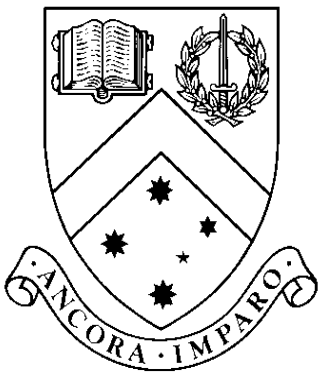
Caulfield School of Information Technology
Faculty of Information Technology
Monash University
Caulfield Campus

Data-centric Parallel Debugging Technique for Petascale Computers

Minh Ngoc Dinh
Bachelor of Computer Science (Honours)

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy
June 2012

Supervised by Professor David Abramson



This thesis is dedicated to my mum, my dad, my wife, and especially my little daughter

This thesis was produced with Microsoft® Word 2010. The main text was typeset using Garamond 12 point font and 1.5-line spacing. Program code and commands use the Consolas font. Diagrams were produced with the Microsoft® PowerPoint 2007 and equations with Microsoft Equation 3.1. The production copies were printed on an HP DeskJet 1120C printer.

Abstract

Petascale computers and computing systems have the potential to solve large-scale, data-intensive problems in science and engineering. Petascale scientific applications, such as the Weather Research and Forecasting Model (WRF), involve enormous multi-dimensional data structures and operate with hundreds of thousands of concurrent processing threads. On the one hand, programming languages and environments have evolved significantly to support parallel application developers to explore the advantages in terms of computational power and memory usage. Co-array Fortran, Split-C, MPI and OpenMP are some successful examples. On the other hand, debugging tools for highly parallel software are still immature, especially in techniques for controlling multiple processes and monitoring large scale data structures during debugging time.

Typically, contemporary parallel debuggers allow users to control more than one processing thread while supporting the same examination and visualisation operations that of sequential debuggers. This approach restricts the use of parallel debuggers when it comes to large scale scientific applications run across hundreds of thousands compute cores. First, manually observing the runtime data to detect error becomes impractical because the data is too big. Second, performing expensive but useful debugging operations, such as distributed expression evaluation, becomes infeasible as the computational codes become more complex, involving larger data structures, and as the machines become larger.

This thesis explores the idea of a data-centric debugging approach, which could be used to make parallel debuggers more powerful. It discusses the use of ad-hoc debug-time assertions that allow a user to reason about the state of a parallel computation. These assertions are modeled on programming language systems that support the verification and validation of program state as a whole rather than focusing on that of only a single process state. The advantage of this approach is the capability to reason about the massive data structure at runtime. Furthermore, on parallel machines, the debugger's performance can be improved by exploiting the underlying parallel platform. The available compute cores can execute parallel debugging functions while idling at a program breakpoint.

Declaration

This thesis contains no material that has been accepted for the award of any other degree in any other university. To the best of my knowledge, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Minh Ngoc Dinh

25th June, 2012

Notice 1

Notice 2

Acknowledgement

I wish to acknowledge the support of a number of people, without whom this thesis would not be completed.

First and foremost is my wife, Thi Thuy Trinh, who has been my constant companion and soul mate throughout this entire process and beyond, and whose support made this thesis possible. I also thank my family, especially my parents for their support and encouragement during the last three years (and a bit more).

Next is my supervisor, Professor David Abramson, who has been not only a teacher but also a great friend. I am forever thankful to the remarkable effort that he has made to provide all the resources and encouragement that any PhD candidate could possibly hope for. His supervision and guidance have provided insightful discussions about the research. I would like to take this opportunity to also thank other team members in the ccdb project including Dr. Donny Kurniawan, Dr. Jin Chao, Mr. Bob Moench and Mr. Andrew Gontarek for their collaboration and support.

Last but not least, I acknowledge the professional proofreading aids from Dr. Megan Seen and Dr. Gillian Fulcher. I am also thankful to Dr. Huy Hoang Ngo and Mr. Harilaos Serpanos for providing constructive feedbacks and comments to improve my writing.

Table of Contents

List of Publications

[1] D. Abramson, M. N. Dinh, D. Kurniawan, B. Moench, and L. DeRose, "Data Centric Highly Parallel Debugging", in *ACM International Symposium on High Performance Distributed Computing (HPDC)* Chicago, Illinois, 2010.

[2] M. N. Dinh, D. Abramson, D. Kurniawan, C. Jin, B. Moench, and L. DeRose, "Assertion based parallel debugging", in *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, Newport Beach, California, 2011 (selected as Best Paper Finalist).

[3] M. N. Dinh, D. Abramson, J. Chao, D. Kurniawan, A. Gontarek, B. Moench, and L. DeRose, "Scalable parallel debugging with statistical assertion", in *Symposium on Principles and Practice of Parallel Programming (PPoPP) - Poster*, New Orleans, LA, USA 2012.

[4] M. N. Dinh, D. Abramson, J. Chao, D. Kurniawan, A. Gontarek, B. Moench, and L. DeRose, "Debugging Scientific Applications With Statistical Assertions", in *International Conference on Computational Science (ICCS)*, Omaha, Nebraska, USA, 2012.

[5] C. Jin, D. Abramson, M. N. Dinh, A. Gontarek, B. Moench, and L. DeRose, "A Scalable Parallel Debugging Library with Pluggable Communication Protocols", in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Ottawa, Canada, 2012.

[6] M. N. Dinh, D. Abramson, and J. Chao, "Scalable Relative Debugging", submitted to *IEEE Transactions on Parallel and Distributed Systems*.

[7] M. N. Dinh, D. Abramson, J. Chao, A. Gontarek, B. Moench, and L. DeRose, "A data-centric framework for debugging highly parallel applications", submitted to *Software: Practice and Experience*.

[8] M. N. Dinh, D. Abramson, and C. Jin, "Statistical assertion: a more powerful method for debugging scientific applications", submitted to *Journal of Computational Science*.

List of Figures

List of Tables

Chapter 1

Introduction

Achieving high performance [on very large parallel computers] is only part of the productivity story. Today, supercomputing sites often spend more on software development than on hardware. Reducing the time needed to write, tune, debug and maintain HPC [high performance computing] applications is therefore critically important. ...Once an application is written, it's generally a time-consuming process to debug and tune it. Traditional debugging models just don't scale well to thousands or tens of thousands of processors.

Steve Scott, Former Chief Technology Officer, Cray Inc.

[We must] create new technology and more competition in the area of debugging tools. For many years, only one tool existed, and most vendors got out of the business of application debugging. The approaches that worked at 10-100's processors become very difficult at 1,000 processors and impossible at 10,000 processors. New funding for debugging tools, both application and system, is critical.

William T.C. Kramer – National Energy Research Scientific Computing Center

1.1 Introduction to Petascale Computing

CPU clock speed has almost reached its limits within current power constraints. To improve application speed and accuracy, more innovative use of software and parallel hardware is required. High performance computing (HPC) refers to the integration of parallel hardware (such as supercomputers or computer clusters), parallel software and network infrastructure to achieve this goal. Scientific research is a major driver of HPC because many real-world scientific problems demand extremely high levels of performance [1]. For example, DNA mapping and sequencing, climate and weather modelling, 3D protein-molecule reconstruction and molecular nanotechnology all require high levels of performance [1]. At the current leading edge, petascale[1] HPC achieves this performance, but requires hundreds of thousands of CPU cores running concurrently on a high-speed interconnected network [2]. These extremely high levels of concurrency create problems, not only in developing, but also in testing, maintaining and debugging the software [3].

1.2 Motivation

Even though programming languages have advanced over the years, parallel debugging has hardly progressed. Existing debuggers do little more than allow a user to control the flow of a program and examine its state. Moreover, while parallel debuggers allow users to control multiple independent threads of execution, they really only possess the same functionality found in their sequential counterparts. This may have been unimportant when both the program state and the size of the parallel machines were small. However, such debuggers have become unwieldy and ineffective when it comes to debugging petascale applications.

Two fundamental challenges in the current approaches to debugging large parallel applications need to be addressed.

1.2.1 The Cognitive Challenge

First, the *state* of a typical scientific application can be enormous, making it infeasible for users to examine individual cells of, what are typically, large multi-dimensional floating-

[1] Petascale computing indicates that approximately 10^{15} floating-point operations are carried out every second (that is petaflop/s)

point structures. It is simply not feasible to print out complete arrays to decide if a program is computing the correct results. We refer to this as a *cognitive* challenge, because programmers cannot integrate the data effectively into a mental model of correct execution. Currently, one of the most powerful techniques available for integrating large amounts of data into a human visible form is to use scientific visualisation tools [4, 5], but in general, it is not always possible to "see" the errors in the data.

In order to address this cognitive challenge, this thesis proposes the use of sophisticated data-mining operations to help identify outlier values in large amounts of program states. However, sophisticated data-mining operations are often more expensive to execute on a single processor. This leads to the second challenge in parallel debugging: performance.

1.2.2 The Performance Issue

A large number of computing processes are involved in the running of a typical scientific application. When simply controlling these processes, the existing parallel debuggers do not make use of the underlying parallel platform to improve their own *performance*. Thus, only debugging techniques that can be executed on a sequential frontend machine in a reasonable time are supported. To overcome this limitation, this research proposes a framework in which debugging primitives can be executed in parallel. By allowing the debugger to exploit the inherent parallel processing capabilities of the supercomputer (which is usually idle when a program is suspended at a breakpoint), we can make these complex operations fast enough to use in real time.

1.3 A Glance at an Assertion-Based Debugging Technique

Users often control the flow of the program execution to perform various debugging tasks. This approach is called *break-and-examine* [6] and it is exercised with the use of *breakpoints* and display functions such as the *print* command. This is the most common debugging method and is used successfully with procedural programming languages and debuggers such as GNU GDB [7, 8]. Examination of the program state at runtime allows users to verify expectations or to refine new expectations, and to localise the erroneous area in the source code.

Whilst effective in terms of debugging, issues such as the cognitive challenge make existing techniques unwieldy to use. A number of researchers have discussed the need

for a more data-centric view of debugging [9-11]. In a data-centric view, a user still executes a break-and-examine strategy but goes on to make statements about the contents of data structures. The veracity of these statements is then checked by the debugger.

In generalising and expanding this model, we draw on research in program proving and verification that provides a fertile base on which to build powerful debugging primitives. Assertions are used in software engineering to verify various expectations of the program's data at runtime. Assertions are usually implemented as additional code that is inserted into the program at compile or link time, making it possible to check the contents of various variables (for example, on the entry and exit of a procedure). However, language-based assertions are not meant for debugging, per se, but rather as sanity checks on the state of the program.

This research argues that the assertions, which can be executed at debug time, are a viable way to address the cognitive challenge (discussed earlier) in debugging large scientific codes. For example, such assertions can be useful for validating the state of a large distributed array. Because they are not statically compiled into the program, the assertions can also be refined iteratively in order to locate the source of an error at debug time. The following list shows examples of assertions that could be used to locate errors at runtime:

- 'The contents of this array should always be positive';
- 'The sum of the contents of this array should always be less than a constant bound';
- 'The value in this scalar should always be greater than the value of another scalar variable'; and
- 'The contents of this array should always be the same as the contents in another array'.

Evaluating assertions in a large parallel machine poses both implementation and performance issues. First, the debugger requires an understanding of the way the data is decomposed across the parallel processors. This is because the assertion may pertain to a data structure that is distributed and may never exist in a single node. Second, as a consequence of the first point, the assertion must be evaluated in each of the processors; this requires a debugging architecture that understands how the data has been

decomposed and distributed across the processors. Finally, if the assertion evaluation is executed in parallel, then the technique can adapt to the increasing machine size.

This research proposes and implements a data-centric debugging framework that uses assertions as its core. Such a framework delivers a solution for the cognitive challenge. The scheme proposed in this research also executes the assertions in parallel, making assertions over large data structures feasible and thus solving the performance issue.

1.4 Contributions

By investigating the challenges in debugging petascale scientific applications, and by developing a data-centric debugging tool using assertions as its core, this research makes a number of significant and novel contributions to the areas of parallel debugging, formal verification, data processing and HPC in general. Contributions include:

1. The analysis and evaluation of various parallel debugging techniques. This review addresses different weaknesses and limitations in using the conventional techniques for debugging large scale applications.
2. The development of a debugging framework. Data-centric approaches in debugging software have been studied previously (discussed in detail in Chapter 2). However, these approaches are either programming-language specific, or dependent on programming-language runtime systems, or cannot be scaled up to work with large parallel codes. This work generalises those attempts and provides a debugging framework in which ad-hoc experiments can be conducted with greater efficiency. In particular, the proposed debugging framework supports reasoning of the global state of the program using advanced data-mining operations and improves the scalability of the debugging process.
3. The design and the implementation of a prototype data-centric parallel debugger. The prototype debugger uses ad-hoc debug-time assertions in order to perform debugging tasks on large-scale data-intensive applications. This development includes an efficient data-reconstruction engine, a scalable data-reduction mechanism and several data-comparison techniques. This is the engineering highlight of this research where a range of advanced methods are architected, developed and evaluated.
4. The development of three different assertion templates. These templates are proposed as part of the general assertion-based debugging framework. These

include general ad-hoc assertions, comparative assertions[2] and statistical assertions. These templates demonstrate different data-reasoning techniques in order to support users in identifying and localising software defects.

5. A set of case studies that demonstrate the use of ad-hoc debug-time assertions in debugging parallel scientific codes. While each case study demonstrates a specific application, the construction and the evaluation of the debug-time assertions are general. In addition, each case study uses different debug-time assertion templates including general ad-hoc assertions, comparative assertions and statistical assertions.

1.5 Thesis Structure

The remainder of this dissertation is presented as follows.

- Chapter 2 provides background information relevant to debugging issues. It also reviews parallel computing and several parallel programming models in order to address the challenges in debugging parallel programs. The review is followed by an analysis of the current parallel/distributed debugging trends in terms of methodologies, design, scalability and applicability. The analysis focuses on the extension of traditional, sequential debugging techniques in debugging parallel problems. The weaknesses in those approaches underline the need for a more data-centric debugging method. Importantly, existing data-centric techniques are reviewed to expose the research gaps and highlight the contributions of this study.
- In Chapter 3, assertions are reviewed to highlight their benefits in debugging parallel software. Several debug-time assertion templates are proposed. Their descriptions guide the architectural decisions discussed in Chapter 4.
- Chapter 4 highlights architectural considerations in designing an assertion-based parallel debugger, which uses ad-hoc debug-time assertions as its foundation. Several key components are discussed and some advanced algorithms are proposed.
- In Chapter 5, the implementation details are provided.

[2] The type of comparative assertion addressed throughout this work is effectively the relative debugging assertion. The work in this research aims to improve the performance and the scalability of this assertion; this work does not deliver any conceptual contributions associated with the related debugging methodology. Nevertheless, the practical achievements associated with the comparative assertions are fully evaluated in Chapter 6.

- Chapter 6 demonstrates the potential of the proposed assertions in debugging large-scale parallel applications through three distinct case studies.
- Chapter 7 analyses and evaluates the performance of these assertions in debugging production codes. The objective is to provide evidence that the proposed debugging framework can be used in large-scale environments.
- Chapter 8 summarises the achievements of this research. It also provides some future directions to further improve the debugging tool and for research into assertion-based parallel debugging technology.

In summary, this chapter has introduced petascale computing and the motivation for the thesis. It has explained two issues in debugging large-scale parallel applications that need addressing: the cognitive issue and the performance issue. Solutions to these issues in the form of a list of contributions to be made by this study were also presented. The chapter closed with a description of the thesis structure. In the next chapter, we provide background information relevant to this study.

Chapter 2

Existing Parallel Debugging Technologies

Debugging is the dirty little secret of computer science. Despite all the progress we have made in the last thirty years: faster computers, networking, easy-to-use graphical interfaces, and everything else, we still face some embarrassing facts. First, all too often, computer programs don't work as they should. This makes software development costly. Too much buggy software reaches end users, leading to needless expense and frustration. That's unfortunate, but what is surprising is the fact that when something does go wrong, the people who write these programs still have no good ways of figuring out exactly what went wrong. Debugging is still, as it was thirty years ago, largely a matter of trial and error.

Henry Lieberman, [12].

This chapter reviews the evolution of debugging from traditional sequential debuggers to highly advanced parallel debuggers.

First, the chapter briefly defines a general software debugging process using two formal models: *Araki's model* [13] and *the scientific model* by Zeller [14]. The examination of these two models creates a conceptual background for discussing conventional sequential debugging methods. Importantly, several fundamental debugging methods that still affect the design and the implementation of today's parallel debuggers are described.

Second, the chapter gives an overview of parallel computing and parallel programming models. This is essential to reveal the core challenges in debugging parallel applications.

Third, the chapter presents modern parallel debugging trends. Here, the text focuses on the crucial role played by the fundamental sequential debugging methods in the design of their parallel counterparts. Describing these roles highlights the limitations of contemporary parallel debugging techniques and emphasizes the importance of a more data-centric debugging approach.

Fourth, the chapter provides an up-to-date literature review of several existing data-centric debugging techniques. Even though some techniques might not be applicable to parallel programs, the concepts involved in these techniques are essential to the construction of the parallel debugging tool proposed in this work.

To finish the chapter, strategies to implement parallel debuggers are presented and discussed.

2.1 Debugging Background

Debugging is the practice of identifying, locating and repairing software faults [15]. A software fault is recognised when one or more unexpected actions occur during the execution of the software. According to Araki et al., experience in program development is important when performing debugging activities because these activities rely on "heuristic insights" [13]. Systematic debugging therefore appears to be an oxymoron. However, a number of studies on the cognitive aspects of programming characterise the debugging process as an iterative process of developing, verifying and refining *hypotheses*. These hypotheses can describe facets such as the potential causes of software faults, their locations and the requirements to fix such faults [16-18]. As a result, authors have developed formal models that establish generic steps in describing how the debugging process can be carried out systematically.

2.1.1 Araki's Debugging Model

Araki et al. proposed a debugging process model that concentrates on the creation and manipulation of *hypotheses* (Figure 2-1) [13]. Upon observing and confirming a program fault, a programmer develops an initial set of hypotheses. A hypothesis in this context may be relative to information such as the location of fault, the types and identity of fault, the expected program behaviour according to the program specification and the condition that may cause the fault [13]. Therefore, a hypothesis set also reflects the programmer's empirical knowledge of the software development process, the program and its specifications. The initial hypothesis set could be the error report that the programmer extracts from the latest run of the program. However, throughout the debugging process, this hypothesis set evolves as the programmer selects and verifies hypotheses. Some hypotheses will be proven during the debugging process; these are called *fact hypotheses*. The iterative process completes when the fault is identified and fixed.

Hypothesis Selection: several strategies are involved in selecting a hypothesis to be verified. A hypothesis may be selected because it simplifies the error condition. Another hypothesis may be selected because it narrows down the suspicious locations of the fault. In contrast, a hypothesis can also be chosen because it widens the error-free area. Hypotheses can also be weighted in order to identify the most significant hypothesis.

Hypothesis Verification: four verification techniques can be used. First, *static analysis* examines program properties such as structures and dependencies, cross-references, inter-module interfaces, type consistency and formal proofs. Second, *dynamic analysis* allows a program to be executed with appropriate inputs so that outputs can be examined systematically. *Semi-dynamic analysis* falls between static and dynamic analysis. Finally, *program modification* permits a program to be altered and re-executed to verify a hypothesis.

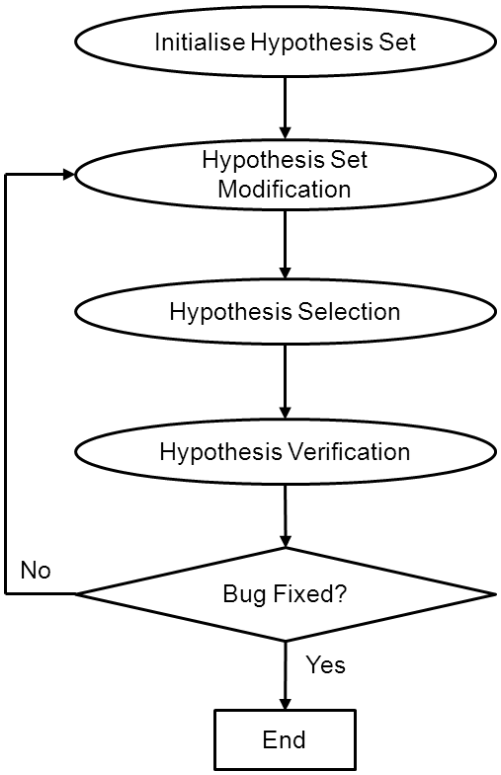


Figure 2-1: Araki's General Debugging Model

2.1.2 The Scientific Debugging Method

Zeller in his book "Why programs fail: A guide to systematic debugging" [14] describes a general debugging method called the *scientific debugging method*. To debug in a more systematic and reproducible fashion, the failing program can be approached as if it were a natural phenomenon. The general scientific process can then be applied to identify the cause of the phenomenon and lead to the required fix. Debugging activities in this model can be conducted following the flowchart illustrated in Figure 2-2. Of these, inventing, refining and alternating a hypothesis are possibly the most important activities.

Similar to Araki's model, the scientific debugging method starts with the formation of a hypothesis. A hypothesis is verified using observation from the execution of the program. The verification result is used as input for either altering the current hypothesis

or refining a new hypothesis. These activities can be repeated to narrow down the potential causes of the program's failure.

Where Araki's model uses a set of hypotheses, Zeller's model focuses on a single hypothesis. Zeller's model therefore does not have a hypothesis-selection phase. Rather, it concentrates on the verification of a hypothesis during the debugging process. In addition, while Araki indicates four different techniques to verify a hypothesis, Zeller's model is only concerned with the state of the program at runtime. This is because Araki's model aims to be a generic model for various debugging methods while the objective for Zeller's model is to follow the scientific method.

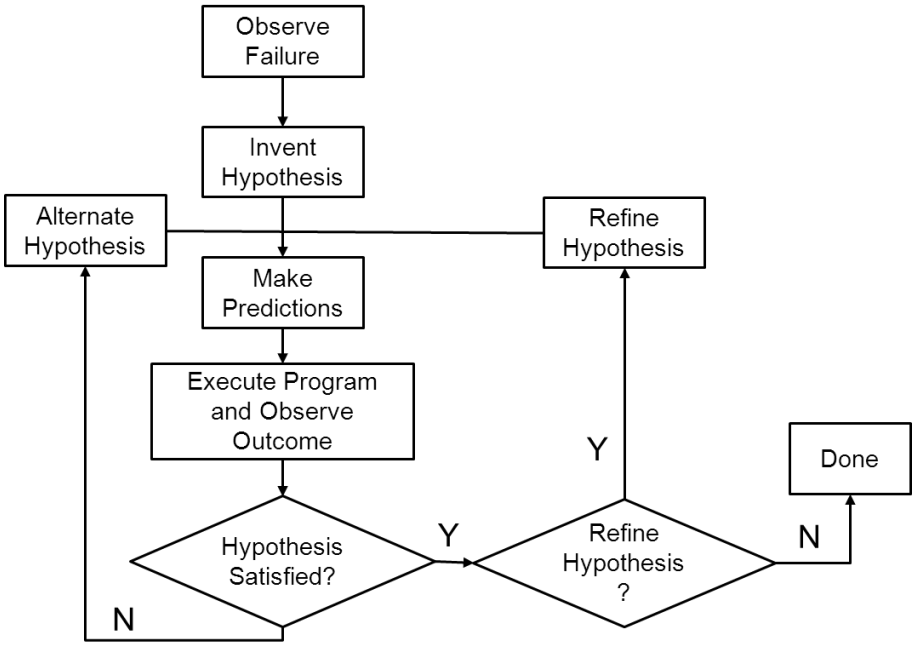


Figure 2-2: Flowchart Describing the Scientific Debugging Method

2.2 Traditional Sequential Debuggers

Traditional sequential debuggers, as the name suggests, deal with applications that have a single thread of control. The execution of the program can thus be easily monitored and altered to allow various types of debugging activities to take place. Since the 1970s, many sequential debugging tools have been developed for different computing platforms, operating systems and programming languages. Successful examples include GNU GDB [7, 8], Microsoft's DEBUG [19], Turbo Debugger [20], Data Display Debugger (DDD) [21], dbx [22] and idb [23]. These tools perform debugging activities ranging from capturing memory dumps to analysing log files. They allow users to step through

program executions, retrieve or alter memory locations and variables and verify predicates. These debugging activities can be categorised according to their underlying process such as *static analysis*, *event-based*, *interactive* or *post-mortem* debugging. Other debugging methods such as the *revert-execution* technique (also called *backward stepping* or *execution backtracking*) and the *pictorial-debugging* technique are also worth discussion. Importantly, while differences between these methods can be clearly seen, they all reflect the general debugging models discussed in the earlier section. A brief overview of these techniques is presented below.

2.2.1 Static Analysis

The *static analysis* method (also known as *anomaly detection* [24]) is often developed with compilers. When compiling the code, a compiler applies different types of analysis such as *control flow*, *data flow* and *inter-procedural flow* in order to identify unexpected behaviours of the program (for example, the unexpected transfer of control or references to non-initialised variables [25]). Static analysis explores and summarises all possibilities in order to provide general and structural information about a program. Further, static analysis permits backward analysis of control flow and data flow.

The static analysis method can be used under Araki's debugging model where hypotheses can be verified using the *static-analysis* technique, as outlined in Section 2.1.1. Fry [26] also highlights the importance of performing debugging at the code-editing stage instead of the execution stage. Nevertheless, this debugging approach may also find "anomalies" that cannot arise during execution because it is not always possible to determine if a given path in a program is executable [24].

Several tools that perform static analysis include the Emacs Menus text editor [26], compilers such as the GNU compiler collection [27] and Integrated Development Environment systems (IDE) such as InterLisp [28], Eclipse [29], Netbeans [30] and .NET platform [31].

2.2.2 Event-Based Debugging

In contrast to static analysis, an *event-based* debugger monitors the progress of a program and responds according to various predefined scenarios [32]. Users can instruct the debugger to respond to certain events and thereby avoid stopping the program at noncritical points. When the debugger determines that a desired event has occurred, it

returns control to the user or performs a predefined action. A specific type of action that is often associated with the event-based debugging method is the recording of the event-history log. This history log can then be compared to the predicted set of behaviours to isolate errors in the program. *Conditional breakpointing* is another mechanism that reflects the event-based debugging method because it suspends the execution of the program when a condition (for example, an event) holds true.

Users can specify the levels of granularity that an event-based debugger needs to monitor. Lower levels of granularity indicate more frequent monitoring and allow users to more accurately determine the location of an event when it occurs. However, a lower level of granularity can increase execution time.

The event-based debugging technique resembles both Araki's model and the scientific debugging model because it supports the selection and verification of hypotheses (in the form of events) during the debugging process. Examples include GNU GDB [7, 8] and Ariadne [33].

2.2.3 Interactive Debugging

Interactive debuggers provide users with the capability to interactively invoke, stop, continue and restart the program via features such as breakpoints [34]. This debugging technique is also known as the *break-and-examine* technique. The freedom in controlling the flow of the program's execution provides the ability to manually observe the content of various data structures and variables at runtime. Users then decide whether the data is as expected based on their precedent knowledge of the code and then identify programming defects in the current code base.

The interactive debugging technique is closer to Zeller's debugging model than Araki's model because users are equipped with functions that enable them to closely observe a hypothesis. This is the most popular type of debugging technique [34] and it has inspired the design of many well-known debuggers including GNU GDB, dbx [22], Sdb [35], VAX Debug [36] and CodeCenter [37].

Recent interactive debuggers not only permit users to dynamically modify runtime data in order to perform dissimilar test cases during the execution, but also to incorporate *graphical user interface* (GUI) for easier visualising and observing of different states of the program. Window-based and mouse-based debuggers with bitmapped displays provide a

better debugging experience than traditional command-driven debuggers [24]. Debuggers including dbx [22], Pi [38] and Saber C [24] can perform debugging commands via mouse and button clicks, display multiple source files at the same time and highlight current control locations. Program variables can also be controlled and observed more easily using such displays. These features are sufficient for developers and debuggers to perform white-box testing prior to the release of any software application. Some IDEs also incorporate interactive debuggers in order to provide users with full software-development packages. Examples include Eclipse, Netbeans and .NET platform.

