7], there is a difference of 2.0. The value of the difference is such because the difference is presented in terms of the average of the two threads. The buggy thread has a value of 1 and the working code has a value of 0, giving an average of a half and 1/0.5 gives a relative difference of 2.0.

The original variables of target and reference code were renamed to broken and working code respectively to aid in development, although these could easily be changed back if so desired. CDTV uses similar functions as CDS for the advance and rewind of execution, ensuring that the timestamps of each thread[5] remain in sync. The functions to determine the value of scalar variables and array elements in CDTV also build upon their counterparts in the original debug script. Functionality was introduced to support both the C and Fortran languages, focusing on handling the differences in array syntax between the two languages.

At the heart of the comparison methods in CDTV is the original code for element comparison used to determine a difference. However CDTV no longer has the concept of a debug point as the original script did. In CDS, the debug point was used to determine the local maximum of the differences at the current timestamp. This capability was not carried over to CDTV, preferring to initially report on only the first difference. It is believed that the ability to determine the local maximum of differences could also be re-included if so desired. CDTV also removed the alias to commands like the advance command to encourage the creation of an automated work flow, rather than one the user controlled.

[5] In this context, each version of the code has one process with one worker thread, as seen by TotalView.

The basic work flow designed at this early stage was as in Figure 3. At this high level, the work flow has not changed from initial design to final script.

```
Load programs
Set breakpoints
Run program to first breakpoint
While processes still running
      Check for difference
      Exit if found
      Advance to next breakpoint
Exit
```

Figure 3. The high level flow of CDTV

The exact output from comparing the SFD programs, using the CDTV script, is shown later in Section 3.2.

3.1.2 Generalising the script

To extend the capability of the original debug script and to generalise its potential application, hard coded values, like the size of arrays, were removed and replaced with user defined variables and methods to compare arrays of different dimensions, as well as scalar variables, were added. The option of specifying multiple breakpoints rather than just one was also implemented using the TCL list structure.

To facilitate the generalization, an outer "run" script was created. This run script consists of a series of hooks into the CDTV script that set key variables. This format was chosen to isolate the domain specific code that a user must change, like the *interesting*[6] array name, from the general debugger code that a user shouldn't change, like the *run* method. Such run scripts have names of the form *run*.cli* for ease of identification.

As a result of these changes, the early CDTV was able to compare various programs of similar style and complexity to the original example provided by Rogue Wave. Two examples of this ability are:

- *gregBroke* and *gregWork* (similar code to SFD, but with a 1D array and written in C)
- *gregFortBroke* and *gregFortWork* (similar code to SFD, but with a "0D" error, written in Fortran)

The results of comparing these programs can be found by running *runGreg.cli* and *runGregFort.cli* respectively, but are not discussed in further this report.

3.1.3 Leap Values

After the initial extended capability had been designed and implemented. Thoughts turned to how the time taken for detections of errors could be improved. A common scenario is where a developer will not know the precise location of the bug and may wish to check several arrays or variables quickly to determine which is causing the difference, before doing an in depth search to determine precisely where the difference is first caused. To support this scenario, a leap and a hop value mechanism were designed and added to the code.

The leap value gives the ability to skip (or leap over) iterations and breakpoints, meaning that checking does not have to occur at every iteration or breakpoint. A leap value of ten will mean that

[6] the interesting array is the one where the developer believes that an error is likely to occur

the CDTV script will advance past ten breakpoints before rechecking for differences. This technique offers a potential speed up at the cost of accuracy by reducing the total number of checks, as the check mechanism can be quite expensive on large arrays.

When combined with TotalView's Replay Engine and backtracking capability derived from the original debug script, the loss of accuracy can be mitigated by allowing the script to skip several iterations at a time, before back tracking over the relatively few skipped iterations once a difference has been found.

3.1.4 Hop Values

The hop value determines the number of elements checked during a comparison. For example, a hop value of two would compare every 2nd cell in the interesting arrays.

To determine the effects of varying this hop value, the CDTV script was run on programs called *bugy_greg_test* and *greg_test*. These programs had a two dimensional array, into which a randomly positioned error was introduced in the buggy version at time step *t*=5. The error was constructed not to occur at boundary points, which remain fixed at zero as with the SFD programs, to ensure a difference occurred.

These experiments can be reproduced using the *runCDTVTest1.cli* script, located in the *hopTests* sub-directory. Changing the variable *nMax* will change the maximum hop value tested and changing the variable *iMax* will change the number of runs performed at each hop value. Raw results will be saved in the results sub directory.

The experiments showed that increasing the hop value on *bugy_greg_test* and *greg_test* reduced the amount of time needed to first find a difference. Moving from a hop value of 1 to a hop value of 2 cut the time by over 50%. However further increases gave a smaller reduction, with the runs peaking around a hop value of 10, suggesting that other factors were slowing the script down at this point, perhaps the necessity of performing extra runs before terminating. An average of 25 iterations up to a hop value of 14 is shown in Figure 4. As the hop value increases so does the delay in finding the first difference, as shown in Figure 5.

Figure 4 and Figure 5 show that in some cases, a minimal comparison approach using hop values can offer speed ups when compared to comparing every element in the array. However some differences will not propagate like this and hence this method is unsuitable generally, for example, for the SBLI code

3.1.5 Statistical Comparisons

As such, a method to compare arrays by using statistics rather than element by element was designed and included. This method makes use of the TotalView *dprint –stats* command to return statistical information about the array, like minimum element, maximum element, median, mean, zero count etc. For both threads, this information is parsed into an array and it is these arrays that are then compared for differences.

Statistical comparisons offer a substantial speed up. Even when using a hopValue of +10, statistical comparison is still more than 50% quicker than an element by element comparison. This method of comparison also correctly identifies the first instance of any differences every time.

Statistical comparison can be switched on by setting compareByStats to 1.

One disadvantage of statistical comparison is that it cannot provide the index of a difference, but this can be overcome by doing a target index by index search at the determined time stamp.