2.2.4 Post-Mortem Debugging

Post-mortem debuggers allow an application to execute to completion and make use of the resulting trace or log file to enable specific scenarios to be repeated [25]. This permits users to focus on the examination of the program state in order to determine what went wrong in a particular setup with the aim of finding out the logical error in their implementation. The post-mortem technique can be categorised as *dynamic analysis* under Araki's model because users can specify different inputs and observe traces of outputs to determine the cause of software errors. An example debugger is Ariadne [33].

2.2.5 Revert-Execution Debugging

Revert execution (or also known as *execution backtracking* [4]) allows a developer to undo a certain number of operations while the program is executed. While it is similar to the interactive-debugging technique because it provides users with the ability to track the progress of the program, a revert execution is relatively harder to conduct. One method is to capture the complete history of a test run. The history tape is then replayed, thus enabling a user to backtrack to any step [39]. A drawback of this method is the loss of interactivity; another problem is that the user is unable to change the values of variables before executing forward again. Another method, *structured backtracking*, enables backtracking without having to save the whole execution history [40]. Instead, it only saves "...the latest value of each variable changed in a statement" [6]. Consequently, this technique only allows simple backtracking over a certain number of source lines and backtracking to the inside of a composite statement is not possible. Revert execution can also be implemented using *periodic checkpointing* of memory pages or file blocks that are modified during program execution [24]. Examples of debuggers that support revert

executions of a program include Exdams (an interactive debugging tool for Fortran) [39], InterLisp [28], Cornell Program Synthesizer [41], Igor [42], Cope [43] and Spyder [24].

2.2.6 Pictorial Debugging

An important aspect of the debugging process is the ability to see the errors or anomalies. Often printing runtime data is only useful if data is a scalar or a small array. Some debuggers present runtime data as graphs and pictures when the amount of data to display exceeds one screen. These graphs can present both data structures and control-flow data. A sample display generated by the VIPS debugger is shown in Figure 2-3. The graphs are updated dynamically at runtime to provide users with a comprehensive picture of the execution. Examples include VIPS [44, 45] and PROVIDE [46].

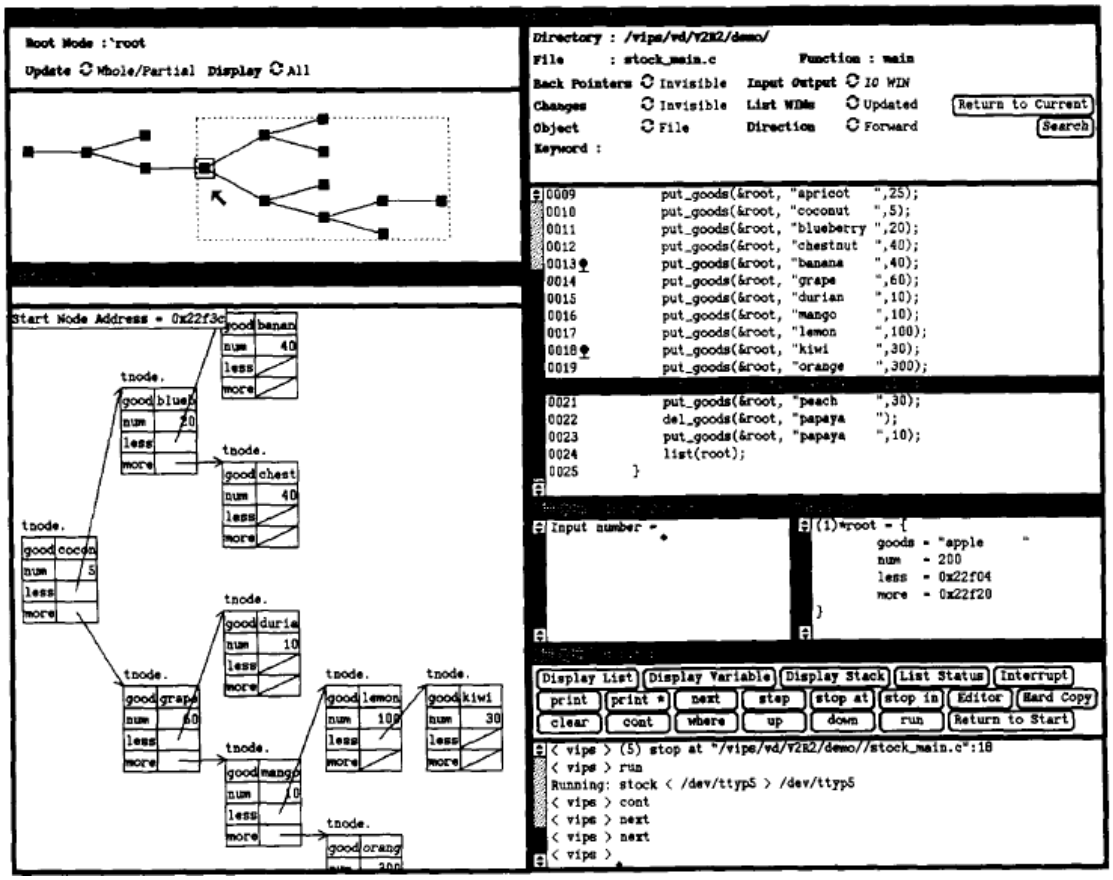


Figure 2-3: VIPS Visualises Program Execution from Various Aspects using Multiple Windows. Source [44]

2.3 An Overview of Parallel Computing

Chapter 1 gave a brief introduction to HPC and its applications in order to set a general context for the discussion in this thesis. Here, we take a step back and review the parallel computing field in general and then focus on various parallel programming models. This discussion is necessary to reveal the core challenges in debugging parallel applications. Because a high volume of studies and publications on both parallel computing and parallel programming models exist in the literature [47-52], this section consequently aims to establish a specific context for the discussion of parallel debugging techniques. It does this by highlighting the attributes of parallel programming that lead to challenges in debugging parallel programming code.

2.3.1 Parallel Computing

2.3.1.1 A Brief Definition

Traditionally, computers were designed to perform serial computation. Problems were broken into sets of discrete instructions and a *Central Processing Unit* (CPU) was used to execute each instruction one at a time [47]. The performance was measured by how fast the CPU could load and execute the instructions in a single unit of time.

In the simplest sense, a parallel computer or a parallel computing platform deploys multiple CPUs simultaneously to solve a computational problem. To achieve this, the problem needs to be *decomposed* into distinct parts that are conducted concurrently. This is the fundamental challenge when adopting parallel computing. For example, the process of making a supreme pizza comprises several tasks such as making the dough, preparing the ham and cheese, slicing the pineapple and other vegetable ingredients and baking the pizza. If there is only one cook, each of these tasks needs to be performed sequentially. However, given two or more cooks, the tasks can be performed in parallel, thus hastening the pizza-making process.

Although logically appealing, parallel computing faces issues. Not all problems can be finely broken into tasks that can be carried out in parallel. In the pizza-making example, although different cooks can perform different preparation tasks at the same time to speed up the process, the baking task has no parallel component, and so the time taken at this part of the process is likely to remain unchanged, thus limiting the overall improvement. This highlights the fact that parallelism also has its limitations and that the

performance of a parallel algorithm is thus restricted by Amdahl's law [53]: "...The nature of this overhead (in parallelism) appears to be sequential so that it is unlikely to be amenable to parallel processing techniques. Overhead alone would then place an upper limit on throughput of five to seven times the sequential processing rate, even if the housekeeping were done in a separate processor".

The process of breaking a problem into parts that can be executed in parallel (or in other words, the design of the parallel algorithm) is important when adopting parallel computing. Parallelism can be achieved with several methods. These are described in the following section.

2.3.1.2 Types of Parallelism

Bit-Level Parallelism

At the lowest level, word size dictates the number of instructions needed to perform operations on variables whose sizes are greater than the length of the word [49]. For example, adding two 16-bit integers using a an 8-bit CPU requires the CPU to first add the 8 *lower-order* bits from each integer using the standard addition instruction and then add the 8 *higher-order* bits using an *add-with-carry instruction* with the carry-bit from the lower order addition. As a result, completing a simple *add* operation executed on an 8-bit CPU requires two instructions, whereas the operation on a 16-bit CPU can be achieved with only one instruction. *Bit-level parallelism* means more bits can be processed in parallel and thus reduces the need for extra instructions [49].

Instruction-Level Parallelism

As explained earlier, a program is composed of a set of instructions executed sequentially by a CPU. Reordering these instructions may allow them to be combined into groups that are then executed in parallel without changing the final result of the program. This is referred to as *instruction-level parallelism* [50].

Instruction-level parallelism is implemented at the processor level using the technique called *multi-stage instruction pipelines*. A pipeline processor such as RISC [50] has five stages in executing an instruction: *instruction fetch*, *decode*, *execute*, *memory access* and *write back*. Pipelining allows the execution of a different instruction while awaiting the completion of the current instruction. In general, a processor with an N-stage pipeline can simultaneously have up to N different instructions at different stages of completion.

One important rule for grouping instructions together is to ensure no *data dependency* between them. The two most common techniques for implementing *out-of-order execution* and instruction-level parallelism are the Scoreboarding algorithm [54] and the Tomasulo algorithm [55] (which is different to Scoreboarding in making use of register renaming).

Data Parallelism

Data parallelism focuses on analysing program loops in order to allow distribution of data across computing units; a large data structure can thus be processed in parallel [49]. Often program loops display a set of operations that are similarly applied on different data elements. Hence, a single loop that iterates through N elements can be broken into M loops that can be conducted by M processing units. Each new loop iterates through N/M elements independently. For example, the following pseudo-code (Figure 2-4) performs a simple linear search given a large array.

```
target, buffer[N]    # N is a very large number
loop i=1 to N
  if target == buffer[i]
    exit loop
  i = i + 1
end loop
```

Figure 2-4: Pseudo-code for a Simple Linear Search Algorithm (no Data Parallelism)

Assuming `M` compute processes, this algorithm can be parallelised with data parallelism by decomposing the array `buffer` into `N/M` sub arrays; the `M` processes will then perform the linear search in parallel. The pseudo-code in Figure 2-5 illustrates the task performed by a single process.

```
target, buffer[N]    # N is a very large number
loop i=(proc_id-1)*(N/M)+1 to proc_id*(N/M)
  if target == buffer[i]
    exit loop
  i = i + 1
end loop
```

Figure 2-5: Pseudo-code for a Simple Linear Search Algorithm (with Data Parallelism)

Similar to instruction-level parallelism, data dependencies or *loop-carried dependencies* prevent the parallelisation of loops [49]. Nevertheless, data parallelism scales well with

problem size because the larger the dataset, the greater the number of processing units that can be invoked in parallel [49].

A recent trend in computer architecture is the use of Graphics Processing Units (GPUs) [56, 57] to achieve high level of parallelism. A "General Purpose GPU" (GPGPU) is designed using the Single Instruction Multiple Data (SIMD) architecture [57]. One of the programming models in the GPGPU paradigm is the *stream processing* model in which a series of *kernel functions* can be applied to each element in a dataset (called a *stream*). Many high-throughput type computations that exhibit data-parallelism including ray tracing, computational fluid dynamics and weather modelling can be computed in parallel using GPGPUs.

Task Parallelism

Task parallelism is recognised when "...entirely different calculations can be performed on either the same or different sets of data" [49]. The example above of parallelising the pizza-making process would fall into this type of parallelism because each cook (or individual processing thread) conducts a totally different task. In addition, different execution threads can communicate with each another in order to pass data from one to another as part of the overall algorithm. While data parallelism focuses on data volume for parallelising, task parallelism emphasizes the distributed (parallelised) nature of the processing (that is, the different compute threads). As a result, a bigger problem size does not usually mean that larger task parallelism is gained. In real life, most problems fall somewhere between task parallelism and data parallelism [49].

2.3.2 Parallel Programming Models

This subsection reviews the two most common parallel programming models: the *shared-memory* model and the *distributed-memory/message-passing* model. These models offer different features for implementing applications parallelised by domain decomposition (that is, task or data parallelism). For each of them, a brief description and application are provided, along with a focus on how the features of the model relate to the common types of coding bugs.

2.3.2.1 Shared Memory

In the shared-memory programming model, multiple compute threads read/write and manipulate a common memory space asynchronously [47] as depicted in Figure 2-6. A

number of synchronisation mechanisms such as *mutexes*, *condition variables*, *read–write locks*, *record locks* and *semaphores* are used to enforce access limits on a shared resource [58].

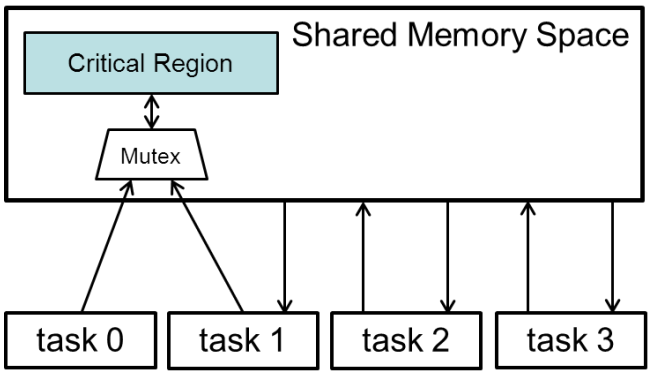


Figure 2-6: A General Shared-Memory Layout

Advantage

The shared-memory programming model is preferred when data *ownership* is lacking [47], because no explicit communication of data between tasks is required. This feature makes data sharing between tasks both fast and uniform and often simplifies the program-development process.

Disadvantages

Managing data locality in the shared-memory programming model is difficult and error-prone [47]. Scalability is also an issue because adding extra CPUs results in increased traffic between the shared memory and the CPUs.

Typical Coding Bugs

- *Race condition* refers to the situation that exists when several compute threads all try to access a shared resource [59]. A race results in non-deterministic behaviour of the program. This type of defect is often controlled and prevented by the use of synchronisation constructs such as mutexes (mutual exclusion) and semaphores. Four types of race conditions exist: *concurrent race*, *general race*, *unordered race* and *omission race* [59].
- *Deadlock* is a program state in which two or more compute threads are each waiting for the other to finish, and thus neither ever does [60]. This problem arises with the careless use of locks for synchronisation purposes. Four necessary and sufficient conditions for the occurrence of deadlock can be identified [60]:
    - a shared resource that must be used under mutual exclusion between multiple compute threads;

- a thread waiting for an additional resource to be granted while holding on to another resource;
- a program that does not have a resource pre-empting (forcible withdraw) mechanism;
- an existing circular chain of resource requests and resource allocations.

- *Livelock* is a situation that arises when two or more compute threads continuously change their states in response to changes in the other processes [61]. In consequence, neither of the threads can proceed further. Livelock occurs most frequently when attempts to prevent or avoid deadlock are performed incorrectly.

2.3.2.2 Distributed Memory / Message Passing

In the distributed-memory/message-passing programming model, while multiple tasks can either reside on the same physical platform or can be distributed across interconnected machines, each task has its own local memory space (Figure 2-7). Communication by sending and receiving messages is necessary to exchange data between tasks [47]. This is the core implementation feature of message-passing applications. Message passing is probably the most widely used parallel-programming model today [47] via the use of the MPI [62] or PVM [63] parallel-programming systems.

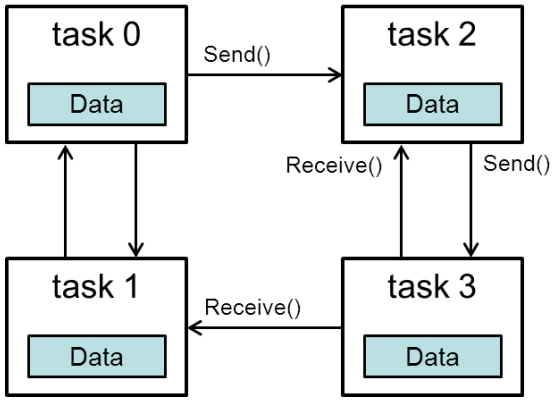


Figure 2-7: Simple Distributed-Memory/Message-Passing Model

Advantages

The core advantage of using the distributed-memory/message-passing model is the scalability gained. In term of space, because each computing unit has its own memory space, the size of memory increases proportionally as the number of processors increases. In terms of speed, accessing owned memory space can be performed by each processor rapidly without incurring overheads such as network traffic, and synchronisation constructs.

Disadvantages

The first disadvantage of using the distributed-memory/message-passing model is that it increases the developer's workload. Because message passing is the only mechanism available for interaction between computing tasks, the developer must take responsibility for managing the communication detail. Second, because data is owned and manipulated by distributed processors, reconstructing the global data structure is normally difficult. Although techniques exist for managing data-decomposition information [64], when it comes to load balancing or dynamic re-meshing, it is not clear how existing data structures can be mapped, based on global memory, to the structures required by a distributed-memory/message-passing model.

Typical Coding Bugs

- *Race condition* can occur in message-passing applications when:
  - individual processing threads try to access global resources such as IO streams; or
  - processing threads perform collective operations (for example, `MPI_Bcast` in the MPI programming language) without being aware of synchronisation issues.
- *Deadlock* can occur in message-passing applications because message passing is a mean, not only for communication, but also for synchronisation between independent processing threads. Poor management of communication routine calls could also cause a serious deadlocking issue. For example, in the following scenario (Figure 2-8), because both processes 0 and 1 try to "blocking send" data to each other, the program falls into deadlock while waiting for the `Receive` routines to be executed.

```
process 0                          process 1
Send(to_send, process 1)           Send(to_send, process 0)
Receive(to_receive, process1)      Receive(to_receive, process0)
```

Figure 2-8: Deadlock with Poor Send/Receive

2.3.3 Implementation of Parallel Programs

Several software packages support the implementation of parallel programs. Different packages provide different support according to (1) the programming model used

(shared memory or message passing); (2) system architecture; and (3) programming languages (for example, C/C++, Java or Fortran).

2.3.3.1 Message Passing Interface (MPI)

The Message Passing Interface (MPI) [62] is a standard, portable message-passing library developed to enable programmers to write portable message-passing applications. It targets distributed-memory systems. It falls into the message-passing programming model. Programs developed with MPI communicate internal tasks solely by passing messages between themselves.

MPI is probably the most popular parallel-programming interface. It provides a rich message-passing application programming interface (API) and supports logical communication topologies. MPI has a number of implementations such as OpenMPI [65], MPICH1 [66], MPICH2 [67] and LAM/MPI [68]. In addition, MPI supports popular programming languages such as C/C++, Java and Fortran.

2.3.3.2 Parallel Virtual Machine (PVM)

The Parallel Virtual Machine (PVM) [63] is a software package that enables the use of interconnected computers for parallel-program execution. Similar to MPI, PVM provides functions to support simple message passing among the processes for coordination. Unlike MPI, PVM supports the virtual-machine concept where users can create a number of programs or processes to run concurrently on same machine.

2.3.3.3 Open Multi-Processing (OpenMP)

Open Multi-Processing (OpenMP) [69] is an API that enables mutli-processing programming on shared-memory platforms. In contrast to MPI and PVM, OpenMP is an implementation of the multithread technique. Threads can be allocated to different CPUs in the parallel system in order to achieve high parallelism. OpenMP also supports popular programming languages including C/C++ and Fortran.

2.3.2.4 Unified Parallel C (UPC)

Unified Parallel C (UPC) [70] is the result of leveraging the traditional C programming language to support HPC on parallel machines. A UPC program is provided with a single shared-address space, where variables may be directly read and written by any processor. However, each variable is only physically associated with a single processor. This model is therefore similar to that used by the message-passing style of programming (MPI or

PVM). UPC also allocates a single thread to a single processor at start-up time. UPC combines the programming flexibility of the shared-memory paradigm with the control over data layout and the performance of the message-passing programming paradigm [70].

2.3.3.5 Global Arrays Toolkit (GA)

The Global Arrays (GA) toolkit [71] is a software package developed to support parallel computing for shared-memory programming on distributed-memory computers. Apart from the core GA library (which provides simple matrix computations such as matrix-matrix multiplication or LU solve), the GA toolkit has additional libraries such as the Memory Allocator (MA) and the Aggregate Remote-Memory Copy Interface (ARMCI).

In contrast to other shared-memory software packages, the GA model exposes the locality information for the shared data and provides a mechanism to directly access the local portions of shared data [71]. The GA library is also compatible with MPI.

2.4 Challenges in Debugging Parallel Programs

As highlighted in Section 2.3, although parallel models show promise in terms of supporting large programs, parallel models are hard to design and implement for several reasons.

- Processes execute in parallel while sharing system resources such as memory space or network bandwidth. Extra effort is then required to manage and control the resources.
- A process may be dependent on other processes in the program. For example, one process may only be able to update a memory space after another process releases it.
- Because each processing unit has its own clock, perfect synchronisation between processes becomes infeasible. This leads to problems in determining the global order of execution. In this situation, the state of the global program is sometimes non-deterministic.

These problems not only affect the process of developing parallel programs but also translate into nontrivial challenges when debugging parallel code. We examine several major challenges below before we discuss parallel debugging technology in Section 2.5.

2.4.1 Non-Deterministic Behaviour

A parallel program often generates non-deterministic behaviours for one of the following reasons. First, it could be the lack of synchronisation. Parallel processes may run on different CPUs that have different loads from time to time. The speed of the executing processes or threads in the distributed application will therefore also vary. Second, communication between processes is dependent on the performance of the network itself, which will also affect the overall behaviour of the program. Last but not least, shared resources (such as memory space) might be badly managed which may also cause differences in interactions of processes or threads in the application.

Using Araki's general debugging model or the scientific debugging method, debugging a program that produces non-deterministic states, given the same input set, is complicated and confusing. This is because a hypothesis cannot be firmly verified. Non-determinism is possibly the hardest problem to manage when debugging parallel programs.

2.4.2 The Probe Effect

Debugging involves the use of extra resources to interfere with the execution of a program in order to verify certain hypotheses. When dealing with parallel/distributed programs, such interference may either cause new bugs to appear or make it impossible to observe previously discovered bugs. This phenomenon is called the *probe effect*. It normally occurs when a parallel debugger masks a non-deterministic behaviour of a concurrent program while trying to synchronously control the parallel processes. It can also occur when extra statements are inserted into the program in order to gain more insight into an observed bug, but may lead to the disappearance of the bug itself.

The probe effect is difficult to manage. Many well-known parallel debuggers suffer from this issue (discussed further in Section 2.5).

2.4.3 Data Decomposition

Parallelisation leads to the decomposition of either multiple tasks or large data structures, or both, as indicated earlier. Under Zeller's debugging model, this decomposition process implies the decomposition of a potential hypothesis if the hypothesis requires the global state of the program in order to be verified. Because the verification process of a hypothesis in this debugging model is centralised, the decomposition needs to be *undone*.

Furthermore, because Zeller's hypothesis is built and updated based on program observation, a global view of the program state is necessary. *Data decomposition* is therefore a challenge in debugging parallel programs. A good parallel debugger would have to provide its users with the means to reconstruct the global state of the parallel program so that the hypothesis about the program can be verified and updated.

Another reason to undo the decomposition performed on a global data structure is to provide users with the correct display of the array. This is not so important if the number of processes is small. However, as the number of processes increases to thousands, printing sub-data structures in order of process ranks prevents users from noticing the data-transformation problems.

2.5 Current Parallel Debugging Trends

While debugging sequential programs is relatively straightforward because a debugger has only one thread of computation to control, observe and verify, debugging parallel programs often poses more difficulties because there is more than one simultaneously executing process to manage. Often these processes have to operate asynchronously in order to take the advantage of the parallel architecture. This leads to phenomena such as non-deterministic behaviours and the probe effect as described above [34, 72]. Furthermore, bringing all compute threads under the control of one debugger can pose significant engineering issues [72]. Finally, because the target data structure can be decomposed and distributed across the parallel-processing threads, retrieving the synchronous overall data structure to produce a global view of the program's state is challenging [34]. These issues call for innovative debugging approaches that not only rely on and leverage fundamental sequential-debugging methods but that also directly address the primary differences between a sequential program and a parallel program.

In their work in 1989, McDowell and Helmbold [34] categorised techniques for debugging concurrent systems into four groups: *traditional debugging*, *event-based debugging*, *static analysis* and *techniques for displaying*. These techniques evolved from their sequential ancestors (discussed in Section 2.2). The following sections review each technique and present some example parallel debuggers.

2.5.1 Traditional Debugging

2.5.1.1 Description

Traditional debugging is also known as *breakpoint debugging*. It employs the interactive debugging model described above to allow a user to start, stop and resume the execution flow of the program at given breakpoints. The framework of this technique contains a set of sequential debuggers in which each debugger controls one computational process. Upon reaching a breakpoint, a programmer steps through program execution to observe the flow of control. Runtime data can also be retrieved from each debugger in order to observe the current state of the parallel program.

2.5.1.2 Drawbacks

Traditional debugging suffers from the probe effect and the non-deterministic nature of parallel programs because inserting a breakpoint explicitly indicates synchronisation of computational threads at a certain location in the program. It also raises a number of issues including how the output from multiple debuggers can be displayed in a coherent fashion, and how the multiple debuggers should be coordinated and controlled. These problems are exacerbated when the applications scale to tens or hundreds of thousands of processing cores. Parallel debugging tools of this type often try to accomplish scalability by improving various aspects including (1) output representation and data visualisation; (2) process management (for example, by using the notion of process sets); and (3) communication management.

2.5.1.3 Examples

Many current parallel debuggers can be categorised as traditional debuggers. Successful examples include:

- TotalView [73]: known for its user-friendly GUI. TotalView supports a range of programming languages include C/C++ and Fortran and various parallel-programming models such as MPI [62], OpenMP [69], UPC [70], Global Arrays [71] and traditional threads.
- Allinea Distributed Debugging Tool (DDT) [74]: similar to TotalView in many aspects such as the easy-to-use GUI and its support for various computing platforms and programming languages. DDT also provides sophisticated memory-manipulating functions to manage memory-leak defects.

- p2d2 [75]: the Portable Parallel/Distributed Debugger, p2d2, aims to provide a unified user interface for debugging on all computing platforms. It supports message-passing systems such as PVM [63] and MPI.
- LadeBug [76]: a simple command-line interface parallel debugger that supports programming languages such as C/C++ or Fortran and provides limited support for Cobol and Ada. Users can perform functions such as set breakpoints, monitor the execution of processes or peek at call stacks.
- Intel® Debugger (*idb*) [77]: enables developers to debug parallel programs that run on Intel machines. idb provides a scalable solution by setting up a tree-like debugger network and using a message-aggregation mechanism to reduce the amounts of data transferred across the network.
- Eclipse Parallel Toolkit Platform (PTP) [78]: provides a full software-development platform for developing and debugging parallel programs.
- IBM Debuggers (PDBX) [79]: is a successor of the dbx sequential debugger [22]. PDBX extends the line-oriented interface and enhances the subcommands to work on parallel programs.

2.5.2 Event-Based Debugging

2.5.2.1 Description

Drawn from the event-based technique addressed earlier, the event-based debugging method views the execution of a parallel program as a sequence of events. This method allows a user to browse the history, replay the execution or simulate the environment in order to debug the execution of individual processes. On a parallel program, events may also include messages or inter-process communication. Event-based debuggers might also link a special library with the target program or might insert statements into the source code.

2.5.2.2 Drawbacks

The event-based technique does not resolve the main problems (that is, the probe effect and the issue of non-determinism) associated with debugging concurrent software. The parallel computation cannot therefore always be exactly reproduced for debugging. In addition, when it comes to highly parallel applications, several serious drawbacks can be identified. First, replaying the execution of the parallel programs requires all processes to participate, even when some of them are clearly not involved with the bug. Second, the

user cannot stop the process being debugged in order to interact with the processes that are performing the suspicious activities. Finally, this technique inherits the performance pitfall of the sequential event-based technique where high level of granularity indicates longer execution time.

2.5.2.3 Examples

Event-based debuggers include:

- TotalView [80]: apart from being an interactive debugger, TotalView provides users with *ActionPoints* that are typically created on a memory location to watch its contents. TotalView generates an event when the contents of the memory location are changed. When the debugger receives such an event, it evaluates an expression (the condition) and reports to the user only if a change occurs in the value of the expression.
- Ariadne [33]: has three notable features. First, it provides feedback on failures. Secondly, it offers scalability through its patterns and non-graphical output. Finally, it supports simple language to construct behavioural abstractions using patterns of events in logical-time. A set of small finite-state recognisers is used to detect these patterns in traces of program behaviour.
- Prims [81]: is part of the Prism programming environment developed by Sun Microsystem Inc. Prims allows a user to associate actions to runtime events that can be controlled using traditional breakpoints. Events can be applied on subsets of all processing threads using the *Pset* qualifiers. A user can also create events using the *Event Table*.
- DETOP [82]: uses an event-based approach to support debugging of applications containing both functional and data parallelism.