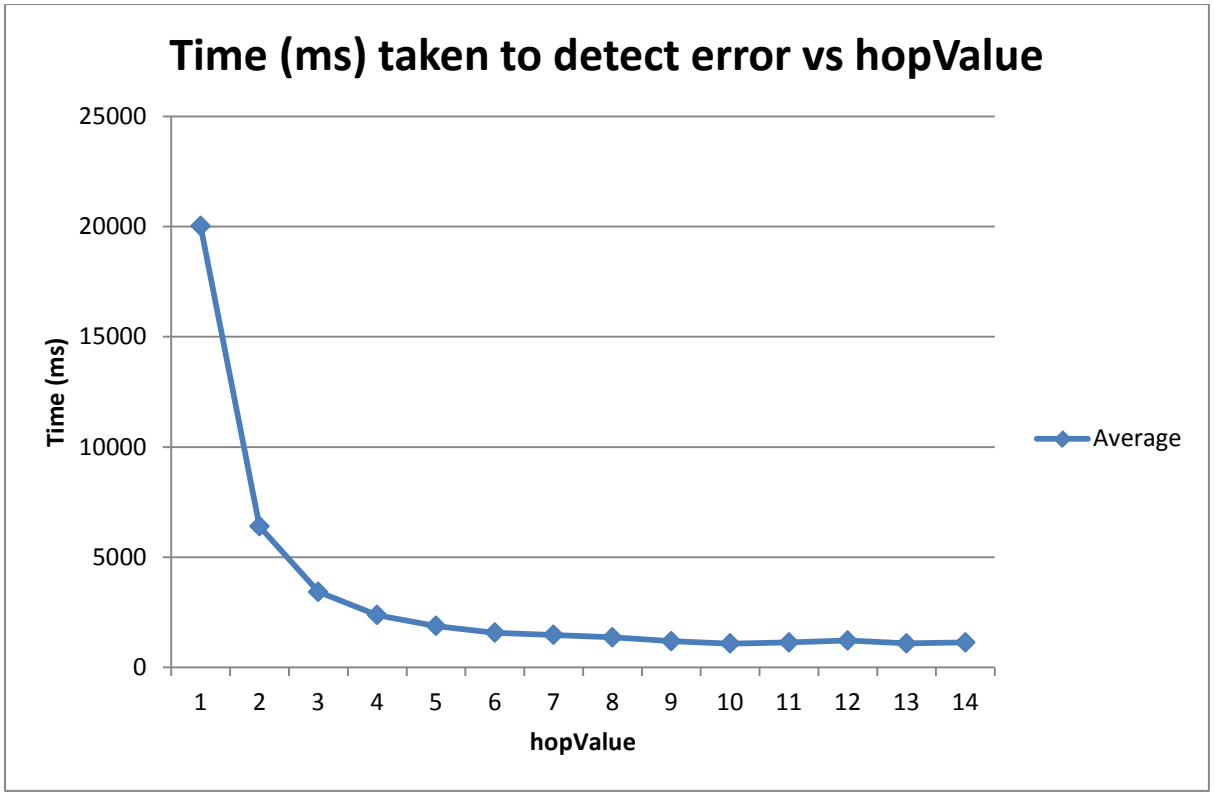

Figure 4. Average time (ms) taken to detect error vs hopValue on *greg_test

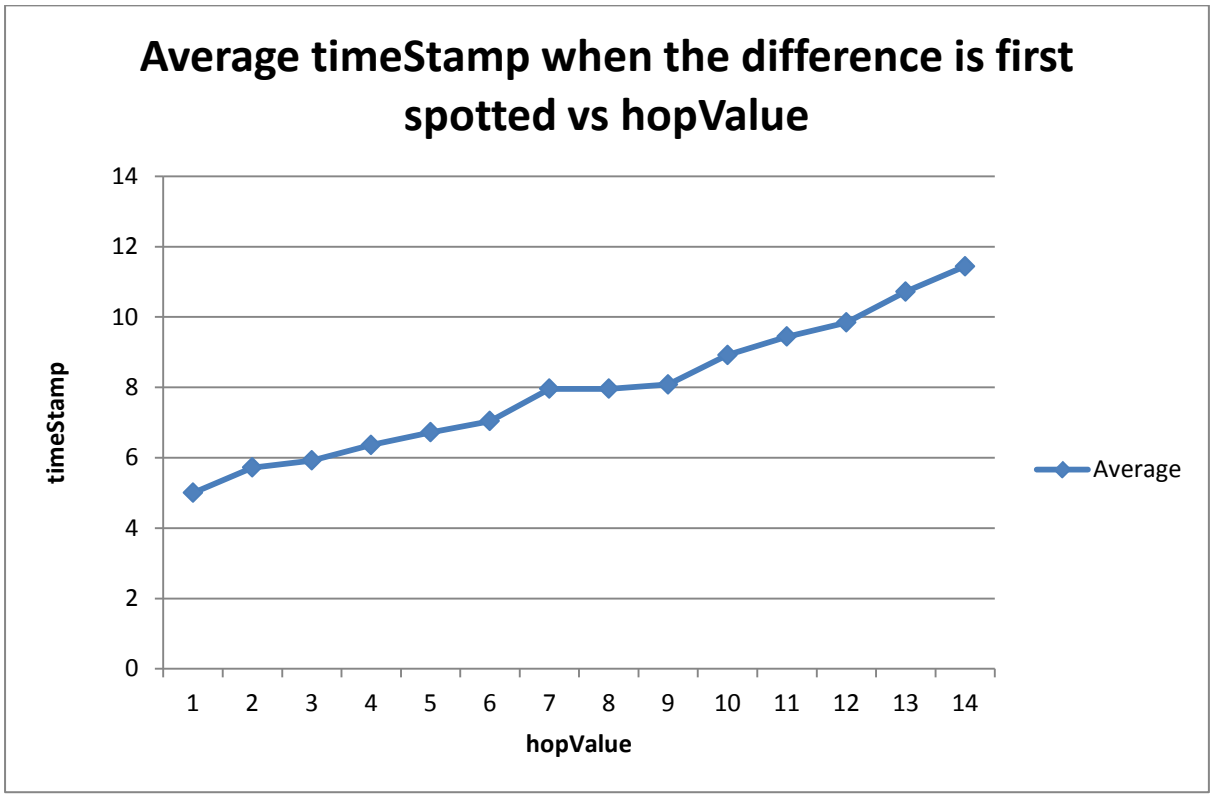

Figure 5. Average timeStamp when the difference is first spotted vs hopValue on *greg_test

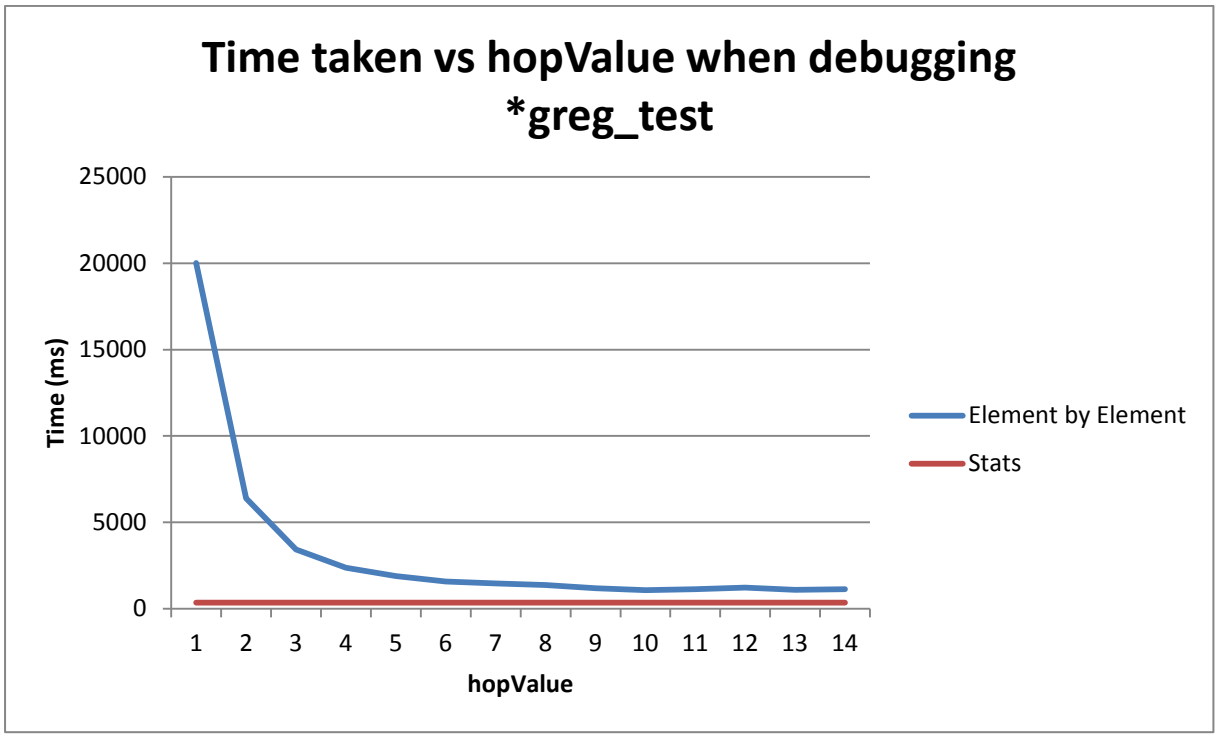

Figure 6. Time taken vs hopValue on *greg_test

3.1.6 Further improvements to serial capability

An internal iteration counter was included to display the count of the number of difference searches. This was implemented because it was noted in the SBLI code that the breakpoint was hit many times during one time iteration, such as a for loop calling the function twice each iteration. The second iteration counter was introduced to further localise the source of the difference, i.e. on the second call to the bugy function, the simulation timestamp would still say 0, but the second counter would say 1. A different approach to this second counter could be to refine the breakpoint specification, by use of *leapValue* to compare every x number of times the breakpoint is hit

The exit process was redesigned to be more programmatically useful. The script no longer just exits but rather returns 1 if it finds an difference, 0 if no difference is found and <0 if an error occurs.

Most print statements can now be diverted to log files if required. Output can be separated into 3 files, a hit log, an error log and a “standard” log. The outer run scripts will create one set of output files per invocation, which may compare multiple arrays; this was because otherwise there would be many hit logs with only one line.

3.1.7 Introducing Parallel Code Support

Support for parallel code was first tested using OpenMP using the recomp family of programs. Whilst these programs use OpenMP to achieve parallel execution, the style is very much that of MPI. This design decision was taken to allow a better migration towards MPI once the major complications of introducing parallel support had been ironed out using the relatively simple OpenMP.

This move required changes to breakpoint width, command focus and the comparison methods.

Breakpoints were required to be set to have a “width” of thread, ensuring all threads across both processes were at the same point in execution, so that meaningful comparisons can be made. This change was made by modifying dbreak to use the –t flag to set a breakpoint which tells the debugger to stop just the thread rather than the whole process when the breakpoint is hit. This

modification also provided a reason to refactor the script, separating loading programs from setting (and deleting) breakpoints allowing for a better standard of code.

Next, methods required for ensuring the two programs remained in sync with each other had to be altered to deal with multiple threads; this required changing the focus in which commands were executed. For example in syncAdvance previously, a command was *f 1 dgo*. This is ambiguous as to whether it is referring to p1 or t1.1, however in a single threaded executable it does not matter as the two are interchangeable. In parallel code however this is an issue, as it was intended that *syncAdvance* to advance the entire process just not a single thread. As such, some methods like *syncAdvance* were changed to affect processes, while others such as the look up of timeStamps were changed to more precisely control threads.

Also introduced into the *findDifference* methods were variables denoting the current thread number. These variables, *foc1* and *foc2*, allow the traversal of threads by the *findDifferencePar* method without too much change to the previous serial methods, *foc1* and *foc2* simply replace the hardcoded process variables which are set to 1.1 and 2.1 for serial execution.

In order to achieve meaningful comparisons between differently composed data sets, a method for single array element comparisons between two differently de-composed but logically identically distributed arrays was needed. This method was first achieved by including a 1 dimensional thread map in the supplied program that would keep an explicit record of which data elements were sent to which process. This map would be populated at domain decomposition time. Using this map required a function to convert from indexes in the interesting array to the indexes used by the thread map.

To differentiate between the various kinds of comparative debugging, a parallel code flag was introduced to determine which debugging scenario is in use, i.e. serial to serial, serial to parallel etc. The flag could accept 5 values, a value of:

- **0** or **SS** indicates the script is debugging two serial versions of the code
- **S2P** indicates the script is debugging a (working) serial code and a broken parallel code
- **PPS** indicates the script is debugging two versions of parallel code that are identically decomposed.
- **PPD** indicates the script is debugging two versions of parallel code that are differently decomposed.