2.5.3 Static Analysis

2.5.3.1 Description

The static-analysis method can be used to detect different types of programming defects, including synchronisation errors such as deadlock, and data-usage errors such as the simultaneous access of shared variables. According to McDowell and Helmbold [34] static-analysis debuggers cover two distinct areas: (1) supporting dataflow analysis techniques to parallel programs; and (2) determining whether two statements in a

program can be executed in parallel [34]. This latter area applies to debuggers that perform a data-dependency analysis on the target program to determine the location of schedule-dependencies [83]. With the static-analysis method, programs are never executed; consequently debuggers are not subject to the probe effect.

2.5.3.2 Drawbacks

Quite often, the dependency analysis is an undecidable problem [84]; thus this technique can only produce conservative approximations. In other words, even though the technique will guarantee to report dependence for a possible data race, it also reports many other potential data races that cannot actually occur. In addition, the main class of defects that can be isolated by using this technique is *program runtime failure*. The static-analysis technique obviously cannot be used to identify bugs that do not abort the operation of the program but instead silently corrupt the final results. Finally, the computational complexity for this technique is exponential, making it difficult to scale this technique to larger runs.

2.5.3.3 Example

- PTOOL [83]: this debugger analyses asynchronous parallel loops in concurrent programs. PTOOL is mainly built on two components: PSERVE and PQUERY. Loops are analysed by PSERVE to discover possible schedule dependencies while PQUERY is use to browse the information collected by PSERVE. PTOOL allows users to access individual dependence information in close detail and to filter options to restrict attention to specific classes of dependencies.

2.5.4 Techniques for Displaying Debugging Information

The origin of the display techniques for parallel debugging lies in the pictorial technique used for sequential debugging. As discussed, control-flow data and runtime data-structure values can be presented as boxes and arrows in order to help users visualise the progress of the program. However, while presenting runtime data is trivial for sequential debugging where only one thread of computation exists and only a small amount of data needs to be viewed, the issues become more complex in a parallel debugging situation. Nonetheless, displaying data is crucial for parallel debugging where both the runtime data and the communication and interaction between concurrent processes are of interest.

McDowell et al. [34] categorises four basic techniques for displaying debugging information:

- ***Textual presentation*** of the data or control-flow information is the simplest way to provide users with the status of the program during debugging. All debuggers incorporate this technique in various forms. However, it is only useful for simple testing tasks. Displaying runtime data structures of large-scale scientific code is impractical during the debugging process.
- ***Time-process diagrams*** can be used to present the execution of the program at runtime. Using a two-dimensional display, the activities of individual processes can be plotted against time. This is a neat way, not only to monitor the progress of each compute process independently, but also to observe the interaction between processes. However, as the number of processes multiplies and as their execution time grows shorter, the time-process diagram becomes too dense to provide helpful debugging information.
- ***Animation*** of program execution using a display corresponds to a single instant in time, or a snapshot, of the program states. By displaying these snapshots in a timely manner, the real-time progress of the application can be animated. A single frame can be formatted in different ways for an animation. Yet animation of a program execution does not clearly show the patterns of behaviour that occur across time. In addition, similar to the previous techniques, animation suffers from poor scalability.
- ***Multiple windows*** display several simultaneous views of the program being debugged. This frequently involves using one window per process. Similarly, this method does not scale well on large machines running large amounts of concurrent code.

To complement techniques involving the textual presentation of data but which are unable to display large datasets, scientific visualisation tools [4] may be used. These tools are used to find programming errors in scientific applications by rendering runtime data as 3D graphics or by supporting visualisation of particle trajectories. Visualisation software applications include open-source packages such as DataPlot [85], ParaView [5], ParaViz [86], ScienceGL [87] and VTK [88], or commercial packages such as NovoSpark Visualizer [4] and BitPlane [89].

While techniques involving data displays are useful for finding anomalies in large datasets, in general, it is not always possible to see the errors in the data. In addition, the

rendering process often requires the program to execute until completion; thus, this method can also be treated as a post-mortem technique.

2.5.4.1 Examples

- Traveler [90]: displays a sequential list of processes that access a shared-memory object in causal order instead of temporal order.
- Idd [91]: produces time-process diagrams in which messages between processes are plotted.
- Belvedere [92]: allows users to specify the placement of processes. This system animates *events* such as a processor, a channel or a data item. An event can also be specified by the user.
- DDT [74]: includes GUI to manage multiple processes in one window. It also visualises slices of multi-dimensional arrays using OpenGL graphics.
- TotalView [73]: graphically allows users to observe the way in which memory is being located during the run of the program. This graphical tool aids users to identify memory-leak bugs.
- Prism [81]: supports advanced visualisation including the ability to define the geometry of decomposed arrays, to visualise MPI's internal message queues and to display linked data structures. Further, a tool from the Prism software package called PrismModelChecker [93] allows users to explore a model by generating sample execution through a Prism model. This is particularly useful for debugging models during development and for running sanity checks on completed models.
- POET (Partial-Order Event Tracer) [94]: is a debugging tool for distributed programs. POET analyses events that occur during the execution of a program using partial-ordered rules. It also provides a visualisation tool for monitoring events between different components of the distributed system.

2.5.4.2 Drawbacks

The drawbacks associated with display techniques have already been addressed. The most significant problem with display techniques when debugging parallel code is the poor scalability offered by the display techniques. Nowadays, the most powerful technique available for integrating large amounts of data into a human-visible form is to use scientific visualisation tools [4]. This approach has been used to find programming

errors in scientific applications, but despite the visualisation, errors in the data may not always be found.

2.6 The Data-centric Debugging Approach

We have reviewed a number of parallel debugging techniques and provided a range of examples to illustrate the current practice in parallel debugging. These techniques, while presenting innovative ideas to support parallel programs, inherit characteristics from their traditional sequential-debugging counterparts and formal debugging models such as Araki's model and the scientific debugging model. In these models, control-flow and runtime data comprise the fundamental information available for constructing hypotheses.

The traditional techniques described above often use both control-flow and runtime data to guide the debugging process. For example, the break-and-examine technique places breakpoints at strategic locations in the program, thus allowing a user to retrieve intermediate values of important variables and to indicate the flow of control. In this way, users gain visual feedback about the internal workings of the program to help them localise a fault. However, this technique is only effective if there is (1) only a small number of computational threads to monitor, and (2) only a small amount of data to be displayed.

A number of researchers have discussed the need for a more data-centric view of debugging [9-11]. Here, we review existing data-centric techniques that have been deployed, not only in debugging software, but also in the formal verification of models. The aim of this section is to present a number of the more innovative ideas associated with the data-centric debugging concept. Limitations of these techniques are discussed in order to identify gaps in contemporary research. The technique *relative debugging* is explored in greater detail as it is the origin of much of the work in this thesis.

2.6.1 Automated Debugging

The area of automated debugging has been explored since the late 1980s [10, 24, 95-101]. A number of debugging systems have been developed based on three core underlying strategies: *"verification with respect to specification"*, *"checking with respect to language knowledge"* and *"filtering with respect to symptoms"* [102]. Each strategy employs different aspects of

either the programming models or the system specifications to create general abstractions used to automate the debugging process.

The "verification with respect to specification" strategy (or verification strategy) compares the formal specification of the intended program with the actual program. In contrast, the "checking with respect to language knowledge" strategy examines the implementation of the program and searches for suspect places that do not follow certain specifications of the programming language. The *"filtering with respect to symptoms"* strategy explores the assumption that correct parts of the code cannot be the cause of the error symptom. Of the three strategies, the verification strategy aims at fully automating the debugging process and is the focus of further discussion in this thesis.

Within the verification strategy, by using formal specifications from the intended program, segments of the actual program can be verified iteratively and any violation occurrence can be translated directly as a potential program defect. This strategy reflects the data-centric debugging paradigm. Here, data can be viewed as specifications and the data can be used to reason against certain expected behaviors.

Several attempts have been made both deploying the verification strategy and using assertion as a tool for debugging sequential programs. Failley [96] describes a debugger called ALADIN that supports the debugging and testing of assembly programs. Instead of using the traditional breakpoint location, ALADIN introduces the *breakpoint assertion* that breaks the running program and returns control to the user when a particular assertion is violated. Using ALADIN, the developer or tester can therefore make use of the nature of the error rather than concentrate on locations where the error might have occurred. Later, Drabent et al. [10] describe the use of assertions in the *algorithmic-debugging* technique. Using this technique, the debugging system acquires knowledge about the expected meaning of a program and uses it to localise errors. The authors suggest using formal specifications for some properties of the intended model to construct the assertions. Similarly, Puebla et al. [11] discuss another effort using assertions in debugging *constraint-logic* programs. This technique allows general properties of a constraint-logic program to be defined as assertion schemas and aims to perceive the program's behavioral variation relating to those assertions either at compile-time or runtime. Furthermore, Auguston et al. [9] present an application framework for the development of automatic-debugging tools, using the FORMAN assertion language

[103] as its foundation. The framework implements functions such as assertion checking, profiling and other debugging queries in a uniform way.

Even though the verification strategy can theoretically locate and identify many different types of errors, it has been limited to debugging logical or constraint-logic programs. The reason is because generating accurate specifications for the intended behaviours is a relatively complex task. As a result, for sequential debugging, the *interactive-debugging* technique is preferred.

Arriving at the HPC era, recent studies have focused on constructing and evaluating assertions in parallel programs. Siegel et al. [104] introduce a novel type of assertion called a *collective assertion* that addresses the importance of evaluating assertions in parallel programs. Collective assertion asserts the global state of the parallel program by taking snapshots of the state of processes and delivering them to the Toolkit for Accurate Scientific Software (TASS). TASS uses symbolic execution and explicit state-enumeration techniques to evaluate the assertion, given a set of local-state snapshots. Nevertheless, because TASS executes sequentially, a bottleneck can still be formed when both the size of the problem and the machine's size increases. The authors only present performance evaluation for up to 16 processes [104]. Chi-Neng Wen et al. [105] and Daniel Schwartz-Narbonne et al. [106] in separate efforts describe a programming environment in which assertions can be declared and evaluated as a parallel program is executed. On the one hand, Wen et al. develop a non-intrusive runtime assertion (RunAssert) which allows developers to detect potential race condition issues and verify acceptable sequence of instructions [105]. On the other hand, Schwartz-Narbonne et al. introduce the semantics and present a prototype implementation (PAssert) that support the declaration and execution of program assertions in parallel [106]. While RunAssert and PAssert are similar to the approach proposed in this thesis, they are designed for debugging programs developed under the shared-memory parallel programming model, instead of under the distributed-memory programming model. Therefore, they do not address issues such as the distribution of program state over many processors. In addition, their performance on debugging large scale programs is not presented in detail thus they might not work well when the target application is at petascale. For instance, the overhead for supporting PAssert in Schwartz-Narbonne et al. system could slow down the execution of the program up to 20 times [106].

In related work, a plug-in is developed to assist CharmDebug [107], a specific debugger for the Charm++ parallel-programming language, in terms of runtime data introspection. The plug-in allows the user to upload specific Python scripts to the runtime parallel applications (known as servers) and to execute them in order to perform data checking for debugging purposes. However, because no data-decomposition information is attached with the Python script, it is not clear how the actions can be applied to the overall data structure that is distributed across many processing units. Likewise, TotalView [73] supports debugging scripts, in a language called *TVScript*, that are executed at breakpoints. TVScript supports actions such as displaying variables, displaying a back trace, showing the memory allocation, evaluating potential memory leaks and generating logs for post-mortem analysis. However, it is difficult to reason about the collective state of an entire program because scripts only access individual processes. It is thus difficult to implement an assertion such as "the sum of all elements of array A to be less than X" without building multi-process aggregation code.

In a similar vein, GNU GDB [7, 8] allows developers to specify conditional breakpoints. A conditional breakpoint is a simple breakpoint that is triggered when the attached condition fails at the specific location in the source code. Using conditional breakpoints, users can assert the state of variables at runtime. In addition, GDB enables user-defined functions that can be used to calculate, test and print data variables accessible from GDB. The combination of conditional breakpoints and user-defined functions is a promising tool for monitoring the state of the application efficiently. Nevertheless, GDB is a sequential debugger and does not support the debugging of parallel programs. Thus, it is not possible to write conditional breakpoints that reason across the state of multiple processes. Moreover, because GDB has no way of describing a globally distributed data structure, it is not possible to write an assertion about such a structure.

2.6.2 Statistical Invariants

The idea of using statistical methods to deduce the status of an application and detect program defects has received attention. Zhou et al. [108] mention a *statistics-rule-based* method to extract statistical rules as the program executes. The statistics-rule-based debugging approach is a newly explored direction of the verification strategy discussed earlier. More specifically, this technique extracts *statistical rules* (for example, statistical invariants) from the records collected after numerous successful executions or multiple periods of a single long-running execution. These rules are later used for detecting

violations in a later execution (or later in the same long-running execution). As a successor to the verification strategy, this technique promises to catch bugs that may satisfy all programming rules but produce non-deterministic behaviours. Zhou et al. [108] also introduce a debugging system that can detect, through observing how memory locations are accessed, illegal memory accesses by outlier instructions. This type of behaviour is often caused by memory corruption, buffer overflow or other memory-related bugs. The anomalies can be outlined using a statistic-based method called *"program counter (PC)-based invariance"*. Similarly, Daikon [109] focuses on techniques that allow program invariants to be discovered dynamically, while DIDUCE [110] supports the formulation of hypotheses by instrumenting the program and observing its behaviours at runtime. These methods demonstrate that statistics-rule-based approaches are promising in terms of detecting bugs that do not violate any programming rules; however, they only work with sequential programs and are not useful in parallel situations.

In the realm of distributed systems, WiDS Checker and D3S [111, 112] allow developers to specify predicates on distributed properties of a deployed system and to check these predicates while the system is running. When D3S finds a problem, it produces the sequence of state changes that led to the problem, allowing developers to quickly find the root cause. Importantly, because developers write predicates in a simple and sequential programming style, D3S can check these predicates in a distributed and parallel manner; this function allows the tool to scale to large systems. By using binary instrumentation, D3S works transparently with legacy systems and can change the predicates at runtime.

Another tool called DMTracker [113] also employs the statistics-rule-based technique and provides a solution for parallel applications. The tool can automatically detect the cause of phenomena such as data corruption or deadlocks by observing data movements between parallel-processing threads.

One application that makes use of *offline debugging* (or post-mortem debugging) is AutomaDeD [114]. AutomaDeD is a debugging framework that locates where a program bug occurs in relation to the parallel tasks, the code region and the program-execution point. To this end, AutomaDeD performs runtime monitoring of the application. The runtime data can be used to statistically model the application's control-flow and timing behaviour using Semi-Markov Models (SMMs). AutomaDeD then groups the parallel-processing tasks and identifies divergence from the intended execution.

Another method of applying statistics for debugging is the *statistical debugging* technique developed by Liblit et al. [115]. This technique was proposed in order to isolate bugs that correlate with program runtime abnormal failures, such as memory corruption or segmentation faults. The author argues that stochastic failures can be reported multiple times and that information extracted from reporting data via various statistical and modelling techniques can be used to deduce the likely location of the bugs. However, the goal of the statistical-debugging technique is only to isolate a certain class of bug: *program runtime failure*. It cannot be used to identify bugs that do not abort the operation of the program but rather silently corrupt the final results.

2.6.3 Relative Debugging

The *relative debugging* technique helps a programmer locate errors in code by observing the divergence in key data structures between two versions of the same program as they are executing [116, 117]. In particular, it allows comparison of a *suspect* program with a *reference* program using assertions. It is a particularly valuable technique when a program is ported to, or rewritten for, another computing platform. Relative debugging is effective because the user can concentrate on where two related codes are producing different results, rather than being concerned with the actual values in the data structures. Various case studies reporting the results of using relative debugging have been published [118-120] and these have demonstrated the efficiency of the technique.

The concept of relative debugging is both language and machine independent. It allows a user to compare data structures without concern for the implementation and thus attention can be focused on the cause of the errors rather than on the implementation details. A relative debugger uses a *comparative assertion*, which consists of a combination of data-structure names, process identifiers and breakpoint locations. Assertions are processed by the debugger before program execution commences. The debugger also builds an internal graph that describes when the two programs must pause and which data structures must be compared. In the following example (Figure 2-9) the debugger compares data from `big_var` in `$a` at line 4300 of `ref.c` source file with `large_var` in `$b` at line 4400 of `sus.c` source file.

```
assert $a::big_var@“ref.c”:4300=$b::large_var@“sus.c”:4300
```

Figure 2-9: A Sample Comparative Assertion in Guard

A user can formulate as many assertions as necessary and can refine them after the programs have begun execution. This feature makes it possible to locate an error by placing new assertions iteratively until the region is small enough to inspect manually. This process is efficient. Even if the programs are millions of lines of code, because the debugging process refines the suspect region recursively with each iteration, it does not take much iteration to reduce it to one screen of code.

Other features have also been introduced to relative-debugging applications. One relative debugger, *Guard*, incorporates key innovations such as a multi-threaded data-flow engine, a data-transformation algebra, an architecture-independent data representation (AIF) and support for parallel and distributed computing [25]. When performing comparisons, errors might be incorrectly attributed to differences in the precision of the program variables or other minor numeric factors. To avoid this, Guard lets the user specify a tolerance threshold. Variables are considered equivalent when the result of a comparison is within this threshold. Importantly, Guard supports both sequential and parallel relative debugging, and has novel features for describing the data decomposition in parallel code [25]. It supports a range of conventional programming languages (for example, C, C++ and Fortran) and also a data parallel research language called ZPL [119].

As users migrate existing parallel (and sequential) applications to petascale machines, programs may fail even if many of the machine characteristics appear to be the same. Different versions of the compilers, link libraries and operating systems can cause programs to produce different results. Different processor architectures can often cause subtle errors that are difficult to trace. Most importantly, programs sometimes generate incorrect results when the number of processors is increased; these can be difficult to diagnose. Relative debugging has application in all of these cases.

Relative debugging suffers several drawbacks. First, it requires an existing version of the same program for referencing purpose. Second, the comparison phase required data from both programs to be collected and combined. This leads to memory capacity issues when large amount of data is involved. Furthermore, the implementation of existing relative debuggers such as Guard [25] or p2d2 [121] suffers serious performance issue because the data combination and comparison tasks are conducted sequentially.

The development of the prototype debugger described in this thesis is based on Guard. Hence, Chapter 3 and Chapter 4 will revisit the relative-debugging framework and Guard to present a more thorough discussion and evaluation of the technique.

2.6.4 Why Do We Need More Research?

The advantages and disadvantages of various data-centric debugging techniques and tools have been discussed in the above subsections. While they are successful in shifting the focus in debugging more towards the runtime data and supporting automation of the debugging process, they suffer from the following drawbacks. First, even though the cognitive challenge has been addressed by several studies above, they still fail to support the exploration of data, especially when reasoning about the computation as a whole. This is because they lack the support for reconstruction of decomposed data at runtime. Second, while tools like Collective Assertion and Guard provide users a way to make statements about the overall state of the program, they do not scale to work with large problems running on large machine. This is because these tools perform primary debugging operations with a single frontend node. In general, most parallel debuggers do not utilise the underlying parallel machine to conduct more computational expensive debugging tasks. Most parallel debuggers, while attempt to debug parallel programs, do not perform their tasks in parallel.

This research aims to improve a data-centric debugging technique so that large runtime data from large parallel programs can be processed more effectively for debugging purposes. This could be achieved by exploiting the available computational power of the parallel machine. To conclude, the key contribution of this research is through providing debugging primitives for reasoning of global program state using advanced data mining operations while delivering sustainable performance.

2.7 Parallel Debugger Implementation

Here we discuss the trends in design and implementation of parallel debuggers. To cope with the large number of computational processes that run concurrently under a high-performance network, new architectural ideas have emerged and gained preliminary success.

2.7.1 Client/Server Architecture

The most popular architectural design is the client/server architecture. This architecture comprises three distinct layers (Figure 2-10). First, the *client* layer hosts a single, centralised, frontend process. The second layer is the *network-infrastructure* layer. The third

layer consists of many independent backend debugging processes called *debug servers*. Network infrastructure supports communication protocol (for example, basic socket connections or communication networks with different topologies). It provides the medium for transferring requests from the frontend client to the backend debug servers and for collecting debug data from those servers to the client processes. The frontend process acts as the user interface while the third layer controls and coordinates the backend debug servers through the network infrastructure. The debug servers may be relatively simple, and perform the low-level control operations on behalf of the frontend client, or they may be full debuggers such as GNU GDB [7, 8]. Each of these attaches to a single processing thread of the parallel program and directly controls it independently. Many parallel debugging tools adopt this architecture, even though they might vary in the implementation of the debug server. For example, DCDB [122], PDBX [79] and XPDBX [79] use dbx [22], whereas DDT [74], DDBG [123] and Guard [25] use GDB as their debug engine. TotalView [73] and Mantis [124] have their own proprietary code, while p2d2 [75] leaves the implementation of the server to the platform's vendor in order to achieve portability.

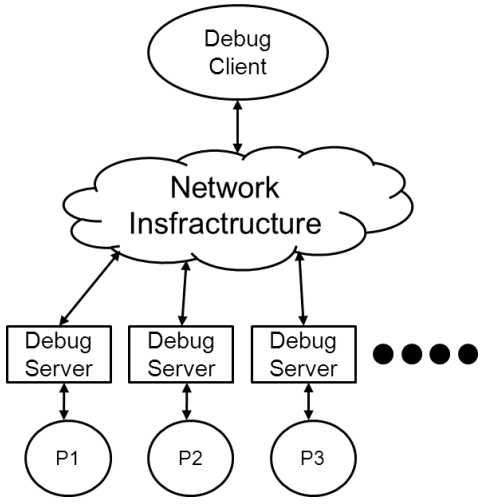


Figure 2-10: General Client/Server Architecture

2.7.1.1 Advantages

- Client/server architecture allows serial debuggers to be reused and deployed.
- Each computational process can be individually monitored, thus allowing errors to be identified.
- Client/server architecture allows the execution of debugger commands on any subset of the parallel processes.

2.7.1.2 Disadvantages

- Client/server architecture limits the number of connections between the root debugger and its debug servers. For example, Linux-based systems allow a maximum of 1024 socket connections per process.
- Systems using this client/server architecture often suffer poor performance because a centralised frontend process has to receive, process and output large amounts of debug data collected from many debug servers.
- Assuming that the frontend process (that is, the client) can handle and display all the results reported by debug servers, users may be overwhelmed by the volume of data. The large amount of data would make it difficult to find the error (the proverbial needle in a haystack).

2.7.1.3 Solutions

To address these issues that mainly emerge when the computing platform and the debugging software scale up, changes have been made to the parallel debuggers. Some now incorporate improved debugging features (such as output representation and data visualisation) while others have enhanced the management of backend debug servers by using the notion of process sets. Yet others have improved the communication layer using advanced network technologies such as a tree-based network. For example, LadeBug [125, 126] connects its independent debug servers using an n-nary aggregating network to overcome the limited number of connections that the root debugger may have to its debug servers. Intel Debugger idb [77] also employs a similar tree-based network to improve the performance in invoking and executing user commands. A general tree-based set up is shown in Figure 2-11.

Debugger outputs can be condensed to present the user with a global view of the application. In another vein, p2d2 [75] provides three different zooming levels to support users when examining program states in either a global application, a group of processes or a single process. p2d2 also uses a notion called *control set* that represents a collection of processes that are subject to process-control requests. Similarly, using IPD [127], users can specify lists and a range of processors, distinguished by node IDs.

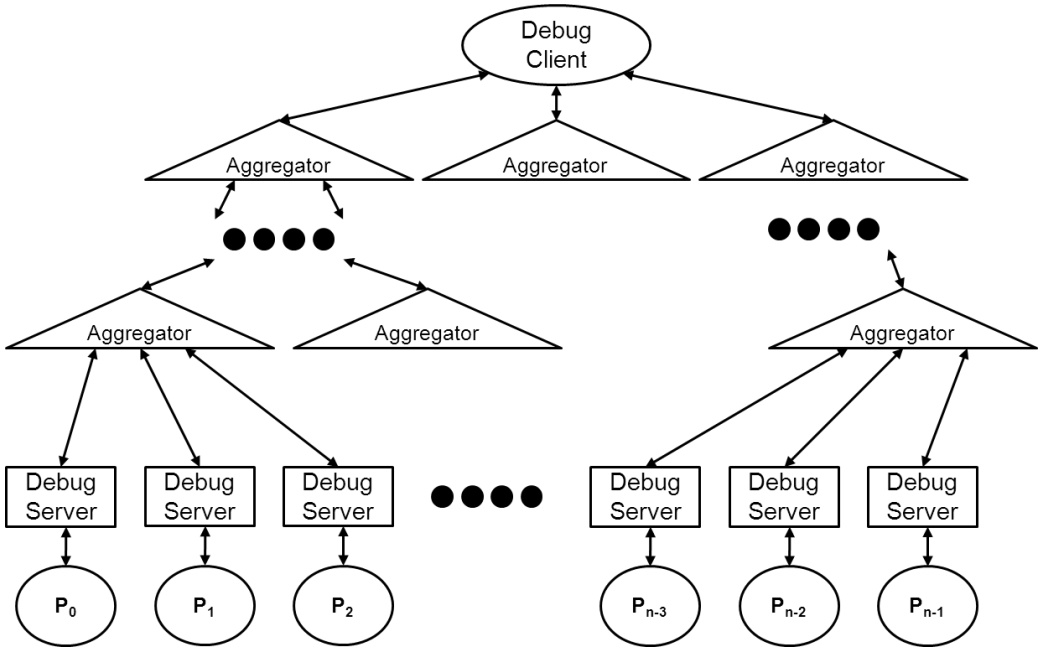


Figure 2-11: General Tree-Based Communication Network

2.7.2 Other Approaches

GDBase [128] is a variation of the client/server parallel debugger where the single frontend process is replaced with a high-performance database system (Figure 2-12). At the backend, programmable GDB instances are invoked and attached to parallel processes. GDBase aims to provide offline debugging functionalities for parallel applications. It gathers runtime information from backend GDB instances and stores those records to a distributed event database. These events can later be analysed to reason about the progress of the program. GDBase is composed of an event-logging handler, an execution engine that is able to process and distribute debugging commands and an analysis system to process the collected runtime data.

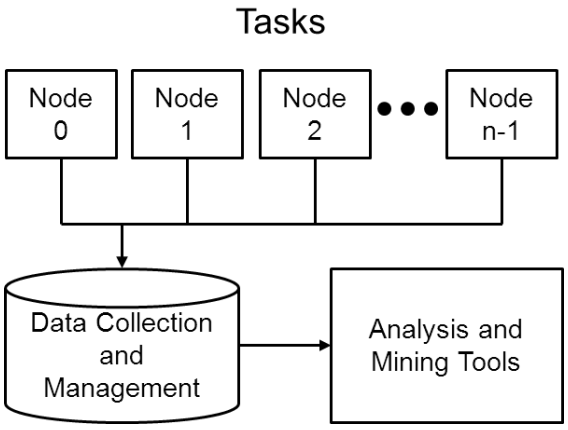


Figure 2-12: GDBase Architecture - Source [128]

One solution to avoid the bottleneck of the client/server architecture is to convert the frontend process into a debug server as well. This is the approach implemented for the Node Prism debugger [129]. This debugging system is itself a message-passing program that consists only of control processors (that is, debug engines) that run on the same nodes as the debugging process (Figure 2-13). Debug commands are broken into requests that are broadcasted to all other control processors. To deal with user-defined process sets, Node Prism uses the *pnset* notion to group control processors into different groups. Upon receiving a broadcasted request, each node independently decides if the request is addressed to its group and performs the requests accordingly. This architecture allows the debugger to scale well with machine size.