The changes described in this section have not changed the script's execution for serial codes.

3.1.8 MPI Support

Initially, a separate branch was created for work on including MPI. In this branch MPI programs can be loaded into the CDTV script the same way as non MPI tasks, using the variables *brokenCode* and *workingCode*, such as in Figure 7.

```
set brokenCode "new/pdns3d-new.x -mpi \"Intel MPI-Hydra\" -np 4"
```

Figure 7. An example of how to debug an MPI enabled program with CDTV

The MPI version of CDTV removes the necessity of a thread map when comparing differently composed datasets. This decision was taken because there was no guarantee that the data decomposed and used to populate the thread map would relate in a bijective way to the array being searched. For example, perhaps only a column, row or subset of the data is used in the interesting array which could result in an array of different size to that which the thread map was populated to handle. The lack of a thread map also minimizes the changes needed to the source code of the programs to be debugged.

Instead, CDTV now expects a function called *transformIndices* to be defined by the developer in the outer run script. This TCL function stub must be provided by the developer and convert between the indexing of the working and broken code. The TCL function can be a wrapper to an already existing function if such a function already exists in the source code.

This function stub is passed the process number of the main working thread, i.e. rank 0 in MPI or process 1 in OpenMP/serial code; it is passed this process number because it is assumed the size of the sub arrays will be equal to the size of the first until the edge cases. It is also passed the indices for the elements location in the working code. The function returns a list of length 4 containing the corresponding working process id and the indices.

The examples included use modular arithmetic and knowledge of the domain size/decomposition to determine which process the corresponding broken element is in and it's indices. These examples are not particularly efficient, as they do not store the size of the first sub domain but rather query TotalView each time, which is an expensive and unneeded operation if a static decomposition could be assumed. *transformIndices* is called each time a comparison is about to be made, however this step could be speed up by the function making use of a look up table.

The MPI version also had to handle different processes being assigned different ranks in different runs. In the contrived OpenMP examples, each thread was assign some data based on its thread ID number, whereas in the SBLI code each process is assigned data based on its rank. Hence for meaningful comparisons, the script should not compare arrays based on process number, but by the rank of the process. To achieve this, functions were included to determine the rank of a given process, as well as the process associated with a rank, given a sub list of processes to check through. This sub list is required because there will be two processes with rank 0, one from the broken code and one from the working one.

3.1.9 Combining OpenMP and MPI

When developing the MPI branch, very few changes had to be introduced to the underlying script and, primarily, these changes were focused on the removal of the thread map handling code. As such, the MPI branch was easily combined with the OpenMP branch, with the thread map capability being migrated to the *transformIndices* function for the early, simple, recomp examples. As such, examples exist of transformIndices correctly mapping both threads and processes

3.1.10 Breakpoint Placement

Design and development of the CDTV script revealed more information on breakpoint placing and, specifically, how parallel regions in OpenMP can affect placement.

As a general rule, breakpoints should be placed at points where no communication is required between threads, whether this communication is via MPI or OpenMP shared memory. This is because it will ensure that each thread is capable of reaching the breakpoint without deadlocking while waiting for a communication.

Placing a breakpoint before the end of a parallel section as in Figure 8 creates the condition where thread 0 has hit the breakpoint but the other threads are still running. The CDTV script waits for the other threads to hit the breakpoint before continuing, which does not occur.

Also, putting a breakpoint before a parallel region can cause similar issues, TotalView still sees there are many threads running, despite *omp_get_num_threads()* showing only 1. This causes the CDTV script to wait for these phantom threads to hit the breakpoint, which does not happen. This is a fatal error as it prevents debugging after a parallel region. Even introducing a *omp_set_num_thread(1)* does not fix this issue.

```
#pragma omp parallel
{
//some stuff
nothing = 0 \\breakpoint
}
```

Figure 8. Breakpoint placement 1

Presumably, the OpenMP runtime is trying to be efficient as there is a fairly high overhead for creating and destroying threads. The OpenMP runtime may attempt to only do that once (or a small number of times). As such, OpenMP may be "idling" threads rather than killing and recreating them, which causes problems for the CDTV script as TotalView can still see them. However, the script could be adapted to handle this more nuanced mode, perhaps by using process width breakpoints when it knows only one thread will hit the breakpoint.

Breakpoints of the form of Figure 9 appear not to cause any problems. Although this has only been tested with one breakpoint, if there were multiple breakpoints then the problem in Figure 8 may come into play again.

```
#pragma omp parallel
{
//some Computation e.g. a for loop
nothing = 0;
//some Computation e.g. a for loop
}
```

Figure 9. Breakpoint placement 2

Breakpoint placement and MPI has not been explored in this project.

3.2 Demonstration: two serial codes

The CDTV script is able to correctly detect the bug in the SFD code. Lines 65 -69 of both programs are as shown in Figure 10**Error! Reference source not found.**, in the buggy version, the variable *bug* has been set to 1, otherwise *bug* is set to 0. The command *totalviewcli –s runSFD.cli* in the cli will load the programs and set a breakpoint at line 78, which is just after the array has been fully iterated through, within a loop. The script will compare *main_data_array* for differences, which it finds as shown in Figure 11, which corresponds to the if-statement in Figure 15.

```
if( (bug==1) && (t == 5) && (i==7) && (j==7)) {
  tempdata=tempdata/1;
}else{
  tempdata=tempdata/9.0;
}
```

Figure 10. The bug in the SFD code

```
Checking main_data_array...
================================================
Difference Detected!!!
t = 5
================================
Working Code:  ../files/simple_finite_diff_code
Location:      t2.1, main_data_array[7][7]
Value:         0
================================
Broken Code:   ../files/bugy-simple_finite_diff_code
Location:      t1.1, main_data_array[7][7]
Value:         1
```

Figure 11. The SFC bug detected

3.2.1 Altering the hop value

The hop value can be increased to decrease the time taken to find a difference. Setting the hop value to two means that the difference is not spotted at time stamp 5 due to the indices [7][7] not being searched, as only even numbers will be searched. When the difference has propagated into an even indexed cell, it is detected. This occurs at time stamp 6 and index [6][6] as shown in Figure 12.

```
CDTV Loop Counter:  6
Checking iteration: 6
Checking main_data_array...
 ================================================
 Difference Detected!!!
 t = 6
 ================================
 Working Code:  simple_finite_diff_code
 Location: t2.1, main_data_array[6][6]
 Value:         1.88168e-06
 ================================
 Broken Code:   bugy-simple_finite_diff_code
 Location: t1.1, main_data_array[6][6]
 Value:         0.111113
```

Figure 12. The SFD bug detected at time stamp 6

3.2.2 Altering the leap value

The leap value can also be increased to reduce the time taken to find a difference. Setting the leap value to 4 and resetting the hop value to 1 means that only one out of every four breakpoints is checked for a difference. The result of this setup is that the CDTV script checks at iterations 0, 4 and 8 and finds a difference at 8. This difference is again found at a different cell (see Figure 13), because the initial difference at [7][7] has been allowed to propagate during the additional iterations.

```
CDTV Loop Counter:  8
Checking iteration: 8
Checking main_data_array...
 ================================================
 Difference Detected!!!
 t = 8
 ================================
 Working Code:  simple_finite_diff_code
 Location: t2.1, main_data_array[4][4]
 Value:         0.00417282
 ================================
 Broken Code:   bugy-simple_finite_diff_code
 Location: t1.1, main_data_array[4][4]
 Value:         0.00554456
```

Figure 13. The SFD bug detected at time stamp 8

3.2.3 Using backtracking

Increasing the leap value can be combined with the boolean backtrack flag, called *backTrackBool*. Setting this flag to 1 will result in the tracking of a difference to its earliest appearance. Setting the leap value to 4, the hop value to 1 and the back track flag to 1 has the result of first detecting a difference at time stamp 8, then back tracking through time stamps 7, 6 and 5. The script then identifies time stamp 5 as the first instance of the difference and detects the original location i=7,j=7. The output of this run is shown in Figure 14.

```
CDTV Loop Counter:  8
Checking iteration: 8
Checking main_data_array...
 ================================================
 Difference Detected!!!
 t = 8
 ================================
 Working Code:  simple_finite_diff_code
 Location: t2.1, main_data_array[4][4]
 Value:         0.00417282
 ================================
 Broken Code:   bugy-simple_finite_diff_code
 Location: t1.1, main_data_array[4][4]
 Value:         0.00554456
Thread 1.1 hit breakpoint 2 at line 77 in "main"
Thread 2.1 hit breakpoint 1 at line 77 in "main"
Checking main_data_array...
 ================================================
```

```
 Difference Detected!!!
 t = 7
 ================================
 Working Code:   simple_finite_diff_code
 Location: t2.1, main_data_array[5][5]
 Value:          0.000228519
 ================================
 Broken Code:    bugy-simple_finite_diff_code
 Location: t1.1, main_data_array[5][5]
 Value:          0.0125742
Thread 1.1 hit breakpoint 2 at line 77 in "main"
Thread 2.1 hit breakpoint 1 at line 77 in "main"
Checking main_data_array...
 ================================================
 Difference Detected!!!
 t = 6
 ================================
 Working Code:   simple_finite_diff_code
 Location: t2.1, main_data_array[6][6]
 Value:          1.88168e-06
 ================================
 Broken Code:    bugy-simple_finite_diff_code
 Location: t1.1, main_data_array[6][6]
 Value:          0.111113
Thread 1.1 hit breakpoint 2 at line 77 in "main"
Thread 2.1 hit breakpoint 1 at line 77 in "main"
Checking main_data_array...
 ================================================
 Difference Detected!!!
 t = 5
 ================================
 Working Code:   simple_finite_diff_code
 Location: t2.1, main_data_array[7][7]
 Value:          0
 ================================
 Broken Code:    bugy-simple_finite_diff_code
 Location: t1.1, main_data_array[7][7]
 Value:          1
Thread 1.1 hit breakpoint 2 at line 77 in "main"
Thread 2.1 hit breakpoint 1 at line 77 in "main"
Checking main_data_array...
```

```
Difference First Spotted At Time: 5, see above for location
```

Figure 14. The SFD bug detected at time stamp 8 and back tracked

3.2.4 Using statistical comparisons

The biggest speed up comes from the use of statistical comparisons between the two arrays. This feature is switched on by setting the boolean flag compareByStats to 1. As a result of this, arrays are first compared using statistical properties of the array, which are accessed relatively quickly via TotalView. If a difference in the statistics is discovered, the script will then perform an element by element comparison to locate the indices of the difference (see Figure 15).

```
CDTV Loop Counter:  5
Checking iteration: 5
Checking main_data_array...
Difference at: Statistics of main_data_array, at timestamp 5
Checking main_data_array...
 ================================================
 Difference Detected!!!
 t = 5
 ================================
 Working Code:   simple_finite_diff_code
 Location: t2.1, main_data_array[7][7]
 Value:          0
 ================================
 Broken Code:    bugy-simple_finite_diff_code
 Location: t1.1, main_data_array[7][7]
 Value:          1
```

Figure 15. The SFD bug detected at time stamp5 using statistical comparisons

A difference tolerance can also be set for direct element by element comparisons; this tolerance is the minimum absolute difference that will be reported between the working and the broken code. As such, if the difference tolerance is set to 1.1 for the SFD programs, no difference will be detected. The tolerance is set in the outer run scripts by changing the variable *diffTolerance*. It currently is unused in statistical comparisons.

3.3 Demonstration: serial and parallel runs

The CDTV script can detect differences in parallel programs using OpenMP and MPI. The script can compare serial programs with parallel programs. The recomp family of programs provide scope to explore this capability using OpenMP and SBLI provides scope to explore the MPI capability.