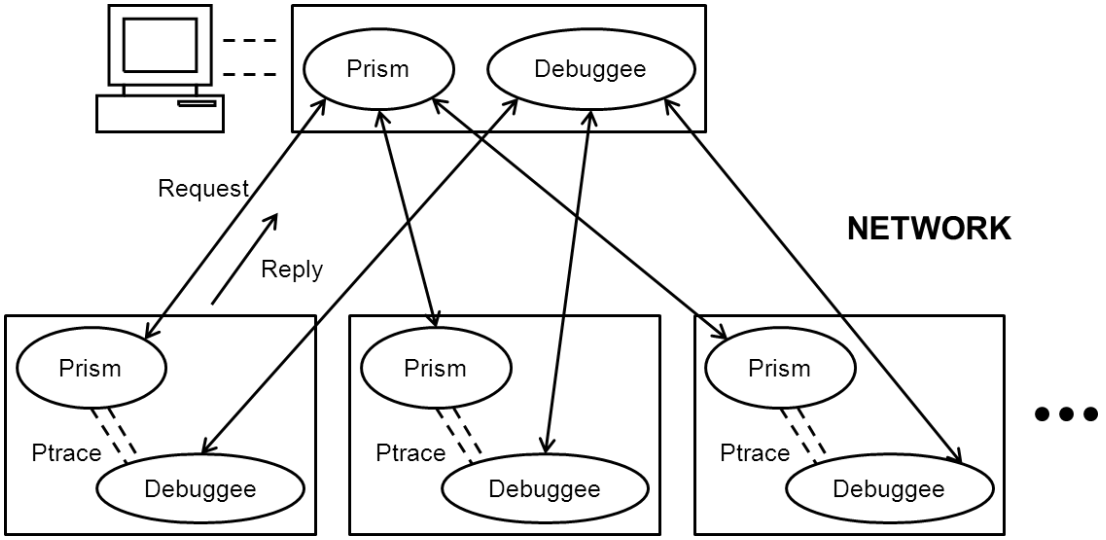


Figure 2-13: Prism Architecture - Source [129]

Another non client/server model architecture is implemented for POET [94]. Similar to other parallel debuggers, POET's system consists of a number of processes (shown in Figure 2-14). However, they are categorised into *event-server* processes, *checkpoint-client* processes and *debug-session* processes. First, the *target-description file* that stores data about the distribution environment is loaded to invoke event-server processes during the POET initialization. Each event-server process is responsible for receiving event data and stores it in an event file. This process collects two types of events: communication events (such as call, call-receive, reply and reply-receive); and local events (such as process invocation and termination). Other processes are treated as clients of event-server processes. For example, the checkpoint process creates a timestamp-checkpoint file useful for computing timestamps while the debug-session process provides user interaction with the debugger.

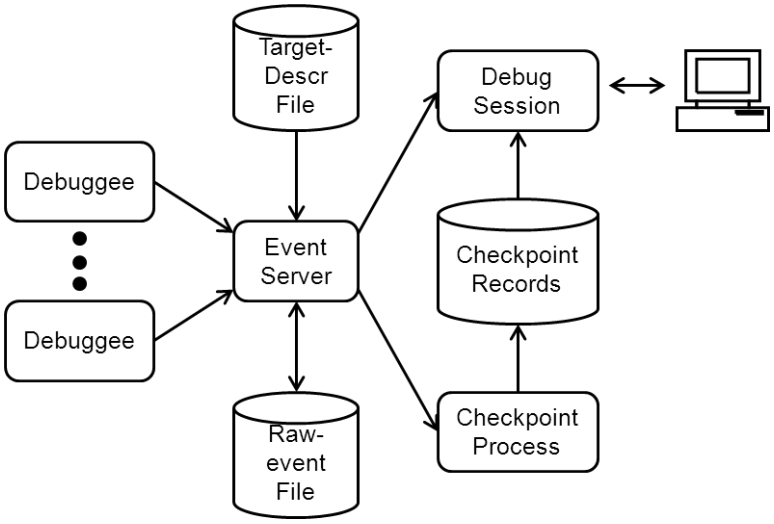


Figure 2-14: POET Overall Architecture - Source [94]

2.8 Summary

This chapter achieved several objectives. First, the debugging environment was reviewed via descriptions of fundamental debugging models and discussion of various sequential-debugging techniques. Second, a high-level overview of parallel computing and the available parallel programming models highlighted the challenges in debugging parallel programs. This part also set the general context for the subsequent discussion on parallel debugging. Third, this chapter reviewed a range of key debugging methodologies. Finally, it delivered various architectural ideas on how parallel debuggers can be implemented.

The main focus of the chapter was the review of parallel debugging techniques. The review showed that many of the debugging techniques used for sequential programming were also applied to parallel and distributed programming. Many of the current debuggers simply improve or combine the traditional debugging techniques discussed in this chapter to support debugging in a parallel environment. Typical features (such as the capability to set breakpoints, to step through program execution, to read and write the variables and memory locations, or to catch program exceptions) are still relied on by many debugging tools, especially interactive debuggers. To cope with the increasing number of processing threads, these typical parallel debuggers enhance general functions such as output representation and data visualisation, improve the management of backend debug servers by using the notion of process sets, and upgrade network-communication capability using advanced technologies such as tree-based networks.

However, this transfer of technique between the two generations of programming is not always successful because assumptions that are true in sequential programming (for example, that parallel-processing threads can be fully controlled, and that runtime data produced by the programs can be processed manually to identify abnormalities) do not hold for parallel and distributed programming. These assumptions fail when more data-intensive applications are developed and run on highly parallel environments such as the petascale systems [130]. Few parallel debuggers allow for the fact that the volume of data to be managed will exponentially escalate when the number of processes multiplies. In this situation, traditional debugging methods become ineffective and impractical. Furthermore, in a petascale context, the imperative to depend on data to perform debugging tasks becomes even more recognisable. Users should utilise parallel debuggers to find abnormalities in programs via automatic data exploring and reasoning. However, in order to focus on data for debugging purposes, new methods and techniques need to be developed because this has not been a focus to date. Identifying and developing an appropriate method is the primary objective of this work.

Finally, standard parallel debugging tools deliver performance that is always bounded by the amount of runtime data as well as the number of processing threads. Therefore, it is important to devise new parallel debugging techniques that can (1) process massive amounts of debug data in order to provide users with relevant insights to program states and (2) better control the underlying computing resource in order to improve the overall performance of the debugging tool. To this end, Chapter 3 revisits the relative debugging framework and the relative debugger, Guard.

Chapter 3

Data-centric Parallel Debugging Using Assertions

Debugging large-scale parallel programs is difficult due to the invocation of many computational threads, the massive amounts of data generated and transformed at runtime and the non-deterministic nature of parallel processing. These issues were all highlighted in Chapter 2. The previous chapter also delivered a review of various debugging strategies, ranging from the traditional such as the interactive, event-based and pictorial techniques, to the more sophisticated methods such as the automatic debugging techniques. Many of these techniques, both traditional and sophisticated, have focused on resolving issues including the non-deterministic property of parallel applications, the control of concurrent processes and the management of distributed data structures. Importantly, some efforts address the cognitive challenge in which programmers cannot integrate the data effectively into a mental model of correct execution because too much data is generated by the program at runtime. However, these solutions, although innovative and addressing necessary issues, often cannot accommodate larger problems run on larger computing systems.

This chapter presents a data-centric parallel debugging technique that addresses the cognitive challenge while providing users with a practical debugging aid. The technique supports advanced data-analysis activities in order to attain valuable runtime information for the debugging process. Although assertions are not inherently a debugging mechanism, the technique uses assertions as a debugging tool. Assertions have been studied and implemented in many programming models and environments for the purpose of improving the quality and correctness of software systems. The advantages in using assertions for debugging software come from three important attributes: their ability to perform error detection, error isolation and error notification. With these attributes, assertions are the ideal tool to perform software-development sanity checks.

The rest of the chapter introduces and describes three distinctive debug-time assertion templates including *comparative assertions*, *general ad-hoc assertions* and *statistical assertions*. Examples for each assertion type are provided to demonstrate how such assertions can be constructed to help find program defects.

3.1 Why Data-Centric Debugging Using Assertions?

Runtime data can be explored, for debugging purposes, both interactively and through a post-mortem. The work in this thesis aims to use assertions as the driving instruments for exploring runtime data. The following discussion reviews assertion usage under the traditional programming model, thus revealing the potential of using assertions for data-centric debugging in a parallel context.

3.1.1 Assertions under the Traditional Programming Model

An assertion is a statement about an intended behaviour of a system's component that must be verified during execution. Assertions were first introduced over fifty years ago by Alan Turing [131]. He expressed the problem of ensuring the correctness of a large system by partitioning the overall verification process into a set of assertions. Later, Dijkstra [132], Hoare [133] and others introduced the idea of program assertions - predicates that are evaluated as a program runs in order to indicate whether an error has occurred. In computer programming, therefore, a programmer defines an assertion to ensure a specific state of the program at runtime. Using assertions, programmers can inject their design intentions such as constraints or contracts into the executable code. As a result, assertions are used extensively for evaluating invariants, checking input parameters and for enhancing program correctness and quality. The key features of assertions are *error detection*, *error isolation* and *error notification* [134].

Using assertions for software development delivers a number of benefits. First, embedding assertions in the code improves *observability* within the design [134]. With the assertion mode enabled, as soon as an assertion violation occurs, the information provided by the assertion failure can be immediately used to identify the defect. Second, using assertions enhances the integration between different modules through correct-usage checking. Finally, as a result of improving observability, the use of assertions effectively reduces the debugging time. These advantages are demonstrated in the following example.

First, consider the following C++ code:

```
int totalFiles = countAvailableFiles(dir);
```

In the code following this assignment, the programmer may assume that the number of available files in a given directory cannot be negative, and consequently may proceed to

use `totalFiles` in subsequent computations based on that assumption. However, if `totalFiles` is negative, due to an error in the `countAvailableFiles` routine, then subsequent computations will fail. Adding the following assertion makes this assumption explicit:

```
assert(totalFiles>=0);
```

If `countAvailableFiles` returns a negative value, the assertion is violated and the programmer can confidently conclude that a bug is in that part of the program. One major benefit of this technique is the ability to detect any runtime error along with the code location. Therefore, a user can often identify the defect without further debugging, thus reducing the debugging time.

Second, using assertions enhances the integration between different modules through correct-usage checking. For example, a pure object-oriented programming language called Eiffel integrates assertions into the language, and introduces a programming paradigm known as *Design by Contract* [135]. Eiffel "...views the construction of a software system as the fulfilment of many small and large contracts between clients and suppliers..." [135]. A *contract* in Eiffel design is an agreement concerning a range of acceptable inputs/outputs between a client (a caller method) and a supplier (a callee method). As a result, an Eiffel programmer can prefix and suffix software components with pre-conditions and post-conditions respectively, to form various contracts between the components in order to ensure correct usage. These contracts are verified during execution, thus offering robustness to the overall system.

```
set_month (a_month: INTEGER)
   require  --  pre-condition  for  a_month  value  between  1-12
      valid_month_value: a_month >= 1 and a_month <= 12
   do
      month := a_month
   ensure -- post-condition for month == a_month
      month_set: month = a_month
end
```

Figure 3-1: Examples of Eiffel Pre-Condition and Post-Condition

Finally, assertions are also used in unit testing which allows software components and modules to be individually tested. An example is the NUNIT unit-testing framework which provides a rich set of assertions as static methods of the *Assert* class [136].

Available assertion subclasses include Equality Asserts, Identity Asserts, Comparison Asserts, Type Asserts, String Asserts, Condition tests, and Utility methods [136].

To conclude, assertion is an effective tool, not only for design and verification, but also for debugging throughout the software-development cycle. In addition, assertions support a more data-centric view of debugging because a user does not focus on the control path, but can assert that various data structures should be in particular states at various stages in the program execution. As stated earlier, language-based assertions are not meant for debugging per se but rather as sanity checks on the state of the program.

3.1.2 Debug-Time Assertions

While the programming-language community has focused on adding assertions to programs, they can also be used as a powerful tool during the debugging phase of software development. When used in this way, a programmer writes ad-hoc assertions at debug time to test the state of a running program. Such *debug-time assertions* can test the state of a large distributed array, for example, and can be refined iteratively by the user in order to narrow down the scope of the problem in the source code. Consider the scientific debugging model as presented in Section 2.1.2. A hypothesis can be generated from the expected state of the program; in other words, the runtime data. Defining assertions to verify the program state at debug time makes the scientific debugging process explicit. A debug-time assertion is a suitable construct for encapsulating and expressing a debugging hypothesis. In addition, such an assertion can be evaluated automatically by the runtime system. That means that the scientific debugging process can be automated which allows it to be conducted on a large-scale problem.

The following are examples of assertions that could be used to locate errors at runtime:

- 'The contents of this array should always be positive';
- 'The sum of the contents of this array should always be less than a constant bound';
- 'The value in this scalar should always be greater than the value of another scalar variable'; or
- 'The contents of this array should be the same as the contents in another array'.

Despite their conceptual appeal, very few debugging environments support assertions. To a limited extent, assertions have been incorporated into some sequential debuggers [9-11, 96, 137]. In the next subsection, a debugging technique discussed in Chapter 2 is

revisited to highlight the potential of using assertions more consistently for debugging production code in parallel environments.

3.1.3 Guard – A Data-centric Debugger Using Assertions

As discussed in Chapter 2, Guard is a relative debugger that adopts the use of comparative assertions for controlling the comparison of runtime data that has been extracted from a reference program and a suspect program. A comparative assertion allows the comparison of two individual data structures. For example, it may be useful to assert that all elements in array A should be less than, greater than or equal to all elements in array B. In this scenario, array A and B could be extracted from the running program or could be populated by the user a priori. By exercising comparative assertions, a user can find errors that are introduced when a program is ported to a new machine, or modified to incorporate parallel-computing techniques, because one version of the code is used for establishing reference values for the comparison.

Implementing comparative assertions in a large parallel machine poses additional implementation, as well as performance, issues over general assertions. For example, a simple implementation of a comparative assertion extracts data from two programs and compares it sequentially in a single (head) node of the cluster. This approach fails to scale to large problems on large machines because there is insufficient room to store all of the data. Furthermore, the transmission of data from each of the nodes, and also the sequential comparison, would take too long for large-scale machines.

3.1.4 A Generic Assertion-Based Parallel Debugger

This research generalises the earlier data-centric debugging approaches in order to create a generic debugging framework, in which users can create more ad-hoc assertions to perform debugging activities, especially in a highly parallel environment. To achieve this goal, this study presents a scheme that executes the assertions in parallel, making assertions over large data structures feasible. Importantly, we base our design on the existing debugger Guard. Therefore, the first sub-goal is to enhance the scalability of a comparative assertion to allow it to debug parallel programs running on tens of thousands of processes. In addition, we develop and explore two novel assertion templates: *general ad-hoc assertions* and *statistical assertions*. These assertion templates are proposed because they provide different ways to reason about the state of the program

through the obtained runtime data. Further, they demonstrate the advantages in deploying data-centric parallel debugging in a highly parallel context because the templates can often be parallelised.

The following sections discuss the details of general ad-hoc and statistical assertions. In particular, the discussion defines and describes these assertions through examples in order to demonstrate how they work and why they are useful in debugging parallel codes. Chapter 4 delivers the design details of an architecture that supports the creation and the execution of these assertions, including comparative assertions.

3.2 General Ad-Hoc Assertions

3.2.1 Definition

General ad-hoc assertions are assertions that can be used on an ad-hoc basis at debug-time. They require data-manipulation operations such as sum or average. For instance, it is often useful to assert that the sum of all elements in a data structure to be less than, greater than or equal to a particular threshold, or to assert that every element of the data structure must be true or false. These simple assertions are fairly easy to establish and control and allow the user to define a series of such assertions to specify the conditions necessary for the correct execution of the program. In addition, the Eiffel programming language allows programmers to inject *design contracts* in the form of pre-conditions and post-conditions. These program invariants are ad-hoc assertions. If a debugger is able to support the use of these ad-hoc assertions, it effectively supports the design-by-contract programming paradigm. For example, given the code snippet below (Figure 3-2) from source file `init.c`:

```
for (j = 0; j < n; j++) {
  for (i = 0; i < m; i++) {
    uold[j][i] = u[j][i];
    …
    pressure[j][i] = 50000.;
  } }
```

Figure 3-2: Code Snippet from init.c

and assuming that the code is invoked with process set `$a`, consider the following assertion:

```
assert $a::pressure@"init.c":180 = 50000
```

This assertion ensures that every element in the data structure `pressure` of process set `$a` is equal to 50,000 at line 180 of `init.c` source file. Such an assertion is useful to verify the success of an initialisation routine before any other computation takes place. Consider another assertion below:

```
assert sum($a::pressure@"calc.c":463) = 50000*n*m
```

This assertion, on the other hand, indicates that the sum value of all elements in array `pressure` collected from parallel processes `$a` at line 463 of source file `calc.c` must be equal to `50000*n*m`. This assertion can be used as a post-condition for a routine that updates the pressure-array elements. Importantly, these assertions can be evaluated in parallel and thus improve the overall performance of the debugger. For example, with the assertion that ensures each element in the pressure array to be 50,000, because multiple debug processes have different portions of the global structure, they can conduct the comparison in parallel and their results can be aggregated to notify any differences found. Or with the sum assertion, each debugging process can work out its subtotals in parallel, before its results can be combined to return the overall sum of the global structure. In conclusion, assertions like those above are useful in allowing programmers to incorporate simple knowledge of the application into the debugging process and thus promote interface correctness between different functions/modules in the application.

3.2.2 Potential Use Cases

In this section we specifically consider potential assertions that are useful for debugging parallel supercomputing applications. While no single template for a parallel program exists, three assumptions are made that underpin the proposed implementation.

First, scalable applications achieve their performance by dividing the problem and the program state into sub-problems and then distributing them across multiple processes. Each process manipulates and transforms the data as independently as possible in order to minimize synchronization overheads. Typical examples are partial differential equation (PDE) solvers in which the key multi-dimensional data structures are distributed across the processors, while the state equations transform the contents of the variables as the algorithm proceeds. Because these solvers march forward in discrete time steps, it is possible to make assertions about the state variables at each iteration.

Second, typical message-passing programs are composed of a single master process and a set of slave processes (often created statically). Usually, the master communicates with the slaves infrequently (the master may do little more than launch the slaves and wait for them to complete, and in fact, it may actually be one of the slaves), but slaves are free to communicate with each other and to synchronise and share their states within iterations. Finally, array data types are first-class objects and can be used in assertion expressions.

In the next section we present some typical use-case scenarios for assertions. While it is unlikely that users would use all the templates presented here in any one debugging session, they are presented here to demonstrate the range of possible debugging operations. While it is possible to implement some of these assertions in existing parallel debuggers, in general, existing parallel debuggers do not support assertions. Later, Chapter 6 will illustrate the use of assertions in debugging a scientific parallel application.

3.2.2.1 Use Case 1: Assertions about Master State

In this first situation, a user wishes to make statements about the state of the master process, typically to test the initial and final conditions. Moreover, if slaves communicate with the master during the execution, these assertions can test the global state of the system as perceived by the master. These assertions can be arbitrary array expressions that evaluate to true or false. For example, one might assert that all elements in a two-dimensional array are > 0 at a particular line of the code. Alternatively, array-slicing operators can be used to select sections of complete arrays.

3.2.2.2 Use Case 2: Assertions about Slave State

In this second situation, the assertions are similar to the master-state assertions of the previous scenario, but apply to either the entire set, or a restricted subset, of slaves. For example, a user may assert that a particular variable in each slave must be < 0, or that all elements of an array in each slave must be > 0. The assertion fails if any of the slaves' assertions fail. Notably, these assertions treat the state of each slave as independent of any other slave, and thus are not used to reason about global data structures that have been distributed across the processors.

3.2.2.3 Use Case 3: Comparing States between Slaves

In this third situation, the goal is to compare variables between different slaves. For example, a user might wish to check that slaves are correctly sharing data in ghost bands

of arrays. Such assertions can verify whether or not the inter-slave communication has been performed correctly.

3.2.2.4 Use Case 4: Comparing Master and Slave State

In this fourth situation, a user wishes to compare variables in the master with slave variables. For example, the master process distributes a set of constants to all of the slaves, and then wants to check that those slave processes have been initialized correctly. In a more complex example, a computation might distribute a data structure across the slaves so that each manipulates only a portion of the original array. In this type of assertion, it is important to check that the data has been correctly distributed to the slaves. In order to do this, the debugger needs to know the data-decomposition scheme used so it knows which elements to compare. The debugger also needs to support the concept of data distribution.

3.2.2.5 Use Case 5: Reductions over Slaves' State

In this fifth situation, these assertions are similar to assertions about a master's state, but they allow the reduction operation across the slaves. For example, a user might want to take the sum or average of values in the slaves and test the reduced result against a threshold. Or, a user might want to compute some reduction operator across a distributed array, involving both the data-decomposition functions as well as global reductions.

In some of the above assertions, the underlying parallel machine can be leveraged to improve performance. In particular, reductions can be computed in parallel during the computation phase and then reduced using tree-based reduction operators where possible.

3.3 Statistical Assertions

3.3.1 Motivation

As more data is produced and gathered, *data mining* [138] is becoming an increasingly important concept in transforming data into *information*. Data mining is popular in a wide range of profiling practices, such as marketing, finance, climate modelling and earth systems [139]. In addition to generating raw data, most high-performance software also produces patterning information in the form of histograms, probability distributions or

data models (that is, mathematical functions). These statistics not only give the users insights to the observed phenomena, but also sometimes display unusual details of the computation.

In our own research, we recently recognised the importance of extracting statistical information for debugging purposes while chasing an error in one of our software tools (the debugger, in fact). Specifically, we had generated a performance model based on a set of simulations and produced a plot that summarized the model behaviour with a two-dimensional graph. This simple display highlighted an error and we subsequently found a coding bug. Importantly, the error became obvious not through the *detailed examination of process state*, as supported by almost all debugging tools, but through a simple proxy – a graph showing one derived variable against another. The location of the bug could then be deduced quite accurately without viewing the source code, because the graph contained sufficient information about the type of error. This example highlights the potential of using statistical data instead of raw data to locate coding defects.

3.3.2 Definition

Until recently, many debuggers were only concerned with the raw runtime data obtained from the executing program. This research introduces a new type of debugging assertion: the *statistical assertion*.

A statistical assertion is defined as a user-defined predicate consisting of two *data models* in the form of either statistical primitives such as mean or standard-deviation values or function models such as histograms or density functions. Statistical assertions allow a user to compare data-pattern information between two data structures whereas earlier assertions required the comparison of exact values. For example, it is possible to assert that the *mean* value of a large dataset is between certain bounds before or after a function call, or that the number of elements in an array needs to be in a specific range. More advanced statistical assertions allow the user to state that the *histogram* formed by all elements in a specific dataset must be equivalent to a *histogram* formed by another dataset, or to assert that all elements in a dataset should be normally distributed. The essence of this approach is (1) to diminish the substantial raw datasets into manageable statistical blocks so that comparisons can be conducted, and (2) to enable real-time statistical observations on debugged datasets.

Statistical assertions are useful in debugging large-scale scientific problems for two reasons: they allow users to focus on the scientific *meaning* of the computations instead of the raw data values produced by them and they reduce the complexity in debugging *stochastic processes* (for example, those found in Monte Carlo methods). Statistics can be used to reflect scientific knowledge behind a computation; thus by using statistical assertions, a user can integrate such knowledge into the debugging process and transform it into runtime invariants that ensure the correct execution of the code at runtime. Failure to comply with such expectations could lead a user to an incorrect computation. Furthermore, stochastic processes are difficult to test and debug because the program state is often non-deterministic. However, if we disregard the exact data values, the data patterns extracted from those datasets are often deterministic. Statistical assertions allow a user to capture such determinism and make debugging stochastic simulations more practical, while reducing the complexity of processing raw data.

3.3.3 Examples

Here we define two subtypes of statistical assertions: *simple statistical assertions* and *assertions using histograms*. This does not mean that statistical assertions can only be constructed as one of these types. The purposes of introducing these two subtypes of statistical assertion are to give a proof-of-concept for statistical assertions and to showcase their potential usage in reasoning over large data structures. The following section describes potential uses of statistical assertions for debugging purposes.

3.3.3.1 Simple Statistical Assertion

It is often useful to calculate simple statistics such as counting the number of zero elements in an array, or identifying the number of elements in an array that fall in a specific range, or computing the mean or standard deviation of the elements in a given dataset. Statistics like these allow a user to detect if a dataset contains unexpected values.

3.3.3.2 Statistical Assertions Using Histograms

A histogram is an important tool for exploratory data analysis. Statisticians have traditionally used histograms to convey information such as the general shape of a frequency distribution, symmetry and skew of a distribution and the modality of the dataset. In some cases, histograms are also used for formal verification of experimental results [140].

3.4 Summary

This chapter introduced a data-centric approach that uses assertions as its primary medium for debugging large-scale applications. Assertion offers key features such as error detection, error isolation and error notification; thus it can be exercised as a potent instrument for debugging software. However, language-based assertions are not meant for debugging tasks but rather as sanity checks on the state of the program. Moreover, evaluating assertions in a very large parallel machine poses a number of implementation as well as performance issues. Therefore, to support the use of assertions in debugging large-scale parallel codes, this study proposes a framework that executes the assertion in parallel, making assertions over large data structures feasible. This framework will be properly discussed and developed in Chapters 4 and 5.

This chapter has also categorised three assertion templates: general ad-hoc assertions, comparative assertions and statistical assertions as shown in Figure 3-3. These assertion templates are proposed because they provide different ways to reason about the state of the program through the obtained runtime data. More importantly, the assertions can often be parallelised.

The following chapter presents the design of the target assertion-based debugging framework. Apart from the fundamental modules that are typical of a parallel debugger (for example, the client/server architecture and network infrastructure), several important components and algorithms must also be integrated into the framework. The next chapter will also explain how different assertions can be conceptually supported.

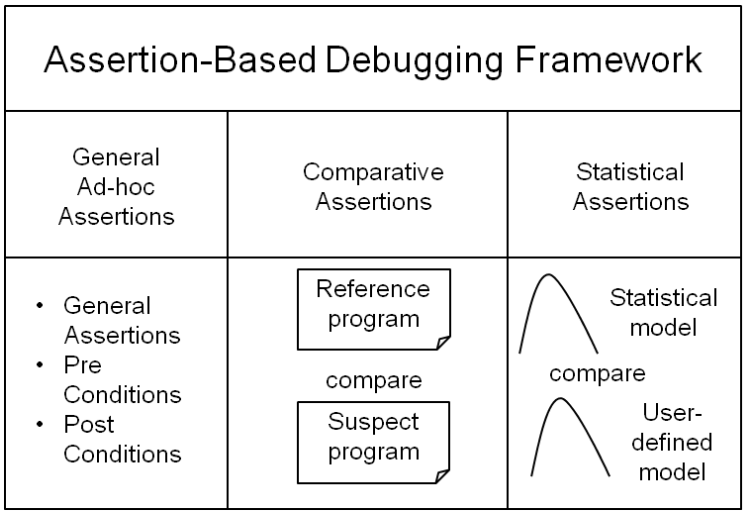


Figure 3-3: Assertion-Based Debugging Framework with Three Assertion Templates

Chapter 4

The Design of an Assertion-Based Parallel Debugger

An assertion-based parallel debugger integrates the functions of a conventional parallel debugger and a number of specific features required to support the data-centric debugging paradigm using assertions as its core. There are two reasons why conventional parallel debugging operations must be included. First, although assertions on their own are powerful enough to detect an anomaly, it is usually necessary to resort to conventional debugging operations to locate the exact source of the error. Second, in order to execute assertions, the debugger requires a base set of functions that support process control and data inspection. These are normally provided in any parallel debugger. For these two reasons, we based our design on an existing debugger, called Guard. Guard supports the control of a multi-process parallel program, using extensions to conventional sequential debugger primitives. Specifically, it uses a command line syntax called HPDF that supports concepts such as parallel program invocation, process sets and the like [25].

Nevertheless, extending a conventional parallel debugger to support the use of debug-time assertions requires a range of significant additional features. This is because the debugger must be able to:

- evaluate ad-hoc debug-time assertions;
- reconstruct a global view of data where users make statement about the state of the computation as a whole; and
- execute the assertion in parallel by exploiting the underlying parallel system.

The first two points are related to the functionality of assertions, and the third relates to the performance of an assertion-based system. Accordingly, the features can be grouped into three categories:

i. ***data decomposition support***, including the formation and evaluation of user-defined data decomposition schemes;
ii. ***advanced parallel data reduction***, including the support for various data reduction algorithms, the construction and evaluation of user-defined reduction functions; and
iii. ***scalable comparison of large datasets***, including the ability to compare large data structures in reasonable amounts of time.