3.3.1 OpenMP

To run the example showing comparisons between serial and parallel programs, lines 232 – 239 of runRecompFinal.cli should be set as in Figure 16 and also line 270, in the same file, as in Figure 17. When doing a serial to parallel debug, the number of threads is not stated explicitly by the developer, as CDTV can assume everything that isn't process 2, the working process, is a thread belonging to the broken code. Also, the size given to the script must correspond to the size variable

of the array in the serial, working, code.

```
#what programs
set brokenCode recomp_test_par_cs25
set workingCode recomp_test_serial
#set if you are debugging a parallel region, if not 0
set parallelCode S2P
set parallelType "OpenMP"
#set brokenThreadNum NA
#set workingThreadNum NA
```

Figure 16. The code necessary to run the OpenMP S2P example

```
set debugElem {privateData main.omp_fn.0 1 {arraySize} main  }
```

Figure 17. The code telling the OpenMP S2P example what to debug

The script then finds the difference as shown in Figure 18. Given the difference in the two programs, this result is expected as in the broken code, the second thread will add 0 to the element at its first index, which is 25.

```
CDTV Loop Counter:  0
Checking iteration: 0
Checking privateData...
findDifferenceS2P overwritten
 ================================================
 Difference Detected!!!
 t = 0
 ================================
 Working Code:  recomp_test_serial
 Location: t2.1, privateData[25]
 Value:         0x00000032
 ================================
 Broken Code:   recomp_test_par_cs25
 Location: t1.2, privateData[0]
 Value:         0x00000019
```

Figure 18. difference detected between recomp_test_par_cs25 and recomp_test_serial

In all the OpenMP examples shown in this report, if the source code is edited to alter the *chunksize*, the use of the thread map concept allows the script to continue to work correctly, without any alteration. In fact, the only difference between *recomp_test_par* and *recomp_test_par_cs25* is the *chunksize* parameter, and the associated parameter of *threadnum*.

3.3.2 MPI

The MPI example does not use the thread map concept as the simple OpenMP examples do. Instead, the *transformIndices* function relies on developer knowledge of the domain decomposition prior to debugging. As a result of this, the MPI demonstrations presented here and later in the paper are less flexible than their OpenMP counter parts. This is not to say that that the script cannot debug different decompositions, just that doing so would require editing of the *transformIndices* function and an understanding of how the domain is decomposed. For the aid of demonstration, two run files have been included. The underlying script for all the examples in the report, other than the hop value testing, has been the same.

- *runSBLIFinal4.cli* has a function for mapping from serial to a 4 way decomposition.
- *runSBLIFinal8.cli* has a function for mapping from a 4 way decomposition to an 8 way

The result of *runSBLIFinal4.cli* is shown below, whilst the result of *runSBLIFinal8.cli* is shown later in Figure 26.

In the serial to 4 way decomposition case, the script correctly detects a difference between the working code and the broken code. It locates this difference to be between t2.1, dfn(0,17,0) in the working code and Rank 1, dfn(0,2,0) in the broken code, shown in Figure 19. This is to be expected because (0,17,0) in the serial case maps to Rank 1 (0,2,0) in the 4 way decomposition.

```
CDTV Loop Counter:  0
Checking iteration: 0x00000000
Checking dfn...
 ================================================
 Difference Detected!!!
 l = 0x00000000
 ================================
 Working Code:  pdns3d-debug.x
 Location: t2.1, dfn(0,17,0)
 Value:         -1.38777878078145e-16
 ================================
 Broken Code:   new/pdns3d-new.x -mpi "Intel MPI-Hydra" -np 4
 Location: Rank 1, dfn(0,2,0)
 Value:         4
```

Figure 19. The SBLI difference detected in a serial to parallel case

3.4 Demonstration: two OpenMP parallel codes

The script can also compare OpenMP codes with both identical and different data decompositions.

3.4.1 Parallel to parallel decomposition with the same decomposition

To run the example showing comparisons between two parallel programs with the same decomposition, lines 232 – 239 in *runRecompFinal.cli* should be set as in Figure 20 and also line 270, in the same file, as in Figure 21.

Again in this setup, the CDTV script does not need to be explicitly told the number of tasks in each program, as due to the way the programs are loaded, the broken code is loaded first. As a result of loading in this way, if the script is in OpenMP mode, the script can infer that the second process, and all its threads, must be the working one.

Figure 21 is similar to Figure 16, with the only difference being the size of variable passed. This is because the script requires the size variable of the working code, which in this case is *chunksize* but in the serial version there is no *chunksize* variable to use, only *arraysize*.

```
#what programs
set brokenCode recomp_test_par_cs10_2
set workingCode recomp_test_par_cs10
#set if you are debugging a parallel region, if not 0
set parallelCode PPS
set parallelType "OpenMP"
#set brokenThreadNum NA
#set workingThreadNum NA
```

Figure 20. The code necessary to run the OpenMP PPS example

```
set fudge {privateData main.omp_fn.0 1 {chunkSize} main  }
```

Figure 21. The code telling the OpenMP S2P example what to debug.

As the script is comparing recomp_test_par_cs10_2, it detects a different difference to that found in Figure 22. The script detects that at OpenMP thread 8 (TotalView Thread 9) at index 4, the value has been set to 999, or 3e7 in hexadecimal (shown in Figure 22).

```
CDTV Loop Counter:  0
Checking iteration: 0
Checking privateData...
 ================================================
 Difference Detected!!!
 t = 0
 ================================
 Working Code:  recomp_test_par_cs10
 Location: t2.9, privateData[4]
 Value:         0x000000a8
 ================================
 Broken Code:   recomp_test_par_cs10_2
 Location: t1.9, privateData[4]
 Value:         0x000003e7
```

Figure 22. The difference detected between recomp_test_par_cs10 and recomp_test_par_cs10_2

3.4.2 Parallel to parallel decomposition with a different decomposition

When in PPD mode, involving parallel to parallel comparisons with different decompositions, the script could have been designed to work only with OpenMP and not require explicit statement of the number of threads in each process. This is because it can still infer that the second process is the working one. However, in MPI mode the thread/task numbers are needed because the script cannot

infer this information as it can in the other two cases, S2P and PPS. As such, both methods require the thread/task number to be stated explicitly, to avoid the *findDifferencePPD* method needing to be aware that what type of parallelism was in use. To run the example, lines 232 – 239 should be set as in Figure 23 and also line 270 as in Figure 21. This example detects the same error as the S2P example in Figure 18 (see Figure 24).

```
#what programs
set brokenCode recomp_test_par_cs25
set workingCode recomp_test_par_cs10
#set to 1 if you are debugging a parallel region
set parallelCode PPD
set parallelType "OpenMP"
set brokenThreadNum 16
set workingThreadNum 40
```

Figure 23. The code necessary to run the OpenMP PPD example

```
CDTV Loop Counter:  0
Checking iteration: 0
Checking privateData...
findDifferencePPD overwritten
 ================================================
 Difference Detected!!!
 t = 0
 ================================
 Working Code:  recomp_test_par_cs10
 Location: Rank t2.3, privateData[5]
 Value:         0x00000032
 ================================
 Broken Code:   recomp_test_par_cs25
 Location: Rank t1.2, privateData[0]
 Value:         0x00000019
```

Figure 24. The difference detected between recomp_test_par_cs10 and recomp_test_par_cs25

The *runRecompFinal.cli* file redefines the compare methods slightly. The change is a purely cosmetic one, altering the format of returned hit messages by changing the number of indices printed, in a 1 or 2 dimensional array the 3rd indices is not needed and potentially confusing. This change is evident in Figure 18 and Figure 24 and can be seen by the lines *findDifferenceS2P/findDifferencePPD overwritten*. Such a change is not needed in the PPS case, shown in Figure 22, as this is handled by repeated serial comparisons which were introduced into the script at the very beginning and as a result have better support for multiple dimensions than the parallel methods.

3.5 Demonstration: two MPI parallel codes

All modes described in the OpenMP section can be used on the SBLI example. However, only the S2P and PPD cases are shown in this report. Serial to serial comparisons are not shown as the bug only occurs in the Rank 1 process, which does not apply to serial code. Also, the result of PPS comparisons are not shown here, because this method of comparison is essentially a serial to serial comparison, which has been demonstrated in both sections 3.2 and 3.3.1 previously. However, PPS comparisons are possible without explicitly telling CDTV the number of broken and working tasks. This is because the script can deduce that the processes loaded have the form in Figure 25 and as such can partition all the tasks into working and broken groups.

$$\{ b\ w\ b^n\ w^n \}$$

Where:

- b = task in the broken code
- w = task in the working code
- b^n = n tasks in the broken code
- w^n = n tasks in the working code

Figure 25. The format of MPI tasks in the PPD example

By running *runSBLIFinal8.cli* the 4 to 8 way decomposition comparison can be demonstrated. The CDTV script correctly detects a difference between the working and broken code, as shown in Figure 26. The script locates a difference at Rank 0, dfn(0,2,15) in the working code and Rank 1, dfn(0,2,0) in the broken code. This is to be expected because Rank 0, dfn(0,2,15) maps to Rank 1, dfn(0,2,0), which is where the bug affects the broken code.

```
CDTV Loop Counter:  0
Checking iteration: 0x00000000
Checking dfn...
 ================================================
 Difference Detected!!!
 l = 0x00000000
 ================================
 Working Code:  pdns3d-debug.x -mpi "Intel MPI-Hydra" -np 4
 Location: Rank 0, dfn(0,2,15)
 Value:         1.69135538907739e-16
 ================================
 Broken Code:   new/pdns3d-new.x -mpi "Intel MPI-Hydra" -np 8
 Location: Rank 1, dfn(0,2,0)
 Value:         4
```

Figure 26. The SBLI difference detected between a 4 way decomposition and a 8 way decomposition

4 Availability

Scripts have been made freely available to the community vi the CCPForge[7] software repository, under project name “cdtv”[8].

5 Suggestions for Future Work

There is much additional capability that could be desirable for the script. The following section details a few of the ideas that have either been identified by project stakeholders or other members of STFC.

5.1 Further Testing

Testing has been limited to on parallel bugs that occur in the first, or only, iteration of program execution. Therefore, more testing is needed to determine if all the various options for parallelism and comparison (i.e. leap and hop values) work together as expected.

Further testing of how the hop value and back tracking features function together is also needed, as this option has undergone little or no testing up till now. It is believed, however, that the hop value

[7] http://ccpforge.cse.rl.ac.uk/gf/

[8] http://ccpforge.cse.rl.ac.uk/gf/project/cdtv/

will not change when a difference is found and as such won't be able to find any earlier difference. Ideally, when back tracking the hop value would set to 1.

5.2 Integrate Iteration Counter

When the script executes, it keeps track of the timestamps of each program as well as maintaining an internal count of the number of its own internal loops. This internal counter can help a developer locate the difference if it occurs some way into the program execution. Currently however, the internal counter is only printed to the terminal; it is not included in the "Difference Detected" messages (DDMs). Further development to the CDTV script, such as a function that determined the look of all DDMs could alleviate this problem, as well as provide better formatting for DDMs of dimensions other than 3.

5.3 Thresholds

Currently the threshold acts and an absolute threshold, with no regard to what is actually being compared. As a result of this, differences that are high in terms of relative difference, but low in terms of absolute difference, could go unnoticed. An option to select which type of threshold could be included to allow both types of thresholds. Another option could be to allow different values of the threshold to be selected, for example the threshold is 1% of the max element in the array during this iteration.

5.4 Compiler flags

TotalView requires compiler flags to correctly see *#define* directives in C and Fortran. This can be overcome in Fortran by using the *-g -debug-parameters all* flag instead of just *-g*. Although the necessary option to do the same in C has not yet been found, the problem can be worked around by creating a variable equal to the directive, such as Figure 27. Alternatively to this, the use of a *static const* declaration rather than a define directive also work around this issue.

```
int chunkSize = CHUNKSIZE;
```

Figure 27. One work around for the define directive

5.5 Visualization of Differences

TotalView's visualization tool could also be exploited to display working and broken arrays as well as the difference between cells to enable a developer to understand the cause of the difference that the CDTV script has discovered. Further to this, upon detection of a difference, rather than exit, the script could load up the GUI. Doing so would again allow the developer to better understand and explore the cause of a difference found. These two suggestions emerged from discussions at STFC/Daresbury.

5.6 Index conventions

Currently, the script will search from 0 to n-1, where *n* is the value of the dimension variable supplied. This is not ideal, as different languages use different conventions; C starts from 0 whereas Fortran starts from 1 for example. Also, some programs, like SBLI, make use of negative indices (-1). Support for these variations in convention could be implemented by changing the dimension variable list to include the start point of the indexing.