This chapter will describe a design that addresses these architectural considerations. In particular, it discusses each of the key objectives in detail, and describes how they meet the functional requirements of an assertion-based parallel debugger. These include:

- the architectural details of the base debugger, Guard;
- a data decomposition engine;
- an advanced data reduction framework; and
- several highly scalable comparison strategies.

4.1 Revisiting Guard

The relative debugging paradigm, as well as the relative debugger Guard, has been discussed a few times in this thesis already. However, since the debugging tool presented in this work is a direct successor of Guard, the following subsection describes the architectural aspects of Guard that largely influence the work here.

4.1.1 The Development of Guard - A Historical Aspect

Abramson and Sosic introduced the technique of relative debugging in 1995 as an automated debugging tool for supporting the process of porting existing sequential and vector codes to new platforms [116]. This technology aids the process of software evolution. The idea is to use a correct version of an application (known as the reference program) to compare against the ported version of the same application (known as the suspect program) at runtime. By running and controlling both programs simultaneously, key data structures from both runs can be extracted and compared at a central client in order to identify the point of divergence. Data differences can then be used to narrow down the region of problems in the new code, which leads to the discovery of bugs in the suspect program.

In 2000, Watson and Abramson extended the framework to cater for the relative debugging process of parallel programs [25, 141-144]. The parallel relative debugging technique is implemented so that key data structures, which are distributed across multiple processes, can be combined and compared centrally. This is where data decomposition information becomes crucial in the process of debugging parallel programs. It resolves the problem of evaluating the global state of the program, instead of dealing with data from each process individually. Importantly, relative debugging introduces a declarative comparison method in which a user defines a series of assertions that specify the required comparisons between the two programs. Assertions are compiled into a graph that specifies the operations required to perform the comparison. This is the core engine for running assertions in the relative debugging framework, and has been used to implement ad-hoc debug-time assertions as discussed in this thesis [130].

4.1.2 Guard's Debug Commands

Guard is primarily a command line debugger (although there are a number of GUI-based versions, including an integration with Eclipse [145]). Its command syntax is based on a standard called HPDF, which supports standard 'GDB like' commands such as *breakpoints, print, step*, and so forth [25]. When a user starts a parallel program using the `invoke` command, Guard returns an array of process identifiers as a process set (`<process_set>`). Process sets can be used to alter the focus of any commands that are issued; thus a user can restrict, or broaden, any command to a range of processes. Process sets can also be used by an *assert* statement. An assert statement contains a source file (`<source_file>`) and a line number (`<line_num>`) to indicate where the assertions are to be evaluated. The general syntax for declaring an assertion in Guard is presented below.

4.1.3 Assertion Syntax

In Guard, a relative debugging assertion is constructed by two *expressions* and they are limited to accept only program variables. The following grammar (Figure 4-1) gives a general format for Guard's expression.

```
E::=<proc_set>::<gdb_var_expr>@<source_file>:<line_num>
```

Figure 4-1: Guard Original Assertion Syntax

and an assertion is built as

`assert` E_1 `=` E_2

where E_1 and E_2 are constructed by the grammar above.

Note that `<gdb_var_expr>` can be any acceptable GDB expression, which indicates that it could be complex when a pointer to an array is to be used or type-casted. Below is a sample relative debugging assertion.

```
assert $a::big_var@"ref.c":400=$b::large_var@"sus.c":500
```

The above syntax compares two runtime variables for equality. More comprehensive assertions require the comparison between variables and constants, between content of arrays and distribution models, or between a histogram formed by one array and a histogram constructed from another array. This requires a broader assertion definition, which is discussed later in Chapter 5, Section 5.1.1.

4.1.4 Data Parallel Decomposition

Almost all parallel algorithms partition and decompose large data structures across the multiple processors (and their associated memories in distributed memory machines). Therefore, in order for the debugger to reason about the global data structure before decomposition, it must be aware of the way that the compiler (or the programmer) has decomposed the data. Watson [25] in his thesis described a parallel mapping function that specifies the data decomposition technique that has been employed; and provides a mechanism to specify such a mapping, using the following *map* function (Figure 4-2):

```
map func(P::V)
  define index(i,x) = expr1
  define proc(j,x) = expr2
end
```

Figure 4-2: Original Map Function Template

Here, `func` is declared as a parallel map which is composed of an `index` function and a `proc` function. While the `index` function specifies the relationship between each element of a serial array and the corresponding element of the parallel arrays, the `proc` function identifies which processor a particular element of the serial array resides on. The algebra of `map` function is powerful enough to describe any arbitrary regular decomposition scheme, and it has been used successfully in reconstructing small global arrays distributed across a small number of processors. Because the algebra goes through each and every element of the global array and performs the mapping sequentially, this mapping mechanism is not efficient when working with large arrays distributed across tens of thousands of processes. Therefore, to accommodate the evaluation of an assertion across many processors, this research proposes a simpler but more scalable mapping method. It is discussed in full detail in Section 4.3.

4.1.5 Guard Architecture

4.1.5.1 Client-Server Architecture

Guard employs a client-server model, where the debugger is divided into a single client and multiple servers, to ensure that the processes being debugged can be distributed onto multiple platforms and can be controlled independently (Figure 4-3). Other parallel debuggers such as TotalView, p2d2 [75], and DDT, have adopted a similar client-server

architecture, albeit with different server technology. For example, DDT uses GDB as its debugging engine, whereas TotalView has its own proprietary code.

Users interact with the debug client, so it is generally executed on the user's local host. The client's user interface accepts and processes debug commands in either of two modes: immediate or deferred. Deferred commands are compiled by the *Dataflow Compiler* into a *dataflow graph*, which is then executed by a runtime *Dataflow Engine* (also called *Graph Interpreter*). The following sections discuss the implementation of the Dataflow Compiler and Dataflow Engine that has been reused to support the evaluation of multiple ad-hoc debug-time assertions. There are other important architectural aspects of Guard including the Architecture Independent Format (AIF), on which interested readers can find a full description in other literature [25, 142, 144].

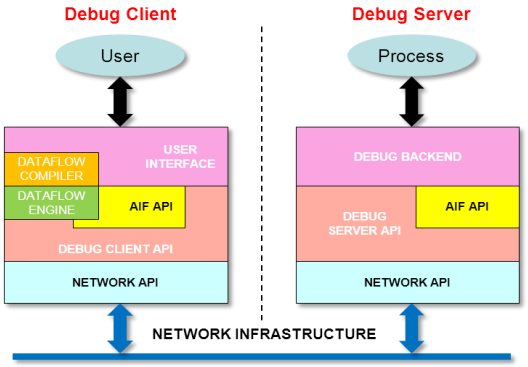


Figure 4-3: Guard Client-Server Architecture

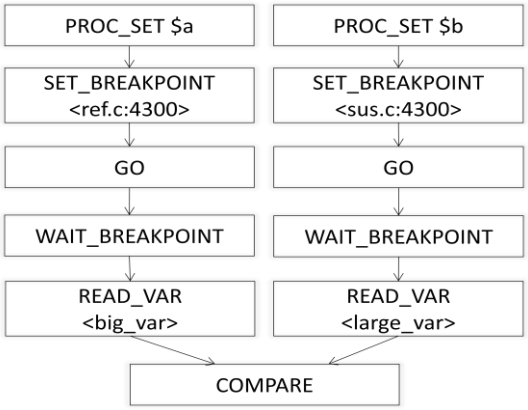


Figure 4-4: Abstract Assertion Dataflow Graph

4.1.5.2 Dataflow Graph and Dataflow Engine

Executing relative debugging assertions requires the debugger to handle asynchronous events. Consider the example of an assertion that requests the comparison of two variables at different breakpoints. With control-flow semantics, the debugger can be deadlocked while waiting for both breakpoints to be reached. Therefore, Guard employs a dataflow execution mechanism to manage the process of capturing runtime data. A full description of this idea is found in [25]. The Dataflow Compiler translates assertions into a low-level graph description. The graph is later used by the Dataflow Engine to execute the assertions using an execution semantics much like early dataflow computers [146]. Figure 4-4 depicts how a comparative assertion like the one below has been compiled into a dataflow graph.

```
assert $a::big_var@"ref.c":4300=$b::large_var@"sus.c":4300
```

When a user starts the graph, the Dataflow Engine executes the nodes sequentially. In Figure 4-4, nodes can be categorised into two types: local command nodes such as `COMPARE` and debug server command nodes such as `GO` and `READ_VAR`. With a debug server command, the graph interpreter packs the command into a request and broadcasts it to the debug servers. Debug servers then translate the request into low-level debug commands (such as break, print or continue) and pass them to GDB[3] instances.

In the example above, the assertion requests data collected from variable `big_var` of process set `$a` to be compared against data collected from variable `large_var` of process set `$b`. The program is executed until the breakpoint occurs. The `READ_VAR` node instructs the debug servers to extract data from `big_var` and `large_var` arrays using the GDB's print command. The data is then collected at the client node. Finally, the outcome of the assertion is performed by the `COMPARE` node.

This research reuses the Dataflow Compiler and Dataflow Engine from previous work [25] and only expands its functions through introducing new command nodes. These new nodes represent new types of data reduction and data comparison techniques that are essential for the execution of the types of ad-hoc debug-time assertions involved in this research. Chapter 5 will discuss the implementation details for the extension of the Dataflow Compiler and Dataflow Engine.

4.1.5.3 Debug Server

While the debug client handles user's commands and processes user-defined assertions, Guard uses *debug servers* to manage and control the target parallel processes. For example, in debugging an MPI application, each debug server is in charge of an MPI rank process. As shown in Figure 4-3, each debug server, through the network infrastructure, accepts and processes requests from the debug client, instructs GDB instances to retrieve debugging information from the running program, and standardises the format of the results (using AIF)[4] before sending them back to the debug client. When implementing ad-hoc assertions of the types discussed earlier in Chapter 3, we have extended the debug servers used in the earlier version of Guard [25]. This is discussed later in Chapter 5, Section 5.1.3.

[3] GDB has been used as a matter of convenience and could replace this with any equivalent debug engine.
[4] AIF is a library that provides standardisation for programs running on different architectures and platforms.

4.1.5.4 Network Infrastructure

The *Network Infrastructure* layer supports different communication protocols, including basic socket connections and tree-based communication networks such as MRNet (Multicast/Reduction Network) [147]. This layer has been recently upgraded to improve the scalability [148]. Despite the significance of this work on the overall performance of the debugger, this thesis will not discuss its implementation details because it does not directly contribute towards the assertion-based debugging paradigm. In addition, there are several studies that have thoroughly discussed the integration of such a communication layer in their solutions [148-151].

4.2 A Generic Assertion-Based Debugging Architecture

Guard's architecture, as discussed above, has been designed to accept and process comparative assertions, but because of various performance limitations, it is restricted to small scale problems. This work targets a general debugging framework, in which more general assertion types (such as those introduced in Chapter 3) can be created and evaluated efficiently in highly parallel environments. Evaluating assertions in a petascale parallel machine causes several implementation challenges and performance issues. In order to parallelise the evaluation of ad-hoc debug-time assertions, a parallel debugger requires some important features. First, it needs to be able to control and collect distributed data from remote debug servers. This feature is currently supported by most parallel debuggers as reviewed in Chapter 2. Second, the debugger needs to aggregate sections of data into a single structure to compare it against predefined values. This requires an ability to reconstruct a global data structure from sub-structures given knowledge of how the global structure has been distributed across many processing cores. However, it is more efficient if the collected data can be reduced using mathematical filters, such as the sum or average operators. In the petascale context, the volume of debugged data is massive, which leads to the problem in holding the overall data structure in one place. Finally, in supporting more scalable comparative assertions, more advanced comparison techniques are required.

Therefore, while inheriting several key components from Guard including the Dataflow Compiler and Engine, and the architecture independent format (AIF), an assertion-based debugging architecture requires (1) enhancement of existing components and (2)

integration of several new components, as depicted in Figure 4-5. In terms of functionality, the new components can be categorised into three groups.

i. ***data decomposition support***, including the formation and evaluation of user-defined decomposition schemes that have been applied on global data structures;
ii. ***advanced parallel data reduction support***, including various data reduction algorithms, the construction and evaluation of user-defined reduction functions; and
iii. ***scalable comparison of large datasets***, including the ability to compare large data structures in a reasonable amount of time.

The rest of the chapter will discuss the design for each of the new components. Since the extension of the existing debugger is mostly an engineering activity, it will be discussed further in Chapter 5.

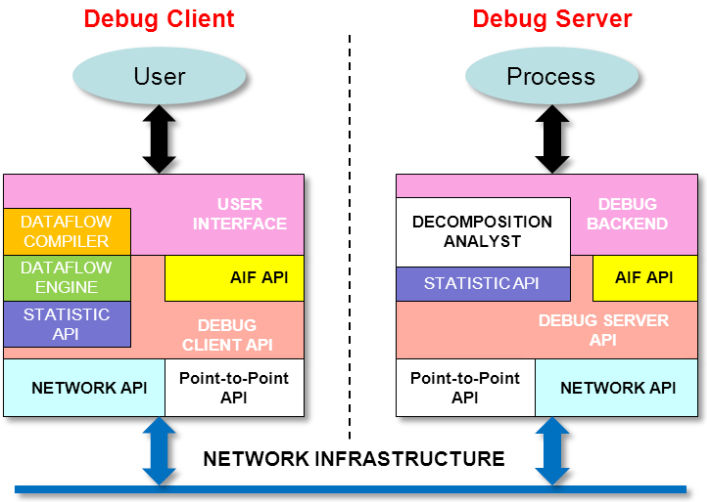


Figure 4-5: Assertion-Based Debugging Architecture

4.3 Data Decomposition

The debugger requires an understanding of the way the data is decomposed across the parallel processors. This is because the assertion may pertain to a data structure that is distributed, and may never exist in a single node. In addition, to evaluate an assertion independently in parallel, one requires a debugging architecture which understands how the data has been decomposed, and knows how to distribute such information across the remote debug servers. This is important because if the assertion can be executed in parallel, then the technique scales as the machine size increases. For example, one may

wish to assert that every element of a distributed 2D array is greater than some constant. To evaluate this assertion, the debug client requests the individual debug servers to extract the portions of the variable that they own, perform comparison of each element against the constant, and return an array of values to the debug client on the frontend node. The debug client then needs to know the decomposition scheme used in order to rebuild the complete resulting array and display it to the user as the outcome of the assertion. Furthermore, in order to evaluate the type of comparative assertion as reviewed in Chapter 3, the debugger must be able to describe the distribution of an array across multiple processes, because the reference program and the suspect program might use different data decomposition schemes. However, not all assertions supported in this framework require data decomposition layout for their evaluation. For instance, the type of statistical assertions as proposed in Chapter 3 use reduction operations such as histogram that are associative and/or commutative. Evaluation of statistical assertions does not require block layout information using the blockmap's syntax.

Earlier in this chapter, a *map* function, which serves as a mechanism for managing data decomposition, was reviewed. Because map's algebra deals with each array element individually, arbitrary regular schemes in data distribution can be undone. However, as a cost of flexibility, the existing map function suffers severe performance issues and is not efficient in reconstructing large data structures. Importantly, in practice, the most common decompositions are typically regular *block*, *cyclic* or a *block-cyclic* decomposition [64, 152]. Blocking information can be used to accelerate the process of mapping distributed data. The following text discusses a mapping algebra that maps blocks of data instead of each single element of the array one by one.

4.3.1 Block-cyclic Decomposition

In order to describe the data decomposition, a special function, called *blockmap,* is developed. Blockmap is inspired by the *block-cyclic data distribution* scheme [153], and it differs from the algebraic technique originally used in Guard for describing data decomposition [25]. In fact, it follows the syntax used in HPF [64] and UPC [70]. It includes a collection of expressions that describe the decomposition scheme. Each dimension of the data array is denoted either as *block*, *cyclic* or asterisk (*). *Block* means a contiguous number of elements of the specified dimension (row or column) will be mapped to a specific processor (i.e. high order bits of the index dictate the processor number). *Cyclic* means that each element of the given dimension will be mapped to a

processor in a circular manner (i.e. low order bits of the index dictate the processor number). Finally, "*" means the decomposition for that dimension is ignored. Figure 4-6 through to Figure 4-9 show examples of typical decompositions for 2D arrays.

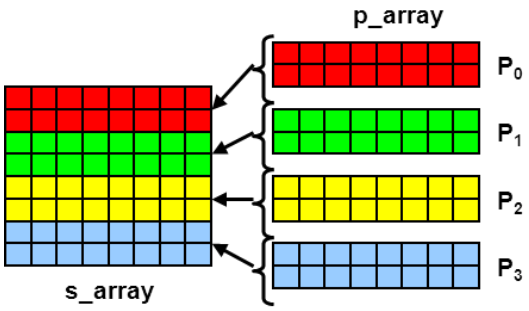


Figure 4-6: (block,*) Decomposition

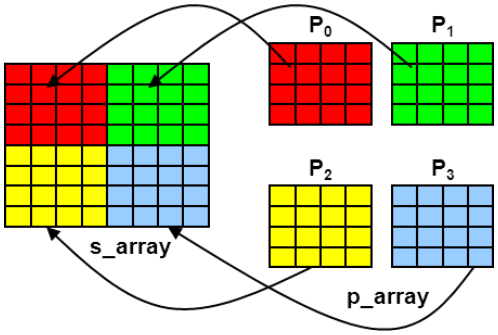


Figure 4-7: (block,block) Decomposition

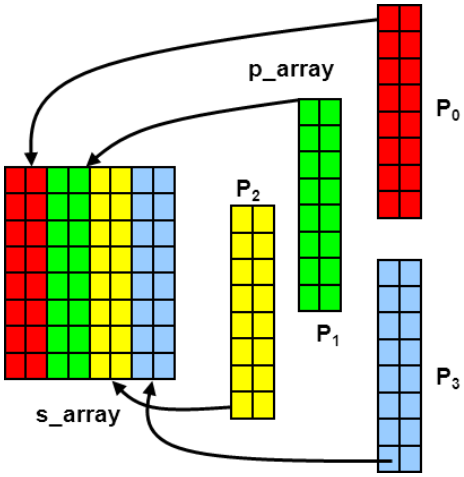


Figure 4-8: (*,block) Decomposition

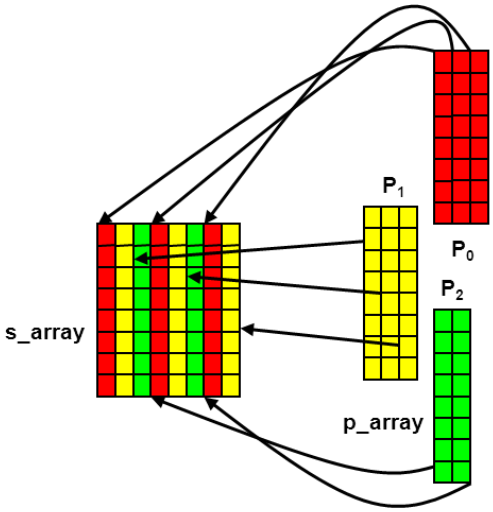


Figure 4-9: (*,cyclic) Decomposition

Note that while blockmap uses an HPF like syntax to describe the way an array is decomposed, this does not mean that programs need to be written in HPF. Programs can be written in any programming language that allows a programmer to write programs that have a regular decomposition strategy – whether it is performed by the compiler or the runtime system. The goal here is simply to inform the debugger of the decomposition that has been used so it can interpret the entire data structure, and other syntactic schemes would suffice.

The work is, however, currently restricted to programs that use regular decompositions. Whilst this presently indicates that users cannot debug programs with irregular decompositions, it still covers an enormous number of practical parallel applications. For example, a block-cyclic scheme is used in the NAS Parallel Benchmarks [154] and the ScaLAPACK library [155]. Moreover, there are numerous PDE solvers with regular

stencils and grids that use block and/or cyclic decompositions [156-159]. Further work, discussed later in the thesis, will hint at ways this restriction could be relaxed.

The following text defines a general set of algebra that specifies how partitioned arrays on each process can be mapped back to the global array. As a simple convention, the overall distributed array is noted as *global array*, while partitioned arrays owned by individual compute processes are called *sub-arrays*.

4.3.2 Blockmapping Algebra

The set of algebra provided in this section describes how a collection of sub-arrays can be *re-blocked* into a global array. In contrast to the previous *map* procedure produced by Watson [25], where individual array's elements are mapped sequentially, we are interested in finding the biggest block of contiguous elements that can be mapped onto the global array.

4.3.2.1 Definition

Let `data(`d_1`,`d_2`,…,`d_k`)` denotes the shape of an array. In this case, `k` is the rank of the array and d_i is the size of i^{th} dimension of the array. For example, if `A=data(4,4,6)`, A is an array of rank 3 (3 dimensional array) with 4 layers of matrices, each has 4 rows and 6 columns.

Let `blockmap(A,`m_1`,`m_2`,…,`m_k`)` represents the decomposition function of a global array. For instance, given the 3D array A above then `blockmap(A,2,2,6)` indicates that the array A is decomposed into 4 sub-arrays of shape `(2,2,6)`. For example, the decomposition of a 3D array A with shape `data(4,4,6)` across 4 processes using decomposition function `blockmap(A,2,2,6)` can be depicted as in Figure 4-10 below.

4.3.2.2 Proposition

Imagine that an array A of rank `k` (i.e. A has the shape as `data(`d_1`,`d_2`,…,`d_k`)`) can be flattened onto a single rank array A' (i.e. A' is one dimensional array), then the location of each element in the original array can be mapped to a new location in the single rank array using the algebra below:

$$A[x_1]\ldots[x_k] \rightarrow A'[y] \text{ where } y = \sum_{i=1}^{k} q_i x_i \text{ and } q_i = \begin{cases} \prod_{j=k+1}^{k} d_j \,|\, 1 \le i \le k \\ 1 \,|\, i = k \end{cases} \qquad (1)$$

To see how the flatten function works, consider the array A in the above example. Using the formula (1), cell A[2][3][3] can be located at index 72 in the vectorised array A'. The vectorisation of a 3D array is shown in Figure 4-11.

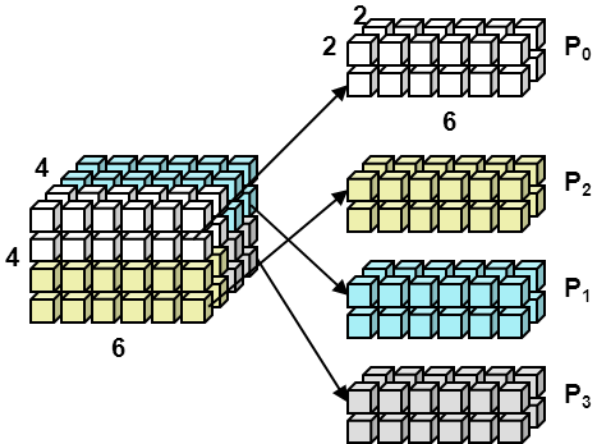


Figure 4-10: Simple Block-Cyclic Decomposition of a 3D Array

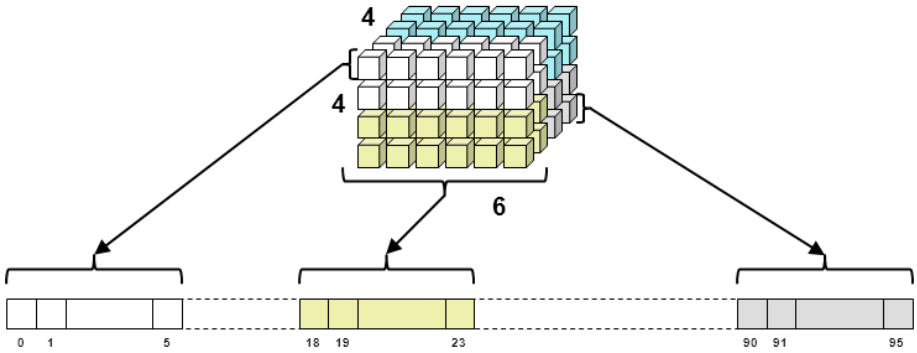


Figure 4-11: Standard Array Vectorisation Function

The process of re-blocking sub-arrays onto the original global array can be done by firstly vectorising the sub-arrays. For each vectorised sub-array, we are interested in finding the biggest block of contiguous elements that can be mapped directly to the flattened global array. For simple cases where a trivial distribution scheme such as `(block,*)` is applied on a 2D array, the biggest block would be the whole sub-array itself. However, with the example global array A illustrated above decomposed with `blockmap(A,2,2,6)`, the biggest mappable block only has a shape of `(2,6)`. This leaves the sub-array to be further partitioned into 2 blocks of `(2,6)`. This desired biggest block is referred to as a *mappable block* and the size of such block is called *block_size* from here onward.

Due to the array vectorisation algebra, elements in less significant ranks are arranged first in the vectorised array. Hence, a mappable block can be identified by iterating in reserve through the data vector $\texttt{data}(d_1,d_2,\ldots,d_k)$ and vector $\texttt{blockmap}(A,m_1,m_2,\ldots,m_k)$, and

block size can be calculated using formula (2) below. The iteration finishes when d_i is not equal to m_i.

$$block\_size = \prod_{i=n}^{m_i \neq d_i} m_i \qquad (2)$$

Let us again consider the global array A with `data(4,4,6)` decomposed with `blockmap(A,2,2,6)`, the size of the mappable block will be $m_3 * m_2 = 6*2 = 12$, and the mappable block has a shape of `(2,6)`. This means that the first 12 cells in each of the sub-array owned by a process P_i can be copied to the global flatten array respectively. Then the second block of 12 elements can be copied next. The process is illustrated in Figure 4-12 below.

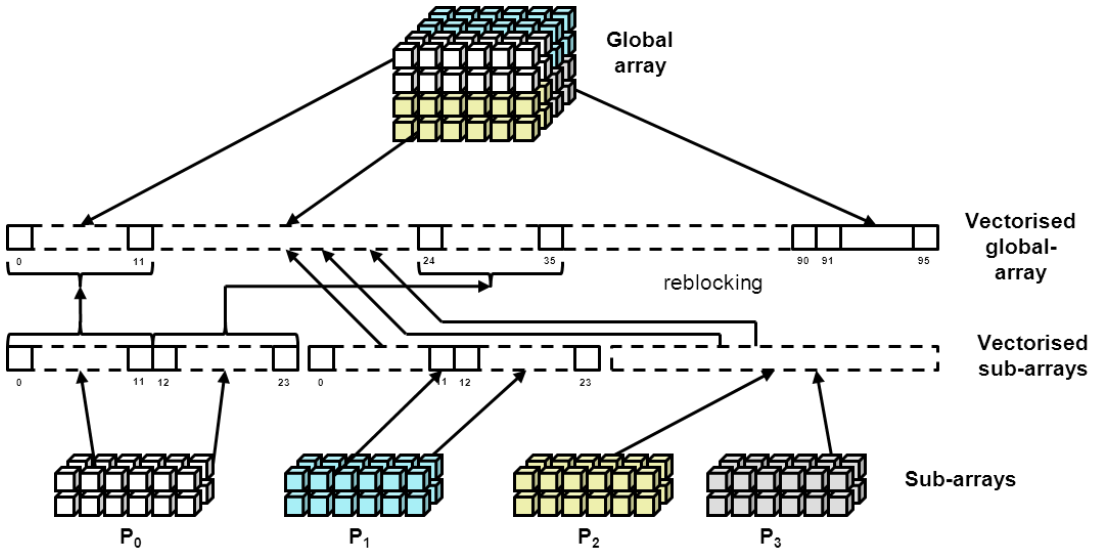


Figure 4-12: Blockmapping Process

Finally, after the flatten global-array is successfully constructed, it can be converted into the original multidimensional global array by restructuring the elements using the formula (1) above in reverse.

4.3.2.3 Example

We have demonstrated how the blockmapping algebra works and provided an example where a distributed 3D array of shape `(4,4,6)` was reconstructed. Another example with a 2D array is presented here in order to provide the reader with more insight into how the algebra works. Consider a global array `B=data(80,100)` which is decomposed across 16 processes using `blockmap(B,20,25)`. Since m_2=25 and d_2=100 ($m_2 \neq d_2$), the mappable block is `(25)` and has 25 cells. Accordingly, an individual sub-array is further

viewed as 20 separated mappable blocks. They are re-blocked into the global vectorised array as depicted in Figure 4-13.

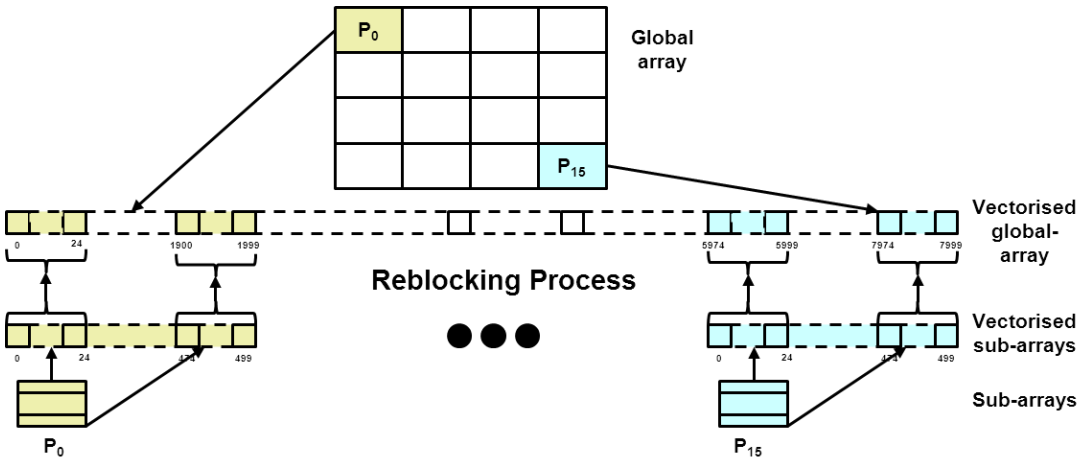


Figure 4-13: Blockmapping Example 2

4.4 Extensible Parallel Statistic Framework

The implementation of statistical assertions in the current debugging architecture requires a framework in which significant modelling and patterning algorithms are presented. In addition, the framework allows users to define their own modelling functions, and create assertions using those functions. This research also investigates how these modelling algorithms are executed in parallel across the debug servers, and how the results of those can be aggregated, and compared. Note that this study assumes the availability of various data patterning algorithms and data mining techniques. Inventing or designing new, innovative data mining methods is not considered within the scope of this work. In addition, statistical assertions are provided as an example of the data-centric debugging technique, rather than the extent of the overall approach. It is possible that further enhancements may be developed in the future, and these can benefit from the new framework. The following sections focus on a design that allows users to create their own statistical reduction routines as well as statistical models. More importantly, it also focuses on how those routines can be parallelised. Details regarding the syntax and the implementation are discussed in detail in Chapter 5.

4.4.1 Split-phase Statistical Operation

The parallel computation of basic statistics such as average, max, min is relatively straight forwards; however, more complex statistics require special handling. For example, given

a dataset X, the typical standard *two-pass algorithm* for computing standard-deviation value [160, 161] is not efficient in parallel, because it requires all elements in X to be examined twice. Even though there are one-pass algorithms that can compute the standard-deviation value by going through the dataset only once, some of them are numerically unstable [160]. However, the *pair-wise algorithm* [161] provides much better accuracy. Given $X=A \cup B$, one can determine the variance in one pass by computing sample mean value μ and the *sum of squares of differences* from μ, denoted as $M_{2,i}$, for A and B independently. These values can later be combined to calculate the overall standard-deviation value using the following formulas:

$$M_{2,A} = \sum_{i=1}^{n_A} (x_i - \mu_A)^2 \qquad (3)$$

$$M_{2,X} = M_{2,A} + M_{2,B} + \delta^2 \frac{n_A \times n_B}{n_X} \qquad (4)$$

$$stdev = \sqrt{\frac{M_{2,X}}{n_X}} \qquad (5)$$

This algorithm can be executed in two phases, and the first phase can be parallelised. Therefore, the statistical reduction process is designed as a *split-phase* operation as below (Figure 4-14). As it turns out, this design actually maps well onto the client-server architecture of the host debugger, Guard, as discussed in Section 4.1.5 [130].

Parallel: calculate a set of primary statistics from the input dataset. Examples are sample size, minimum, maximum, mean etc. This phase is embarrassingly parallel as it does not involve any inter-process communication; and

Aggregation: assemble a collection of primary statistics to form a full statistical model. This contains both primary and derived statistics: for instance, variance or standard-deviation value.

The *split-phase* scheme implemented in this work resembles the *MapReduce* programming paradigm [162]. In the MapReduce paradigm, the *Map* step involves the partition of sub-problems onto multiple worker nodes, and these worker nodes perform the assigned tasks in parallel before passing the answer back to its master node. These activities are carried out by the *Parallel* phase in our scheme. In addition, both the *Reduce* step in MapReduce and the *Aggregation* phase in the split-phase scheme take the answers to all the sub-problems and couple them in some way to get the output. It would be interesting

to evaluate whether existing MapReduce runtimes could be used to support this work, although the infrastructure for parallel evaluation and reduction already exist in Guard.

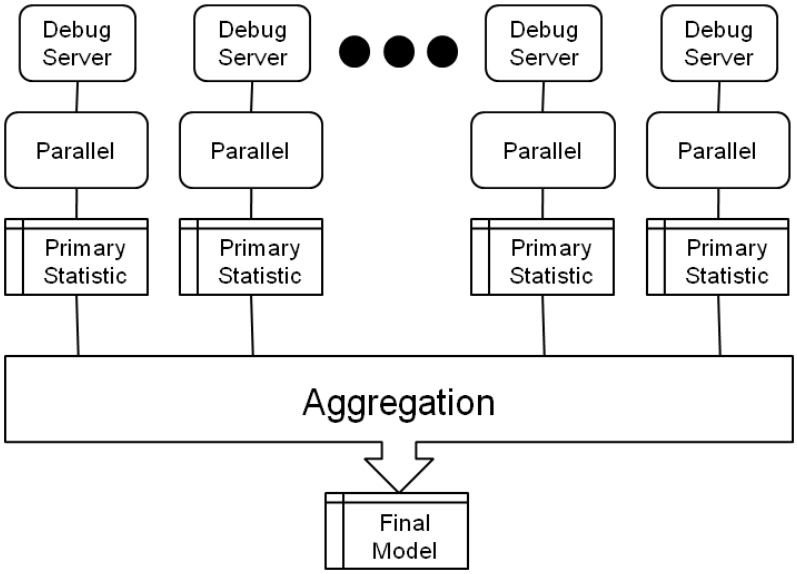


Figure 4-14: Split-phase Statistical Operation

4.4.2 User-defined Abstract Data Models

Using statistical assertions to verify patterns extracted from large runtime data structures requires users to create arbitrary data models. For example, to assert that elements in a dataset follow a Gaussian distribution, the histogram constructed using the target dataset must be compared against a histogram built with random numbers, generated from the Gaussian distribution. Therefore, this work extends the current debugging language in order to support the creation of large user-defined datasets. These can then be transformed into statistical models or distributions. Importantly, the computations required for constructing such datasets and creating accurate distributions are often expensive and time consuming. Therefore, it is desirable to parallelise this operation as well. First, the debug client can inform the debug servers to create user-defined datasets in parallel. This permits large samples to be populated, which enhances the accuracy of the expected models. Second, the split-phase scheme presented above can be used to reduce the large sample into user-defined models in parallel.

4.5 Scalable Comparison Techniques

As discussed in Chapter 3, when debugging an application with Guard, using the relative debugging paradigm, comparative assertions can be placed on various data structures

between a reference code and a suspect code. The user also describes the data decomposition using a special mapping function. The current implementation of Guard uses a *Copy*, *Combine* and *Compare* scheme, illustrated in Figure 4-15, to evaluate a particular comparative assertion.

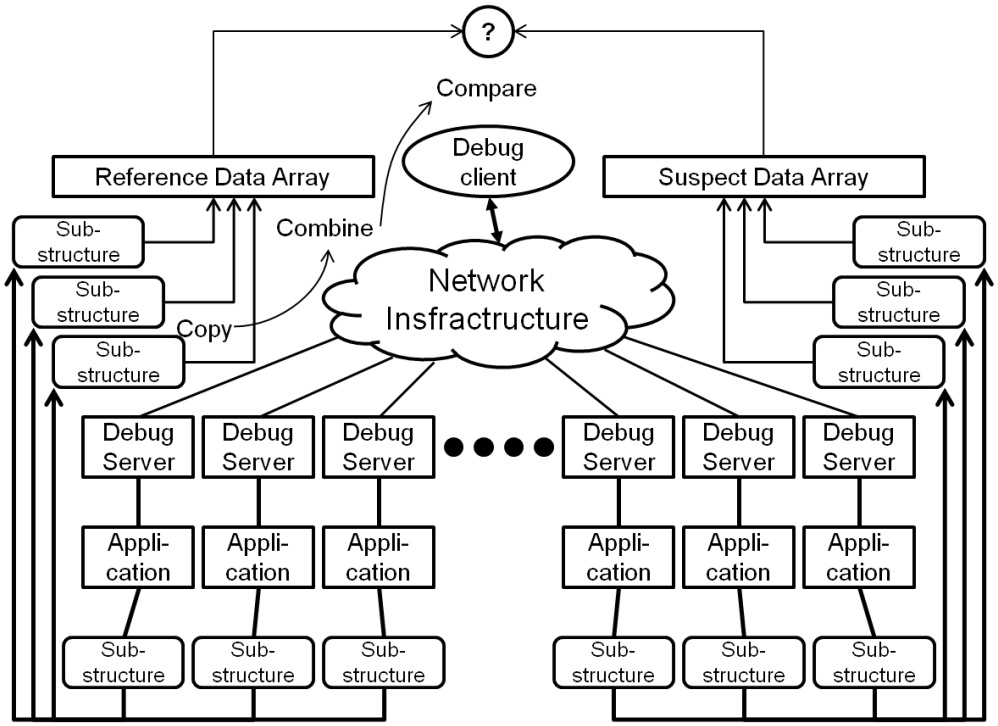


Figure 4-15: Guard's Copy-Combine-Compare Scheme

The process commences when the debugger encounters an assertion breakpoint in the slave processes. Each debug server performs expression evaluation in parallel to retrieve their portion of the variable. Following that, the sub-structures are *copied* to the head node, where they are *combined* into a single data structure (possibly using the blockmap algebra). If both the reference and suspect programs run on the same system (possibly using different compute nodes), then the two data structures are reconstituted in the head node, and then *compared* locally. This scheme works well for applications that produce a small amount of data. As criticised previously, this comparison scheme is impractical if the data transferred to the head node exceeds its memory capacity. Further, assuming the head node can hold that much data, the Combine and Compare phases still suffer severe performance issues because they are carried out sequentially. Finally, most communication networks do not have appropriate mechanisms to combine the data in the switch, thus, retrieving sub-structures is likely to be performed sequentially. This makes the overall approach relatively slow. This research investigates two advanced comparison techniques in order to improve the system performance on large machines.

4.5.1 Hash-based Comparison Method

To solve the problem of too much data at the combination and comparison phases, one desirable approach is to reduce the size of the distributed data structures before they are collected at the head node. In order to do that, a hash function can be used to condense each sub-structure into one or more smaller signatures. As a result, instead of bringing all the raw sub-structures back to the head node from the backend debug servers, the generated signatures can be transferred instead. After that, the signatures can be combined and compared. If the key data structures from both the reference program and the suspect program are equivalent, the hash signatures will also be identical. However, if there is any mismatch between the key data structures, the signatures will also differ, and this can be detected much more quickly than comparing the entire data structure.

Even though hashing is appealing, there are a number of practical issues that need to be resolved. First, all hash functions have the potential to generate false results (that is hash collisions) because multiple values are mapped onto a single signature. Second, the hash function must handle a variety of scalar data types, ranging from integers to floats. Hashing integer data is relatively straight forward because the same integer will always generate the same signature value. However, two floats that are close might be considered the same value within some tolerances. In Guard, this is resolved by a tolerance, under which floating-point values are considered the same. However, two slightly different floats will generate wildly different signatures, and thus a simple minded tolerance scheme cannot be implemented. Finally, the reference and suspect programs might invoke different numbers of processes, thus hashing the data held in each processor will generate different signatures even when the data is the same. Accordingly, we need to devise a hashing scheme that is insensitive to the number of processes. These challenges are dealt with in the next three sessions, respectively.

4.5.1.1 Desirable Hash Function Properties

A number of good hash functions and algorithms have been devised over the years [20, 21]. Each hash function has a set of properties and designed goals. Some particular properties suit data comparison purposes, and are listed here. In particular, the hash function should:

- minimise the number of collisions. This is important because a high collision rate will indicate that arrays have the same contents when they actually do not, and will confuse the debugging process;
- distribute signatures uniformly;
- have an *avalanche effect* to ensure that the output varies widely (e.g. half the bits changed) for a small change in input (e.g. changing a single bit). This is important because data may be skewed and only use a small part of the potential range of input values. This will allow us to detect changes even when data is skewed in this way;
- minimise the size of the signature;
- detect permutations on data order within a structure. This is important because index permutation is a common error when a program code is parallelised and evolved.

According to Mulvey [163], hash functions such as the Bob Jenkins hash function, FNV hash functions, the SHA hash family, and the MD hash family all exhibit the above properties [163, 164]. They have been evaluated systematically as discussed in [165-168]. As a result, Bob Jenkins' hash function, FNV hash function and AP hash function [169] have been chosen because they are relatively simple to implement, work consistently on different data types, and promise efficient performance, while being hardware and platform independent. Regarding the performance in dealing with collisions, a recent application of Bob Jenkins' function, for a similar application, suggests that the collision rate for real datasets might be as low as 10^{-10} when generating a 32 bit signature [168]. Further, *rsync* [170] uses the MD5 hash functions to perform data synchronisation over the network; and the authors argue that the collision probability is too low (2^{-160}) to be nontrivial. In spite of these results, it is of concern that even a small collision probability might make the proposed hash-based comparison scheme ineffective. Accordingly, the probability of collisions in the AP hash function, FNV hash function and Bob Jenkins' hash functions are evaluated to confirm their suitability for data comparison. To match real world conditions, different criteria for selecting test data were applied. For example, various data types including character string, signed or unsigned integers and floating-point numbers were considered. Furthermore, data was also arranged in arrays with different patterns such as random, skewed (increase/decrease incrementally in values), and permuted. The total amount of data used in the evaluation was around 46 Gbytes, distributed into blocks of 1 Kbyte to 1 Mbyte of data. Each block was hashed

individually to generate a 32 bits hash signature. The result of the experiment shows no collisions at all. The detailed description and results of these hash function experiments are provided in *Appendix A - Evaluation of Hash Functions*.

In the unlikely event that the debugger misses detecting an error, it would likely be picked up later in the execution, mitigating the effect of collisions even further. For example, data structures are often manipulated in loops, in which case the assertion fires multiple times. Thus, even if the assertion misses an error the first time a divergence occurs, it would almost certainly detect the error on the next loop iteration.

4.5.1.2 Floating-point Precision

As discussed, *comparison tolerances* allow floating-point numbers to be considered equal if their magnitude of the difference is within the tolerance [25]. Both absolute and relative tolerance schemes were implemented in the past and they have been effective as a technique to remove insignificant differences in data [25]. However, this scheme cannot be used in combination with hashing because hashing is performed on the source data before the two structures are aggregated.

An alternative way of masking insignificant changes is to use a slightly different definition of tolerance. Instead of subtracting two numbers, if floating-point values are pre-rounded (or truncated) prior to comparison, then some of the low order digits will be removed before they are subtracted. Thus, two numbers that are close will round to the same number, and thus the hash values will be the same. However, it is noteworthy that, rounding is not the same as using a fixed tolerance. For example, given that a user wants two numbers to be equal only to the nearest 0.001, consider the following floating-point numbers: 0.1239, 0.1244, and 0.1245. Clearly, these are rounded to 0.124, 0.124 and 0.125 respectively. Thus, the first two numbers will be considered within tolerance, but the latter two will not.

In spite of this difference, this study has adopted this approach because it provides a tolerance-like behaviour and can be combined with hashing. In either scheme, the choice of tolerance is somewhat arbitrary and it is an experimental parameter used at runtime.

4.5.1.3 Decomposition Independence

A simple minded implementation of hashing computes a single signature for the data held by each processor, then combines and compares these in the head node. However,

in order for this scheme to work, each of the programs must use the same number of processes and also identical data decomposition functions.

A solution to this problem involves evaluating multiple signatures per processor rather than one, and computing these in a way that is independent of the number of processors and decomposition strategy. By making the number of hash signatures returned by all processes the same in both programs, they can be compared directly.

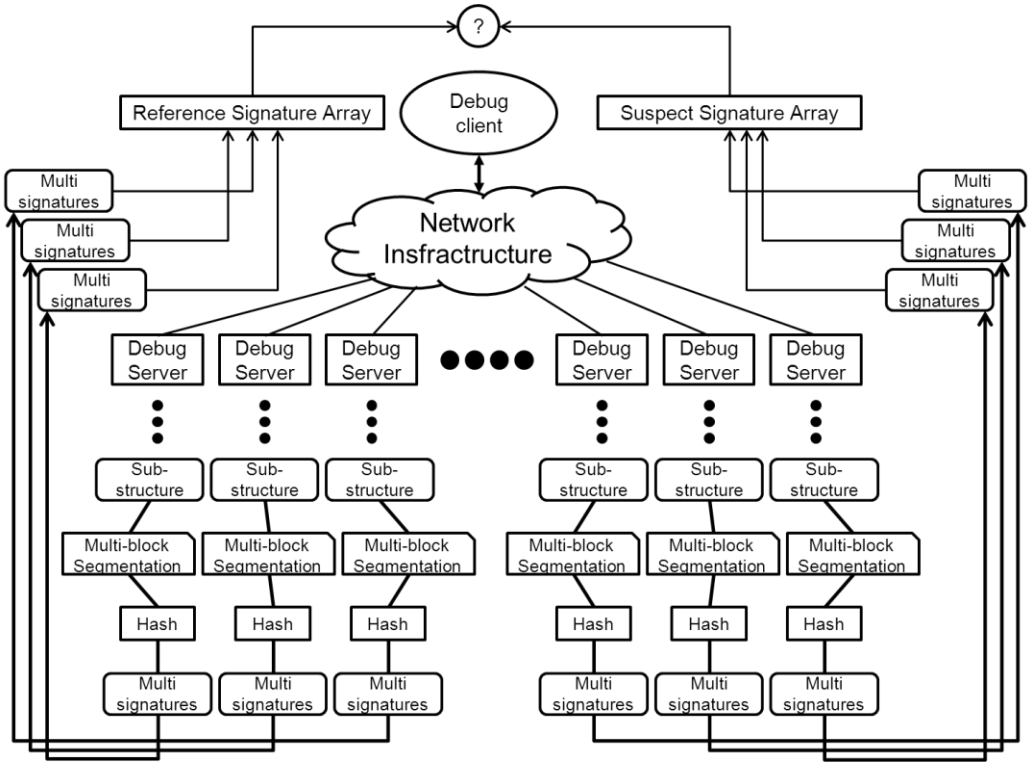


Figure 4-16: Hash-based Comparison Approach

Definition

Because the hash-based comparison scheme depends on the blockmap mechanism to rebuild the global array of signatures, notations used in the blockmapping algebra are reused here. Accordingly, `p` denotes the number of processes in a parallel program. We have a vector $\texttt{data}(d_1,d_2,\ldots,d_k)$ to denote the shape of an array in which `k` is the rank of the array and d_i is the size of i^{th} dimension of the array. For example, with `A=data(100,105)`, A is an array of rank 2 with 100 rows and 105 columns. We also have a vector $\texttt{blockmap}(A,m_1,m_2,\ldots,m_k)$ represent the decomposition function of a given array A. Therefore, `blockmap(A,10,105)` indicates that the global array A is decomposed into 10 sub-arrays of shape `(10,105)`. Accordingly, the following condition is held:

$$\prod_{i=1}^{k} d_i = p \times \prod_{i=1}^{k} m_i \qquad (6)$$

Proposition

Given reference program R with p_R processes and suspect program S with p_S processes, assume both programs generate a global array A of shape `data(`d_1`,`d_2`,…,`d_k`)`.

Suppose that R implements decomposition `blockmap1(A,`b_1`,`b_2`,…,`b_k`)` and S implements decomposition `blockmap2(A,`c_1`,`c_2`,…,`c_k`)`. If `hash_size(A,`x_1`,`x_2`,…,`x_k`)` is a function that takes an array of rank `k` and returns an array of hash signatures where each signature corresponds to a sub-array of shape `(x1,x2,…,xk)`, then the formula to calculate x_i`(1≤i≤k)`, so that after hashing both programs end up with exactly the same array of hash signatures, is as follows:

$$x_i = \gcd(b_i, c_i) \qquad (7)$$

Where function `gcd(x,y)` returns the greatest common divisor of x, y.

In other words, we are interested in finding the greatest overlap between the two disparate blockmap functions. This overlapping area is illustrated in Figure 4-17 below.

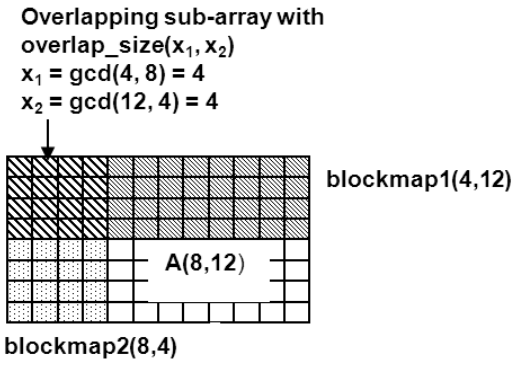


Figure 4-17: Blockmap Overlap

Proof

Let n_{hashR} be the number of hash signatures generated by one reference process and n_{hashS} be the number of hash signatures generated by one suspect process. Then we have:

$$n_{hashR} = \prod_{i=1}^{k} \frac{b_i}{x_i} \text{ and } n_{hashS} = \prod_{i=1}^{k} \frac{c_i}{x_i} \qquad (8)$$

Since there are p_R processes in R and p_S processes in S, the total number of hash signatures produced by R and S are $n_{hashR} * p_R$ and $n_{hashS} * p_S$ respectively. Due to condition (6) above, we have:

$$\prod_{i=1}^{k} b_i \times p_R = \prod_{i=1}^{k} c_i \times p_S \Rightarrow n_{hashR} \times p_R = n_{hashS} \times p_S \quad (9)$$

As a result, using this strategy, both programs will produce the same number of hash signatures regardless of either number of processes or data decomposition scheme.

Example 1

Reference program R and suspect program S have array `A=data(100,105)`. R has p_R=10 and performs `blockmap1(A,10,105)`. S performs `blockmap2(A,5,105)` with p_S=20. Using the formula (7), both programs will compute function `hash_size(A,`x_1`,`x_2`)` with:

x_1 = gcd(10,5) = 5

x_2 = gcd(105,105) = 105

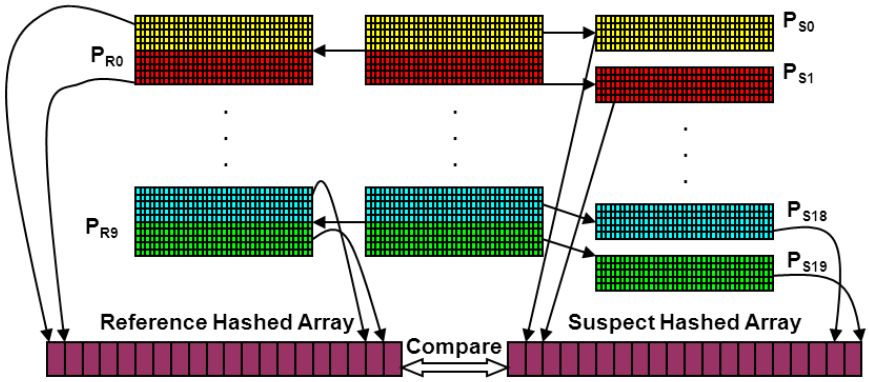


Figure 4-18: Hash Signatures Comparison - Example 1

Therefore, since each reference process holds a data array of shape `(10, 105)`, these processes will each produce two hash signatures. On the other hand, each suspect process only needs to produce one signature. In other words, since data decomposed to each reference process is twice as much the data distributed to each suspect process, reference processes have to generate twice the number of signatures. Upon completion, it is expected that two arrays with shape `(1,20)` of hash signatures will be compared directly at the client side.

Example 2

Reference program *R* and suspect program *S* have an array `A=data(100,105)`. R has p_R=10 and performs `blockmap1(A,10,105)`. S performs `blockmap2(A,100,15)` with p_S=7. Similarly, both program compute function `hash_size(A,`x_1`,`x_2`)` where:

x_1 = gcd(10,100) = 10

x_2 = gcd(105,15) = 15

Consequently, since each reference process holds a data array of shape `(10,105)` and the function `hash_size` above maps one signature to a sub-array `(10,15)`, each process produces `105/15=7` hash signatures. Similarly, each suspect process will need to produce `100/10=10` signatures. However, after collecting the hashes from each program, we can see that R has a `(7,10)` array of signatures while S has a `(10,7)` array of signatures. The relevant *Index Permutation* [25] operation will be performed on one of the arrays so that two arrays of signatures of shape `(10,7)` will be compared at the client side.

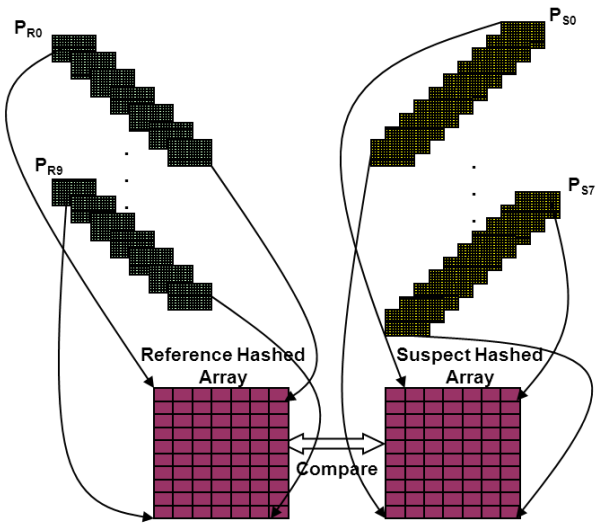


Figure 4-19: Hash Signatures Comparison - Example 2

Example 3

Reference program R and suspect program S have an array `A=data(32,50)`. R has p_R`=8` and performs `blockmap1(A,4,50)` while S performs `blockmap2(A,32,1)` with p_S`=50` (e.g. (*,cyclic) distribution). Using the above formula, both programs will have to compute function `hash_size(A,`x_1`,`x_2`)` where:

x_1 `= gcd(4,32) = 4`

x_2 `= gcd(50,1) = 1`

Hence, since each reference process holds a data array of shape `(4,50)` and the function `hash_size` above maps one signature to a sub-array `(4,1)`, each reference process produces `50/1=50` hash signatures. Similarly, each suspect process will need to produce `32/4=8` signatures. As expected, a same number of hash signatures is produced by both programs. Upon collection, these signatures are arranged using the blockmap information into comparable signatures arrays.

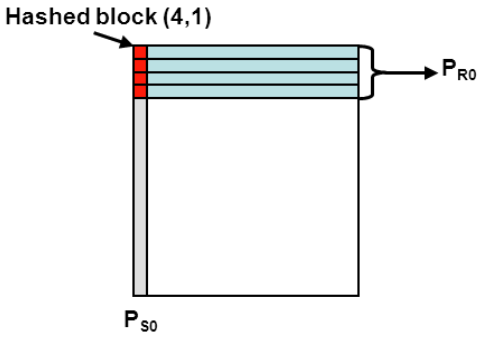


Figure 4-20: Hash Signatures Comparison - Example 3

Efficiency

The above strategy ensures that there is always an overlapping region between two arbitrary blockmap functions. However, if the reference program and the suspect program deploy blockmap functions such as *(*, cyclic)* and *(cyclic, *)*, the overlapping region may be quite small – at the limit, only a single cell. In this case hashing would add no value because each hash function would only hash one array element. On the other hand, it is unlikely that any parallel program would be efficient if it only allocated a single row or a single column to a given processor, and thus, we can assume that in any real program on any real machine there are likely to be multiple rows or columns assigned to a given processor.