5.7 Multi language comparison

During discussions at Daresbury, both before and during the project, the desire for multi-language comparative debugging has been identified. This could take the form of comparing a working C code and a broken Fortran port or vice versa. In theory, this would be possible as TotalView can abstract

away the majority of the specifics of what language the programs are written in. Such capability should also be easy to implement as the comparison methods are language agnostic, only the *findElement* methods which deal directly with the TotalView interface would have to be changed. This change could be as simple as having a list of languages rather than a singleton, where the language being used to find an element determined by the *processId* passed. E.g. the script is told the broken code is written in Fortran, *findElement* is passed a *processId* of 1 (corresponding to the broken code), hence use Fortran syntax to find an element. This could all be wrapped up in a if statement so that if the language list is of length 1, current behavior is maintained, thus negating the need for another option flag.

5.8 Submit CDTV as a job on the iDataplex

When CDTV has been run on the iDataplex, it has only been run on the login node. It has not been investigated how CDTV could be submitted to compute nodes as a job using the *bsub* command. Such capability is important as the programs the script would debug in practice are run on the compute nodes due to their high degrees of parallelism.

5.9 Multiple errors

Currently, the CDTV script will exit after finding one error. The CDTV scripts usefulness could be improved if all difference were reported on, as opposed to only the first or the most prominent. This change would allow the developer to track the difference forward through program execution.

5.10 Save and Restore

The upcoming 8.14 release adds “save/restore” capability to ReplayEngine. This could allow for the ability to save a run and use that as the comparative baseline or save a buggy instance. Such a workflow would be particularly useful for transient bugs that come and go.

5.11 MemoryScape

TotalView also includes MemoryScape, which provides heap memory debugging functionality that can be accessed from TotalView scripts. Memory State can be stored in memory debugging data files that can be examined offline or used for comparisons that show the evolution of a single process through time. As with “save/restore”, this feature could be used to provide a baseline or repeating example of a transient bug for comparative debugging.

6 Conclusions

Over the six months of this project, several demonstration comparative debugging scenarios using TotalView with TCL has been created. Two are summarized here below.

Both scenarios use an outer “run” script to convey information to the inner comparative debugging TotalView script (CDTV). These run scripts convey to the CDTV script information that includes:

- Which logical arrays can be recomposed (*interestingArray*)
- The arrays logical dimensions
- How the data is decomposed (*transformIndices*) if applicable.

One scenario uses the simple_finite_difference_code (SFD) programs provided by TotalView to demonstrate serial to serial comparative debugging as well as the range of speed up options such as skipping breakpoints, less thorough search and statistical comparisons of arrays. The SFD programs both feature a 2d data array that is "evolved" forwards over a series of timestamps, with the bug causing the value at [7][7] in the *main_data_array* to be set to 1 at time stamp 5.

The *runSFDFinal.cli* script will correctly detect the difference caused by the bug and by changing variable such as *hopValue*, *leapValue*, *backTrackBool* and *compareByStats* the performance can be altered as below[9]:

- Setting *hopValue* to 2 will delay finding the difference to time stamp 5
- Setting *leapValue* to 4 will delay finding the difference to time stamp 8
- Setting *leapValue* to 4 and *backTrackBool* to 1 will find the difference at time stamp 8 before finding its initial appearance at time stamp 5.
- Setting *compareByStats* to 1 will detect a statistical difference at time stamp 5 before localising it at that time stamp.

The serial to serial case has been tested extensively using a variety of test programs. CDTV supports multiple dimensions and languages, as well as offering a range of speed up options such as skipping breakpoints, less thorough search and statistical comparisons of arrays. Some of these features, such as multiple languages and dimensions, have been fairly well tested in the parallel case, whilst the speed-up options are relatively untested in the parallel case.

The second scenario involves the SBLI code. In this scenario, a bug has been introduced into the new version of the SBLI code, which affects the *dfn* array in the *d1eta_2* subroutine in the *deriv* module in the *cent2.f* file. The effect of this bug is to set the value in cell (0,2,0) in the Rank 1 process to 4. The prototype comparative debugging script is capable of correctly detecting the difference in the *dfn* array caused by the bug. Detection occurs in the serial to parallel case (*runSBLIFinal4.cli*), parallel to parallel case with the same domain distribution and the parallel to parallel case with different domain decompositions (*runSBLIFinal8.cli*).

In the parallel to parallel case with the same decomposition, any number of threads can be used and the difference is detected, although a four-way decomposition is the most tested.

In the parallel to parallel case with different decompositions, only a four to eight way comparison can currently be demonstrated. However, the thread numbers are only limited by the *transformIndices* function provided by the run script, which requires developer knowledge of how the domain is decomposed. There is no foreseeable reason why a function to map between arbitrary domains of the SBLI domain could not be implemented.

It is also believed that similar test cases could be made for other real parallel code without major changes to the underlying CDTV script.

Acknowledgements

The authors would like to express their gratitude to the considerable help provided by Chris Gottbrath and Dean Stewart of RogueWave Software during the course of the project.

References

Ashworth, M., D.R. Emerson, N.D. Sandham, Y.F. Yao, and Qinling Li. "Parallel DNS using a compressible turbulent channel flow benchmark" in Proc. ECCOMAS CFD Conference, 2001

Gottbrath, C., RogueWave Software, private communication, 2013

Rogue Wave Software, 2013, TotalView® Graphical Debugger. Retrieved January 29, 2014, from Rogue Wave Software: http://www.roguewave.com/products/totalview.aspx

[9] If a variable is not mentioned, set to its default value

Sandham, N.D., Direct Numerical Simulation of Transitional and Turbulent Flows with Strong Compressibility Effects, Proceedings of the 22nd International Conference on Parallel Computational Fluid Dynamics, 2010

Sunderland, A.G., M. Ashworth, C. Moulinec, N. Li, J. Uribe and Y. Fournier, Towards Petascale Computing with Parallel CFD codes, Parallel Computational Fluid Dynamics 2008, Lecture Notes in Computational Science and Engineering Volume 74, 2010, pp 309-320

Yao, Yu-Feng, A.A. Lawal, N.D. Sandham, I.C. Wolton, M. Ashworth, and D.R. Emerson “Massively Parallel Simulation of Shock/Boundary-Layer Interactions”, in Proc. Inter. Conf. Applied Computational Fluid Dynamics (Beijing), pp. 728-735. 2000