4.5.2 Direct Point-to-Point Comparison Method

Even though the hash-based comparison solution described in Section 4.5.1 is effective, there are occasions when raw data comparison is preferable. First, since hash signatures are compared instead of raw data, there is a probability of false positives (even though it can be made arbitrarily small). Second, while the hashing phase is completely parallelised, the comparison phase is still conducted sequentially in the head node, limiting the speedup at high core counts.

Another approach to improve the scalability in evaluating comparative assertions while avoiding losing data integrity through a reduction process such as hashing is to parallelise the raw data comparison phase. Therefore, instead of assembling sub-structures sequentially at the client node, those sub-structures can be communicated directly between debug servers to perform the comparison phase in parallel. This solution promises significant speed increases; because not only the combination phase can be omitted completely, but also both the data communication phase and the data comparison phase can be parallelised. Even though the technique is appealing in terms of

performance, there are some critical issues in the implementation that are worth serious discussion.

In this section, we discuss a *direct point-to-point* (P2P) comparison technique that parallelises the data comparison across a set of debug servers (shown in Figure 4-21). The scheme executes in three phases. First, the client process notifies remote debug servers about the way data has been distributed in the parallel application. Second, the debug servers analyse the data decomposition and determine the communication patterns required. Using communication information, each debug server transfers (or receives) data to (or from) other debug servers and comparison is carried out in parallel. Finally, the individual comparison results are collected, combined and presented in the head node. The scheme is highly scalable because the comparison task can be performed in parallel; and the amount of data to transfer to the head node is relatively small. However, challenges must be addressed if the technique is to be feasible. For example, if the two programs use the same decomposition strategy, and the same number of processors, then it is only necessary to send the sub-structures from, say, the reference processes to the corresponding suspect ones. On the other hand, it becomes more difficult when the decompositions and number of processors differ.

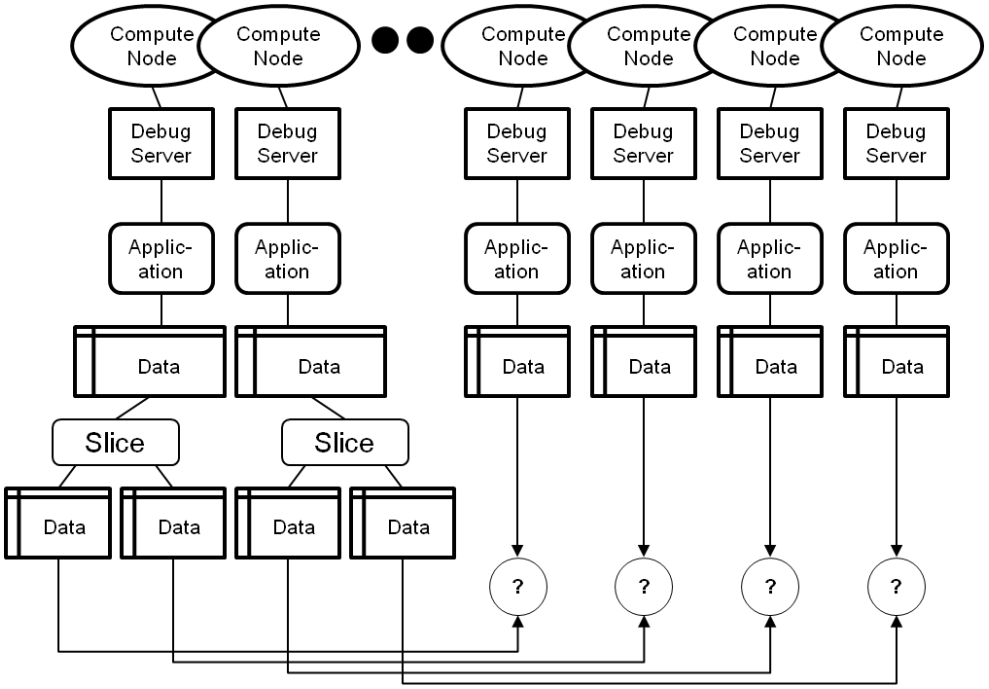


Figure 4-21: Point-to-Point Comparison Approach

This work introduces new components into the current debugging architecture, including the *Decomposition Analyst* and the *Point-to-Point* API (depicted in Figure 4-22). We discuss

how these components can deal with the challenges addressed above respectively. Importantly, we focus on a direct comparison scheme that is insensitive to the number of processes.

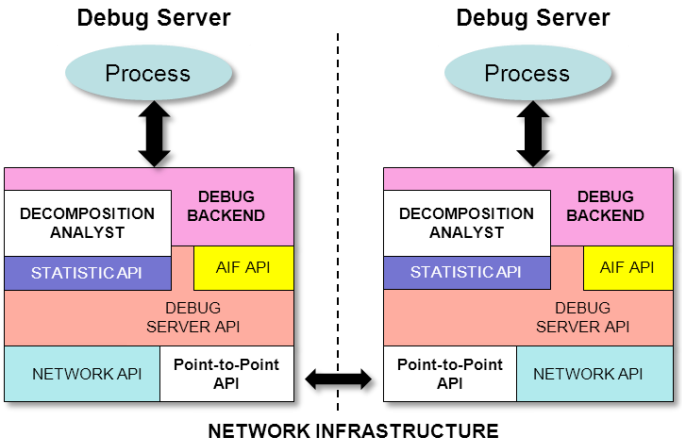


Figure 4-22: Point-to-Point Architecture

4.5.2.1 Decomposition Analyst

As stated above, if the two programs use the same number of processes and deploy the same data decomposition scheme, parallel P2P comparison is relatively straightforward. However, quite often relative debugging is used when the target program is changed to work on a larger machine. Thus, the number of processes in each program will not be the same.

Objectives

The *Decomposition Analyst* performs two important tasks. First, it examines the decomposition definitions employed by the reference and suspect programs. This decomposition scheme is described in the blockmap function definitions provided by the user. Second, it generates the mapping between data segments held by various peer processes, for the comparison purpose. Such mapping information allows debug servers to independently communicate with particular peer debug servers from the other program, transfer the required data blocks, and then perform direct comparison without going through the client (head) node. The Decomposition Analyst is built as part of each debug server. Therefore, this analysis process is carried out in parallel and is highly scalable.

The reason for analysing the differences between decomposition schemes is to identify the greatest overlapping area between these two blockmap functions. Using this information, each sub-structure held by an individual process can be divided into a

number of equal size sub-blocks. These blocks can be directly compared against each other as illustrated in Figure 4-23. The formal proof for this algebra is presented in the hashing technique discussion. Below are some examples of using *Decomposition Analyst* to build P2P mapping information.

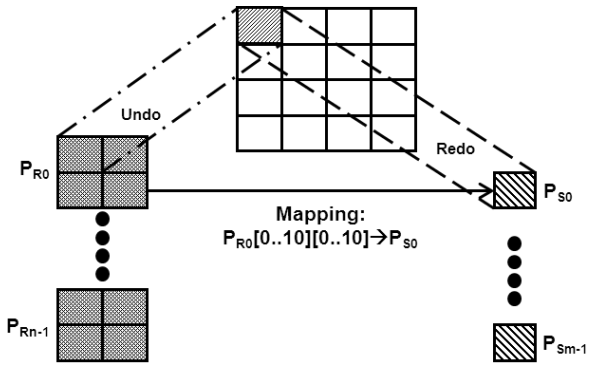


Figure 4-23: P2P Mapping Process

Example 1

Reference program R and suspect program S have array `A=data(100,105)`. R has p_R=10 and performs `blockmap1(A,10,105)`. S performs `blockmap2(A,5,105)` with p_S=20. Using the above formula, both programs indicate the greatest overlapping area (x_1,x_2) where:

x_1 = gcd(10,5) = 5

x_2 = gcd(105,105) = 105

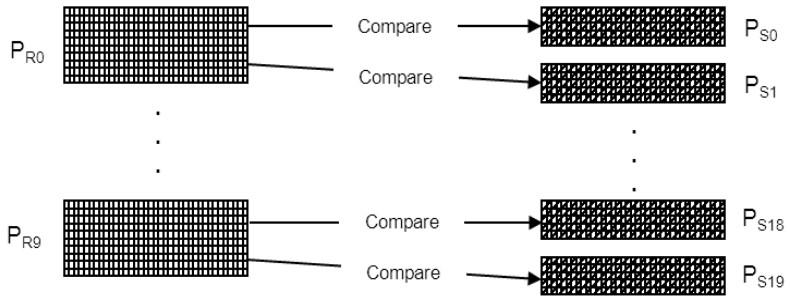


Figure 4-24: P2P Mapping - Example 1

Therefore, since each reference process holds a data array of shape `(10,105)`, these processes will each produce two sub-blocks. On the other hand, the sub-structure held by each suspect process does not need to be divided. The mapping algebra produces the following mapping for direct P2P comparison:

```
R[0](0..0,4..104)→S[0]
R[0](5..0,9..104)→S[1]
…
```

R[9](90..0,94..104)→S[18]
R[9](95..0,99..104)→S[19]

Example 2

Reference program R and suspect program S have an array A=data(100,105). R has p_R=10 and performs blockmap1(A,10,105) while S performs blockmap2(A,100,15) with p_S=7. Using the above formula, both programs indicate the greatest overlapping area (x_1,x_2) where:

x_1 = gcd(10,100) = 10
x_2 = gcd(105,15) = 15

Consequently, each reference process slices its data array into 7 sub-arrays of size (10,15), while each suspect process will need to produce 100/10=10 such sub-arrays. Below is the mapping information required for direct P2P comparison.

S[0](0..0,9..14)→R[0]
S[0](15..0,9..29)→R[1]
S[0](30..0,9..44)→R[2]
S[0](45..0,9..59)→R[3]
S[0](60..0,9..74)→R[4]
S[0](75..0,9..89)→R[5]
S[0](90..0,9..104)→R[6]
…

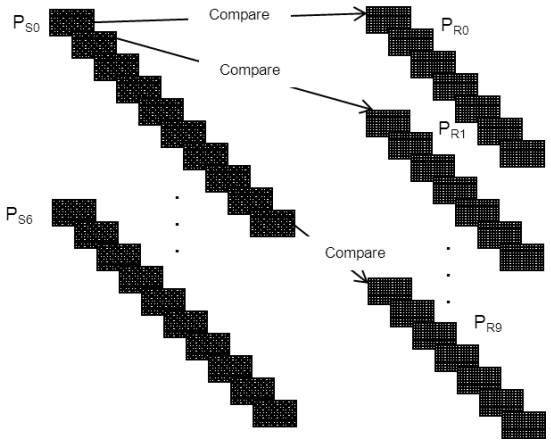


Figure 4-25: P2P Mapping - Example 2

Example 3

Reference program R and suspect program S have an array A=data(80,100). R has p_R=8 and performs blockmap1(A,10,100) and S performs blockmap2(A,20,25) (e.g.

(block,block) distribution) with p_S=16. Using the above formula, both programs indicate the greatest overlapping area (x_1,x_2) where:

x_1 = gcd(10,20) = 10

x_2 = gcd(100,25) = 25

Hence, each reference process breaks its overall data structure of size (10,100) into 4 blocks of size (10,25). Similarly, each suspect process generates 2 blocks. As expected, the total number of blocks created by all reference processes is the same with the total number of blocks generated by all suspect processes. Using the mapping algebra, each process in both reference program and the suspect program can be mapped as follows:

R[0](0..0,9..24)→S[0]
R[0](0..25,9..49)→S[1]
R[0](0..50,9..74)→S[2]
R[0](0..75,9..99)→S[3]
R[1](0..0,9..24)→S[0]
R[1](0..25,9..49)→S[1]
R[1](0..50,9..74)→S[2]
R[1](0..75,9..99)→S[3]
…

4.5.2.2 Point-to-Point (P2P) API

Apart from the Network Infrastructure layer, a P2P network layer needs to be integrated for direct communication between remote debug servers. The implementation details of this component are presented in Chapter 5.

4.5.2.3 Efficiency

This comparison strategy suffers the same efficiency issue as the hash-based comparison scheme. While it ensures that there is always an overlapped region between two arbitrary blockmap functions, the overlapping region could be as small as one cell (if the reference and suspect programs deploy blockmap functions such as *(*, cyclic)* and *(cyclic, *)*). In this case, there will be an enormous number of mapping as well as data transfers between the two programs. Therefore, it would add no value compared to the traditional comparison scheme.

4.6 Summary

The assertion-based debugging framework proposed in this thesis requires a number of key components compared to traditional parallel debuggers. First, it needs an architecture that supports the operation of each individual debug server. Second, it needs a mechanism to execute user-defined assertions in parallel. These requirements are delivered by a parallel debugger named Guard, which is used as the foundation for the debugger implemented in this research. Prior to this work, Guard has been developed to allow comparison between a sequential program and a small parallel program. Such implementation includes:

- the functions for a frontend client which accepts and processes user's commands. To evaluate advanced commands such as assertions, advanced components such as Graph Compiler/Interpreter are integrated with the client;
- the control of multiple backend debug servers. A debug server is capable of processing requests from frontend client and query GDB for debugging information and runtime data;
- the use of a network communication layer which supports different network protocols such as traditional socket and MRNet [147];
- and an Architectural Independent Format (AIF) library;

To support the new types of ad-hoc debug-time assertions proposed in Chapter 3, several unique components must be integrated into the existing architecture. The components and functionality listed below are the new extension developed as part of this research.

- the blockmap algebra to provide the ability to reconstruct distributed data structures;
- two scalable comparison techniques that leverage the use of comparative assertions in highly parallel environment; and
- an extensible data reduction framework which allows users to define and execute arbitrary reduction routines in parallel;

Chapter 5 outlines the implementation details of these components.

Chapter 5

Implementations Details

As detailed in Chapter 4, a data-centric parallel debugger using assertions must provide a range of functions to support various types of ad-hoc debug-time assertions for debugging in a parallel environment. These functions can be categorised as data decomposition support, advanced parallel data reduction support and support for scalable comparison of large datasets. In addition, a data-centric debugger must also permit conventional debugging activities, and be able to control multiple processes independently. As a result, an existing parallel debugger called Guard is utilised as the groundwork. This chapter will describe the specific implementation details. In particular, it presents the enhancement of several existing Guard components including the Dataflow Compiler, the Dataflow Engine, and the debug servers. In addition, key technologies such as the blockmap algebra, an extensible statistic toolbox, and the machinery for hash-based and direct P2P comparison schemes are developed.

5.1 Extension of Existing Guard Components

This section describes how the existing Guard's infrastructure has been used to build a proof of concept for an assertion-based parallel debugger. The focus is on the expansion of the existing assertion syntax, the enhancement of the Dataflow Compiler and Dataflow Engine, and the upgrade in functions of the backend debug server.

5.1.1 Assertion Syntax

More comprehensive assertions require the comparison between variables and constants, between content of arrays and distribution models, or between histogram formed by one array and histogram constructed from another array. This requires more powerful assertion syntax, based on the syntax discussed in Section 4.1.3. Accordingly, extra comparison operators and computation operators have been added. We also need to support the creation and the evaluation of *user-defined* variables (i.e. as debugger variables). In this work, an expression in an assertion-based debugger is defined using the following grammar (Figure 5-1):

```
E ::= E <compute_op> E
     |constant
     |user_defined_variables
     |<reduce>(E)
     |<proc_set>::<gdb_var_expr>@<source_file>:<line_num>
<compute_op> ::= [+,-,*,/]
<compare_op> ::= [=,>,<,>=,<=]
```

Figure 5-1: Enhanced Assertion Syntax

An assertion is therefore defined as:

`assert` E_1 `<compare_op>` E_2

where `<reduce>` can be user-defined or built-in reduction function

E_1 and E_2 are constructed by the grammar above.

5.1.2 Dataflow Compiler and Dataflow Engine

Currently in Guard, the Dataflow Compiler is responsible for converting a user-defined assertion into low-level graph description while the Dataflow Engine is used to interpret the graph and execute the assertion at runtime. In this research, the design and

implementation of both components are reused. For example, the graph compilation procedure where assertions and control statements are converted into command nodes; and the graph execution process in which nodes are executed sequentially at the frontend, remains unchanged. However, further enhancement is required to accommodate the extended assertion syntax, the new reduction techniques and also the blockmap engine as highlighted in Chapter 4. Figure 5-2 below is an example of an abstract dataflow graph generated for the assertion beneath.

```
set reduce sum
assert $a::p_array@"par.c":34 > 50000
```

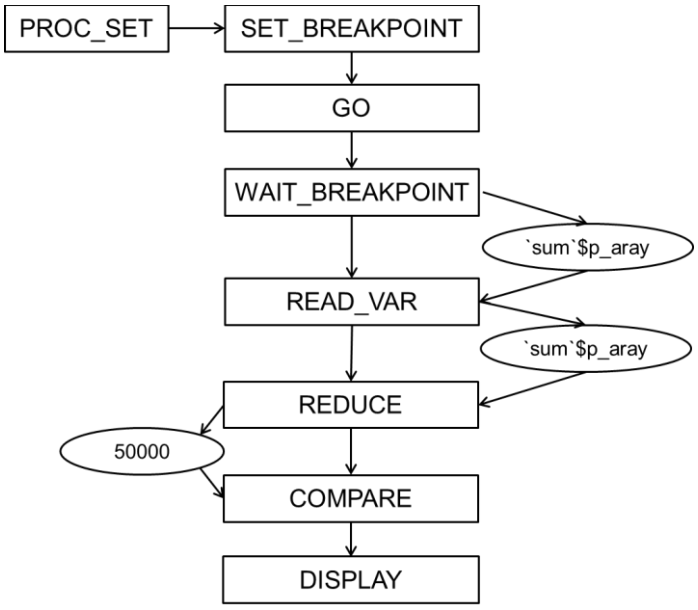


Figure 5-2: Abstract Assertion Dataflow Graph

5.1.2.1 Accommodating Extended Assertion Syntax

In the current implementation, a dataflow graph is generated using a number of standard graph templates. These templates are supplied with information derived from the assertion statements and are used to specify the nodes in the graph. The primary templates are the `ASSERT` template and the `EXTRACT` template which define the process of capturing and comparing runtime data to evaluate an assertion [25]. These are illustrated in [25]. The special symbols ⊗ and ⊕ represent a synchronisation node and a merge node in the graph respectively [25]. Complex operations such as `COMPARE` or `DISPLAY` are encapsulated into sub-graphs and presented as boxes in the diagram for simplicity purpose.

We implement the extension of assertion syntax in the graph description and the `ASSERT` template. This requires the expression parser to be modified to accept constants and local

debugger variables. The `ASSERT` template presented in Figure 5-3 is enhanced to accommodate the use of constants and user-defined variables in an assertion statement. In addition, different types of comparison operation can also be invoked as well using this template. Therefore, an assertion statement such as:

```
assert $p::v1@"file1.c":35 > 10
```

will result in the generation of the `ASSERT` description:

```
ASSERT($p, "v1", "file1.c", 35, '>', 10)
```

The basic `EXTRACT` template takes three parameters consisting of a variable name, plus file name and line number to define a breakpoint location. This information is used in a number of sub-graphs including `SET_BREAKPOINT`, `GO`, `WAIT_BREAKPOINT` and `READ_VAR`.

Because this work reuses the fundamental implementation of the Dataflow Compiler and the Dataflow Engine, the support for multiple ad-hoc assertions remains effective. Templates for dealing with multiple assertions are examined in more detail in [25].

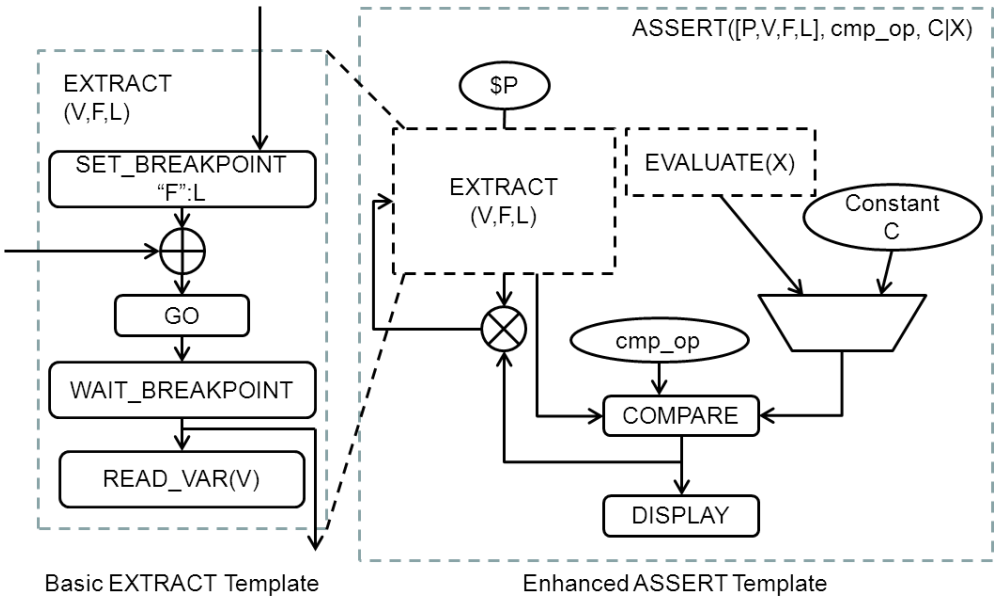


Figure 5-3: Enhanced ASSERT Template

5.1.2.2 Introducing Deferred Debugger Variables

There are two main types of variables in the current design of Guard. These are the *debugger variables* (variables that are defined by the debugger or by the user manually during the debugging process) and the *program variables* (variables that are extracted from the debug programs). Until now, all debugger variables were created and evaluated locally

by the frontend client. However, for advanced assertions such the statistical assertions, we support the creation of user-defined data models as local debugger variables. If such models are evaluated at the frontend, it could become a bottleneck because the process requires the population of large datasets. To deal efficiently with this new type of user-defined debugger variable, only the definition of a variable is stored at the client side while the content is populated by the backend debug servers. This step is performed to support the parallel execution of the assertions using the available computing resources that sit idly at the breakpoints. We call this new debugger variable the *deferred debugger variable*.

Accordingly, a new `EXTRACT&EVALUATE` template is introduced to evaluate the *deferred debugger variables*. For an assertion statement that involves the use of deferred debugger variable, the assertion graph is compiled as shown in Figure 5-4. The `EXTRACT&EVALUATE` template takes four parameters instead of three parameters like the basic `EXTRACT` template. These consist of a program variable `V`, breakpoint location (`F` and `L`), and a deferred debugger variable `X`. Debug servers, upon receiving an extract and evaluate request via the new template, will not only perform runtime data extract but will also populate partial content of the variable as defined by the debugger variable `X`. Therefore, data returns from a remote debug server will be a structure with two datasets.

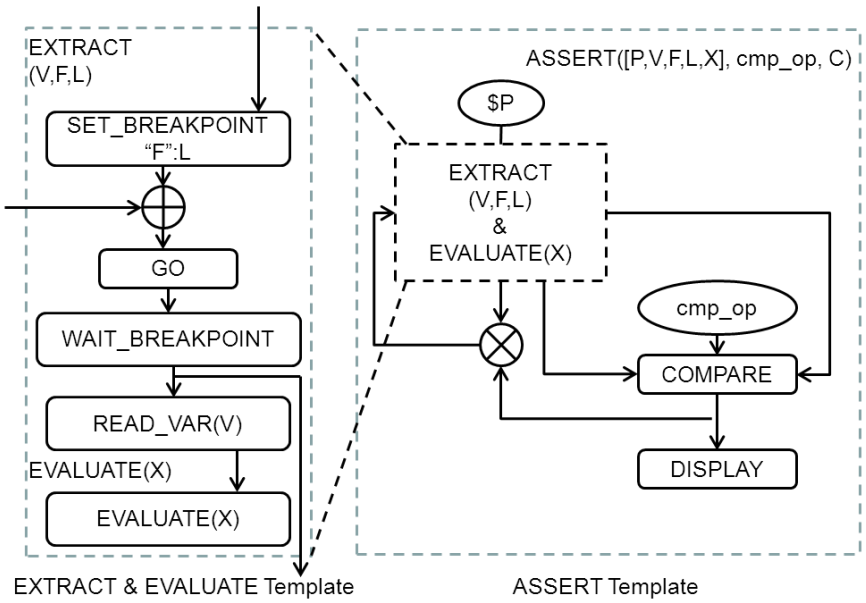


Figure 5-4: New EXTRACT & EVALUATE Template

5.1.2.3 Supporting of Blockmap and Split-phase Reduction Operations

Regarding the integration for new components such as the data decomposition engine or the parallel data reduction engine, a set of new command nodes are introduced. These include the blockmap related nodes, data reduction operation nodes, and user-defined function nodes.

First, while the blockmap auxiliary nodes perform a range of activities including capturing the blockmap description provided by the user, translating the details into a compact description, and attaching those details to the assertion expression, the central `BLOCKMAP` node is used to execute the blockmapping algebra presented in Chapter 4. In terms of graph compilation and execution, the `BLOCKMAP` node is used and executed in similar fashion to the current `MAP` node [25].

Second, supporting parallel reduction of runtime data requires a template that not only extracts the data but also encapsulates the reduction description so that debug servers can perform reduction individually. Accordingly, this implementation introduces a new `EXTRACT&REDUCE` template as depicted in Figure 5-5.

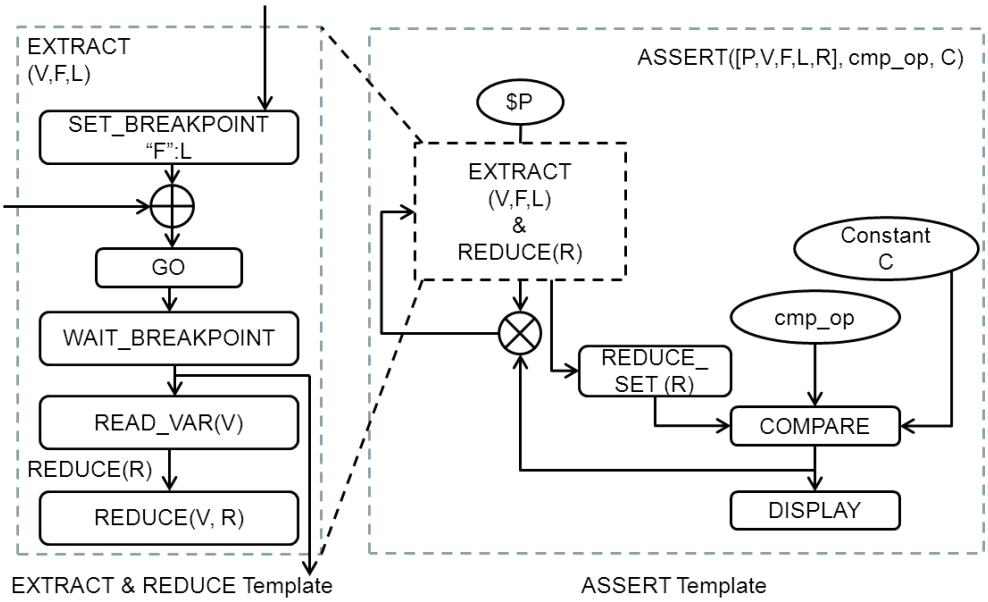


Figure 5-5: New EXTRACT & REDUCE Template

Similar to the EXTRACT&EVALUATE template, EXTRACT&REDUCE template has an extra parameter that describes the reduction function and performs the function after the raw runtime data is extracted from the process. A new REDUCE_SET sub-graph is introduced at the ASSERT template which performs further reduction at the client site, depending on the reduction function. In the next subsection, we will further explain the

use of these nodes when we discuss the expansion of the debug server's functions. Furthermore, to support users in creating and executing user-defined statistical routines, the same class of command nodes and templates used for general data reduction requests is employed. However, to perform the split-phase operation, further details are encapsulated within the backend requests. We also shall examine further the implementation details of the split-phase operation in Section 5.3.1.

5.1.2.4 Supporting Scalable P2P Comparison Scheme

In order to support comparative assertions for multiple parallel programs, the graph is further developed to cope with two parallel programs instead of being limited to comparison between a sequential and a parallel program, as in the previous version of Guard [25].

Chapter 4 describes two advanced comparison techniques including hash-based comparison and P2P comparison, in order to improve the scalability of the comparative assertions. While the hash-based comparison scheme can be described as a simple data reduction scheme using a hash function (hence using the `EXTRACT&REDUCE` template is sufficient), the P2P comparison scheme requires further enhancement of the Dataflow Compiler. Specifically, a new template called `EXTRACT&COMPARE` is developed as in Figure 5-6. Similar to the other new templates such as `EXTRACT&REDUCE` or `EXTRACT&EVALUATE`, the `EXTRACT&COMPARE` template takes two more parameters called `bm1` and `bm2` which are blockmap descriptions for the two program variables involved in the assertion statement. This is to inform debug servers of the way the two datasets are decomposed so that P2P mapping can be computed and direct communication and comparison of sub data structures can be carried out. The `P2P_COMPARE` sub-graph comprises of activities that a debug server needs to conduct as part of the P2P process. Note that the `COMPARE` sub-graph is no longer needed in the `ASSERT` template as the real comparison is carried out at the backend. Details of how the P2P comparison activities are implemented are presented later in this chapter.

Another important enhancement in the graph is the synchronisation in retrieving runtime data between the two `EXTRACT` templates. Earlier versions of Guard extracted runtime data from the reference program and the suspect program sequentially. This scheme is slow but acceptable since the amount of data captured was small, and the comparison occurred at the client site. However, it does not work with the new P2P comparison scheme because data from both programs must be available at the same time for the P2P

comparison to proceed. Therefore, the `ASSERT` template is enhanced to make sure data from both programs are extracted and compared remotely before the graph proceeds further.

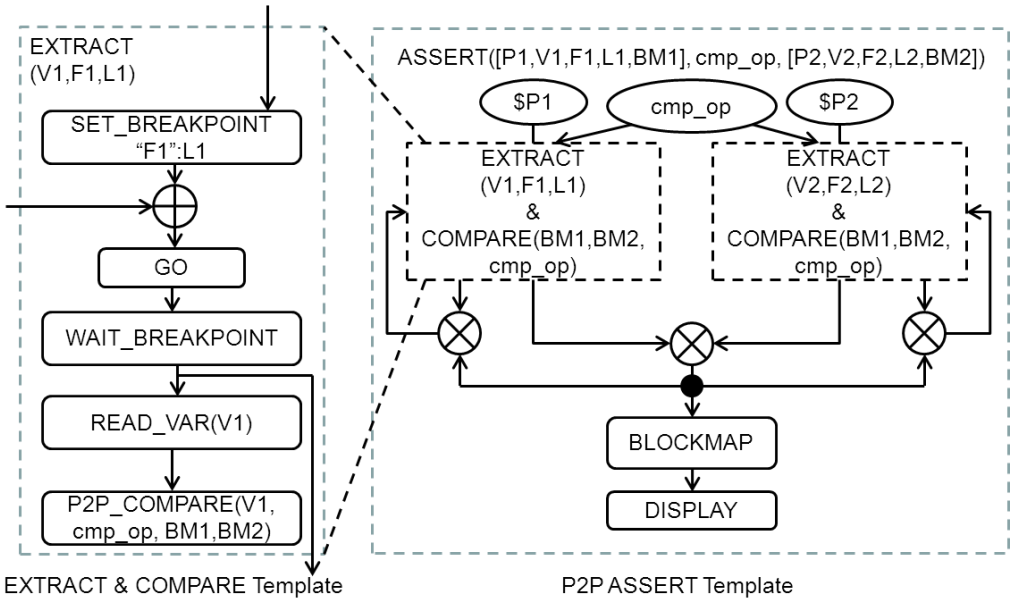


Figure 5-6: EXTRACT & COMPARE Template

5.1.3 Debug Server

When implementing ad-hoc debug-time assertions of the type discussed in this thesis, the author has extended the functions of the debug servers used in the earlier version of Guard [7]. In particular, remote debug servers now execute the following operations: data retrieval, data reduction, and data transmission to the frontend client.

5.1.3.1 Data Retrieval

Through the debug server, the backend debug engine such as GDB can be instructed to retrieve runtime data from program variables. The data is directly converted into the AIF format [25]. This procedure allows the comparison of datasets obtained from disparate computing platforms.

5.1.3.2 Data Reduction

In contrast to comparative assertions where data divergence between two datasets is the focus, the types of ad-hoc debug-time assertion supported in this work often explore the semantics acquired from the datasets. For that to work, general reduction functions such as sum, product, min, max, and average are used. Hence, through the client interface, users can specify which type of reduction function is used for different assertions. To

enable a reduction function, this work introduces the `set reduce` command. Below is a simple example of how to trigger the summation function in our debugger.

```
set reduce sum
```

The debug server performs the data reduction function before the result is transferred back to the client. To notify the debug server which type of operations is triggered, the frontend client is enhanced to deliver extra information to the debug servers. This is actually specified in the dataflow graph as described in Section 5.1.2. The extra information includes the reduction function's ID and data decomposition information. As a result, the remote debug servers have sufficient information to perform the reduction operation in parallel and transmit the reduced dataset back to the client. A reduction function can be a built-in function or a user-defined function. The mechanism to invoke and deploy a user-defined function is discussed in more details in Section 5.3.

5.1.3.3 Data Transmission

After completing the reduction operation, debug servers transfer the results to the client. Different assertions require different handling in the client and in the servers. For example, in Figure 5-2, since the assertion calculates the sum of all elements in array `p_array`, upon receiving sub-total values from remote debug servers, the client performs the final sum reduction before comparing the result to 50,000. In this case, we can assert the sum of all elements in array `p_array` to be greater than 50,000 even though `p_array` has been decomposed across many processing units.

5.2 Data Decomposition

In this work, the debugging language is extended to cater for the definition of the blockmap function. A user can define several blockmap functions which describe different decomposition schemes, then later use them in assertions. Each blockmap function is constructed using three sub-define functions: *distribute, data*, and *grid*. Function *distribute* describes the decomposition scheme using a combination of keywords such as block, cyclic and '*' as explained in Chapter 4, while the function *data* explicitly informs the size of the global structure. Finally, function *grid* indicates the topology of processors in holding the distributed blocks. Consider the example shown in Figure 5-7.

```
blockmap test(P::V)
  define distribute(block, *)
```

```
  define data(80,160)
  define grid(10,1)
end
graph $g
  assert test($a::var@"par.c":100) > 50,000
end
```

Figure 5-7: A Sample Assertion Using Blockmap

In this example, a user first defines a blockmap function named `test` which shows that a 2D array of size `(80,160)` (according to `data` function) is distributed as blocks of rows as described by the `distribute` function, followed by an assertion using the `assert` command. Here the number of rows in each block is dynamically decided by the number of processes that the `$a` process set holds. The number of processes is also made explicit via the `grid` function. For example, given `$a` has 10 processes, each process is expected to retrieve a sub-structure of size `(8,160)`. When the assertion is executed, for each of the running process in the process set `$a`, data from array `var` is collected at line 100 in source file `par.c`. Each element in the array is compared against 50,000 to generate a collection of Boolean results. On the client side, these result collections are aggregated into one big overall result array using the blockmap `test` function.

Chapter 4 presents the algebra for computing mappable blocks from sub-arrays and mapping those to the resulting global array. The algorithm can be performed by the following pseudo-code. Note that we input a `bmap` vector which stores the actual shape of a sub-array given the process ID. This vector can also be computed from the content of the `distribute` function. The `data` vector, on the other hand is just the replication of the `data` function.

```
input:
num_procs       // number of distributed processes
bmap[]          // blockmap vector
data[]          // orginal array shape
for process=1 to num_procs
  // work out number of mappable blocks
  num_blocks=1
  while bmap[i] equals data[i] do
    num_blocks = num_blocks * bmap[i]
  end while
```

```
    // break sub_array in process p into mappable blocks
    blocks=patirtion(sub_array[p],num_blocks)
    // each mappable block is mapped to global array
    for block=1 to num_blocks
      block_start_index=get_start_index(block,p,global_array)
      copy(blocks[block],global_array[block_start_index])
    end for
  end for
```

Figure 5-8: Pseudo-code for Blockmapping Process

Note that the pseudo-code above presents the implementation of the blockmapping algebra examined in Chapter 4. The blockmap description is actually reused for a number of algorithms implemented in this study. For example, the blockmap description is crucial information for the implementation of scalable comparison techniques such as hash-based or direct P2P, because comparison between two global arrays is only feasible when the decomposition can be undone. Therefore, blockmap decomposition description will be revisited later in this chapter.

5.3 Extensible Parallel Statistic Framework

Chapter 4 discussed a *split-phase* mechanism to support the evaluation of statistical operations in parallel. In the following subsections, the corresponding implementation for such design is presented. Importantly, the text focuses on an approach that allows users to extend basic statistical functions to build a more comprehensive set of statistical assertions. This approach first requires a basic *Statistic API*. Second, it reuses the conventional C language and GCC compiler [27] to support the creation and development of user-defined statistical functions. Finally, it supports users to define and evaluate arbitrary statistical models. Later in this chapter, these components are integrated to construct an advanced histogram assertion. In Chapter 6, the actual usage of this technique is illustrated as part of a case study.

5.3.1 User-defined Statistic Function

There are a few options to support users to create statistical reduction functions. First, a debugger command language can be introduced. For example, TotalView defines a special scripting language called *TVScript* [171]. The scripting language allows users to

build functions or actions that can be attached to a certain breakpoint. Another approach is to integrate an existing scripting language (and associated runtime), such as Python [172], into the debugger. This method allows the reuse of a comprehensive scripting engine such as Python; thereby reducing the complexity in developing and supporting a new scripting language. Finally, it is possible to leverage a conventional programming language, such as C, and its compiler and runtime, by dynamically compiling arbitrary modules and linking these into the debugger binary. This approach has the disadvantage that an erroneous library code can crash the debugger itself; however, it significantly simplifies the implementation of a prototype. Accordingly, this technique has been used in this research. To make it easier for a user to write statistical functions, we have defined an API that standardises the interface. Furthermore, a set of pre-defined statistics, using the same technique, are implemented a priori, and these act as templates for users wishing to develop their own functions. Therefore, this debugger comes with functions such as min/max, element counting, standard-deviation, variance, histogram and so forth. They are implemented as an externally defined and compiled library, and these can be extended as required. In the next section, we will discuss this API in further details.

5.3.1.1 Function Template

To enforce the split-phase statistical framework discussed in Chapter 4, a general template is provided in which a few compulsory functions are expected. Consider the pseudo-code in Figure 5-9:

```
func my_func(data, arg1, arg2, ...)
  define my_func_server(data, arg1,arg2,...)=server_result
  define my_func_client(collection,arg1,arg2,...)=client_result
end
```

Figure 5-9: Function Template for Split-phase Operation

Here, `my_func` is defined as a parallel reduction function, and the user must define two sub-functions: `my_func_server` and `my_func_client`. `my_func_server` specifies the computation that can be performed on sub-structures retrieved from the backend debug servers. This function represents the first phase, the parallel phase, of the split-phase mechanism. `my_func_client` specifies how the results of the server function (e.g. `server_results`) are collected from debug servers, merged and transformed (e.g. `client_result`) during assertion evaluation.

5.3.1.2 Compilation and Deployment

After coding, users are not required to compile the code. A debugger command called `register` compiles the source and links its executable against the provided API to create a shared library object. The function name (e.g. `my_func`) is stored in the debugger function table, and the shared library object is shipped to a location which is accessible to both remote debug servers and the frontend client. When the assertion is compiled using the previously discussed Dataflow Compiler, this information is encapsulated along with the variable's name (e.g. via `EXTRACT&REDUCE` template) and sent to debug servers.

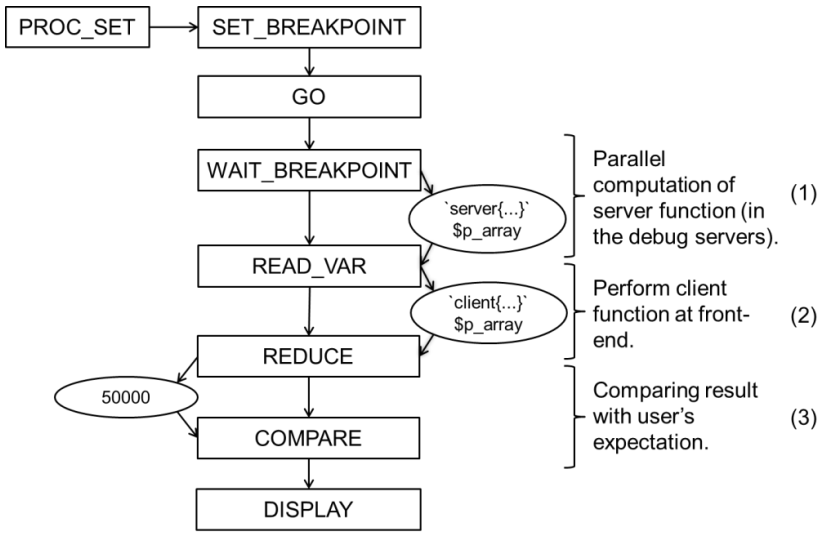


Figure 5-10: Abstract Statistical Assertion with Split-Phase Operation Dataflow Graph

Figure 5-10 above captures an abstract dataflow graph generated for a general statistical assertion using the split-phase reduction scheme. Debug servers perform their assigned tasks in parallel using the `server` function (1). The results are collected and aggregated using the `client` function at the frontend (2). The overall statistical measure is then compared against the user's expectation (3) to flag the final outcome of the assertion.

5.3.1.3 Example

Here, we consider an example where the standard-deviation operation can be specified in the debugger. Because the computation of the standard-deviation value for a distributed dataset cannot be done easily in one pass, we employ the template provided above to build the server function and client function to perform the split-phase mechanism.

```
func stdev_server(a[], n) {
  for i from 1 to n
```

```
      sum += a[i]
    result.mean = sum / n
    for i from 1 to n
      result.sum2 += sqr(a[i]-result.mean)
    result.size = n
    return result
}
func stdev_client(a[], n) {
    cur_mean=servers[1].mean
    cur_size=servers[1].size
    cur_sum2=servers[1].sum2
    for i from 2 to num_procs {
      sigma=servers[i].mean–cur_mean
      cur_size=cur_size+servers[i].size
      product_size=cur_size*servers[i].size
      cur_mean=cur_mean+sigma*servers[i].size/cur_size
      cur_sum2+=servers[i].sum2+sigma*sigma*(product_size/cur_size)
      cur_size=servers[i].size + cur_size;
    }
    return sqrt (cur_sum2/cur_size)
}
```

Figure 5-11: Pseudo-code for User-defined Stdev Function using the Split-phase Template

The pseudo-code above defines two functions: `stdev_server` and `stdev_client`. `stdev_server` calculates a set of values including `sum2`, `mean` and `size`, while `stdev_client` aggregates these values to compute the overall standard-deviation value of the decomposed dataset, using the formulas presented in Chapter 4. To use these functions in a standard-deviation assertion, a user issues the following commands:

```
register <path_to_C_file>/stdev
graph $g
    assert stdev($a::var@“par.c”:100) > 0.2
end
```

Figure 5-12: A Simple Statistical Assertion using the User-defined Function

The file `stdev.c` is compiled, and the `stdev` function is registered. The execution of the assertion computes the standard-deviation value using each element in the array `var` at line 100 of source file `par.c`, and then compares it against the constant 0.2. If elements

in array `var` deviate more than 0.2, the assertion fails and the program will be stopped for further examination.

5.3.2 User-defined Data Models

To help users create abstract data models, this work introduces a new debugger built-in function, called `randset` that defines random variates. The type of probability density function, for example Gaussian, Cauchy, Poisson etc, and the number of samples, describe a random dataset. In addition, different distribution functions require different sets of parameters including mean, standard-deviation, and scale parameter values.

```
randset(<distribution_name>,<dataset_size>,...)
```

The debugger in this work currently supports various typical distribution models including Binomial, Gaussian, Cauchy, Poisson, and Maxwell-Boltzmann distribution. Together with the use of the `create` and/or the `assign` commands, the `randset` function returns a debugger variable which can be populated in parallel by the debug backend through the execution of the split-phase mechanism.

5.3.3 The Histogram Reduction Command

While the histogram reduction operation can be defined using the user-defined function template presented above, it is also provided as a predefined debugger command. The definition of histograms can be simplified by assuming that the range values for each bin are uniformly distributed between a pair of values (`range_start`, `range_end`), given a number of bins `n`. Hence, a user can invoke a histogram reduction command in Guard using the following syntax:

```
set reduce histogram n, range_start, range_end
```

5.3.4 The Built-in Parallel Statistic API

A new component, *Statistic API*, is added into the current architecture (shown in Figure 4-3). This API contains a collection of functions, which validate, compile and communicate user-defined statistical functions between the frontend client and the backend debug servers. The `register` command discussed earlier makes use of these API functions. Likewise, the debug servers invoke functions in this API to perform various statistical reduction activities. Furthermore, when the assertion requires the

creation of user-defined data models, statistic API provides routines to evaluate the `randset` command and produce the random variates.

5.3.5 Evaluation of Histogram Assertion

Here, we demonstrate how the implementation presented earlier can be used to build and execute a histogram assertion. Through the example, we highlight how statistical tests can be employed to verify a runtime statistical model against the user's expectation.

To define a complete histogram assertion, one needs to:

- define the expected data distribution model using the `randset` function described above; and
- provide the information such as the start/end range and the number of bins for constructing histograms through the histogram reduction command.

An assertion can then be built using the syntax denoted earlier. Since we do not compare two sets of exact values, but rather compare two abstract data models, a new *estimate operator* "~" is introduced. The output of the "~" operator is the result of a χ^2 *goodness of fit test* [173] between the two given histograms. The χ^2 goodness-of-fit test is used to establish whether or not an observed frequency distribution differs from a theoretical distribution. Recently, Gagunashvili discussed the generalisation of the technique to compare different classes of histograms such as unweighted versus weighted histograms, weighted versus weighted histograms, unweighted histogram versus normalised histogram, and so forth [174, 175]. We adopt a similar technique, but focus on the simpler types of histograms: non-normalised histograms that have the same number of bins, and distributed through the same values domain. The statistic χ^2 can be calculated using the general formula given below [173].

$$\chi^2 = \sum_{i=1}^{N} \frac{(H_i - E_i)^2}{E_i} \qquad (10)$$

where: H_i: the observed frequency for bin i

E_i: the theoretical frequency for bin i

The value of χ^2 will then be used to determine the *p-value* by comparing it to a chi-squared distribution. The p-value is used to assert the hypothesis by comparing it with

user-defined *significance level* α. This parameter is also specified in the assertion. Consider the following Guard specification in Figure 5-13:

```
create $model
assign $model = randset(gaussian,100000,0.05)
set reduce histogram 1000,0.0,1.0
assert $a::p_array@"code.c":10 ~ $model < 0.02
```

Figure 5-13: A Sample Histogram Assertion

The example above describes the comparison of the histogram constructed using data obtained from the runtime variable `p_array` at line 10 of source file `code.c`, and the histogram generated using the dataset pointed by the debugger variable `$model`. According to the `assign` command, `$model` is a random variate consisting of 100,000 samples from the Gaussian distribution with a standard-deviation of `0.05`. If the χ^2 test result is smaller than the significance level `α=0.02`, the hypothesis is accepted and the runtime data at that breakpoint follows the Gaussian distribution model. An abstract dataflow graph captured in Figure 5-14 summarises the evaluation of the histogram assertion above.

5.4 Result Visualisation

The visualisation of differences obtained by the execution of user-defined assertions underpins the data-centric debugging paradigm. Currently, a command line user interface is the primary means of interaction with the debugger. Therefore, to support more useful visualisation, the previous version of Guard allows assertions to generate intermediate data files that can then be examined by a visualisation package such as IBM's Open Visualization Data Explorer [106] or VIS5D [105]. In addition, a number of tools are provided to assist with this visualisation process. These mechanisms are designed to display data differences between two datasets. In this study, a simpler visualisation solution is developed to better visualise the outcomes of assertions such as the statistical assertions. For simple statistical assertions, the outcome of the assertion can be presented as text via the console. However, in order to support the visualisation of statistical results such as a histogram, this work enhances Guard to plot charts and graphs. According to earlier conventions, when defining an assertion in Guard, the user can select a particular display type using the `set display` command, and set the output file name using the `set output` command. These commands are upgraded to support the generation of charts and

graphs using the well-known *plotutils* library packages [176]. Importantly, to present two histograms on the same graph, we normalise one of the histogram into a curve. This technique assumes that the normalised histogram has been constructed from a probability distribution function (e.g. user-defined random variates). Examples of visualisation output are shown in Chapter 6.

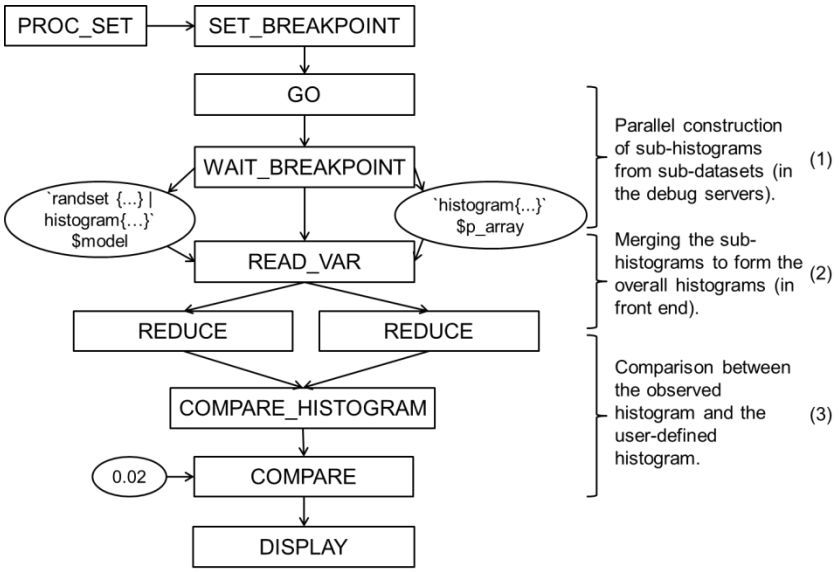


Figure 5-14: Abstract Histogram Assertion Dataflow Graph

5.5 Scalable Comparison Techniques

In Chapter 4, through examining the limitations in the Copy-Combine-Compare scheme used in the earlier version of Guard, we discussed the design for several scalable solutions including the hash-based comparison and the direct P2P comparison techniques. For both solutions, the application of blockmap decomposition information is recognised as crucial. This information is primarily used to determine how debug servers can actively participate in the comparison process given each of them only owns a portion of the global array. This is where the execution of comparative assertions becomes more scalable. In this section, we present the implementation of those scalable schemes.

5.5.1 Hashed-based Comparison Technique

5.5.1.1 Enabling Hashing Reduction

To enable the use of the hashing to evaluate comparative assertions, a set of hash functions consists of the Bob Jenkins' hash function, the FNV hash functions, and the

AP hash functions [169] are incorporated into the debugger. These function can be triggered using the `set reduce` command as shown earlier in Section 5.1.3. Note, in Chapter 4, we also addressed the problem of hashing data such as floating-point numbers. The conventional tolerance method is ineffective when it comes to hashing data before comparison. Thus, a rounding operation was suggested. To implement this, a hash function can be requested with two additional parameters specifying the lower bound and upper bound for floating-point number truncation. Therefore, an example of a comparative assertion using FNV hash function can be defined as in Figure 5-15:

```
invoke $master,$b[4] ./p5_4a
invoke $m2,$c[8] ./p5_4b
blockmap bm1(P::V)
  define distribute(block,*)
  define data(8,16)
  define grid(4,1)
end
blockmap bm2(P::V)
  define distribute(*,block)
  define data(8,16)
  define grid(1,8)
end
set reduce fnvhash 5 6
graph $g
  assert bm1($b::big@“pa5a.c”:42)=bm2($c::large@“pa5b.c”:55)
end
```

Figure 5-15: Script for Executing a Comparative Assertion with the Hash-based Scheme

The above script invokes two MPI programs, one with four slaves and the other with eight slaves. Two blockmap definitions are provided. Function `bm1` performs the row major data distribution while function `bm2` performs the column major data distribution. The graph `$g` is created to assert the content of the variable `big` at line 42 in `pa5a.c` against the content of the variable `large` at line 55 in `pa5b.c`. The `set reduce` command enables the `fnvhash` function in the scope of this graph. A hash function also allows the user to define an absolute comparison tolerance much like the error tolerance described in Section 4.5.1. It truncates floating-point values to a specified digit. A lower

precision value is defined to specify when a difference should indicate an error and an upper precision value is used to specify when a difference should indicate a warning.

5.5.1.1 The Overlap Algorithm

The blockmap functions `bm1` and `bm2` described above perform a row major decomposition and a column major decomposition, respectively. In addition, `$b` and `$c` are process sets with different numbers of processes (4 and 8). Therefore, simply computing a single signature for the data held by each processor will return different numbers of signatures from the two programs. The previous chapter outlines an algebra which computes the greatest overlapping area between the two disparate blockmap descriptions. The implementation for this algebra is nontrivial and can be portrayed as follows.

In Chapter 4, we call the size of the greatest overlapping area `hash_size`. Debug servers from both programs need to receive this information which they can use to decide how much data can be hashed into one signature so that the total numbers of signatures attained from both programs are the same. The frontend client, through the user interface has access to both decomposition descriptions such as `bm1` and `bm2`, thus it is responsible for computing the `hash_size` value. Because no program has been invoked and executed yet, the size of the data block that a processor potentially holds can only be calculated using the blockmap information and the number of processes. This value is computed with the pseudo-code below.

```
input:
  rank               // number of dimensions for the global array
  bmap_distribute[]  // blockmap distribute vector
  bmap_data[]        // blockmap data vector
  bmap_grid[]        // blockmap data vector
output:
  blocksize[]        // size of data a processor holds
for i=1 to rank:
  if bmap_distribute[i] == BLOCK
    blocksize[i] = bmap_data[i] / bmap_grid[i];
  else if bmap_distribute[i] == STAR
    blocksize[i] = bmap_data[i]
  else if bmap_distribute[i] == CYCLIC
```

```
        blocksize[i] = 1;
    end for
```

Figure 5-16: Pseudo-code to Compute the Size of Data Held by a Debug Server

The following pseudo-code describes how `hash_size` value is computed given the block size from both programs.

```
input:
  rank         // number of dimensions for the global array
  blocksize1[] // blocksize for program 1
  blocksize2[] // blocksize for program 2
output:
  hash_size[]  // vector describes hashable block
for i=1 to rank:
  hash_size[i] = gcd(block_size1[i], block_size2[i])
end for
```

Figure 5-17: Pseudo-code to Compute Block Size for Hashing

This process is carried out when the assertion description is compiled by the Dataflow Compiler.

5.5.1.2 Hashing Runtime Data by Debug Servers

Hash-based comparison benefits from the use of a hash function to reduce the volume of data that needs to be transferred back to the frontend client for comparison. Theoretically, the technique achieves good performance because both the communication, combination, as well as comparison phases can be carried out in reasonable amounts of time given the large datasets involved. However, the scalable part of this technique actually lies in the ability to parallelise the hashing process, since the hashing time may be quite long. Hence, the key implementation of the hash-based technique is the support for backend debug servers to perform hashing on their sub-arrays independent of each other given some arbitrary decomposition descriptions.

A hashing request is composed of the `hash_size` value as computed above, the hash function ID and the lower/upper bound for floating-point truncation. Through the graph engine, a hashing request is attached with the regular variable request, using the `EXTRACT&REDUCE` template as described in Section 5.2.1. The whole request is sent to the debug servers. The debug server is also further developed to parse the request and look for any hashing information. Upon attaining their portions of the runtime variable,

the debug servers slice the structure into *hashable blocks* using the `hash_size` value. Following that, these blocks are hashed sequentially to produce a hash signatures array. This array is transferred back to the frontend client for reconstruction of the global structure and then comparison follows. Below are the sample hashing requests for the above comparative assertion.

```
`hash(2,2#35,0,0)`big
`hash(2,2#35,0,0)`large
```

The above requests indicate hashing reduction is required on variables `big` and `large`. The list of number on the left side of the '#' character specifies the `hash_size` while the numbers on its right side suggests which hash function to use along with the upper and lower bound for floating-point truncation.

5.5.1.3 Collecting Hash Signatures for Comparison

Regarding the actual communication between the frontend client and backend debug servers, we use collective operations similar to the broadcast and gather operations in MPI in order to collect the data from the processes as quickly as possible. In particular, broadcast is used to request the debug processors to hash their data in parallel – all responding to a single broadcast command. Collective gather operations are used to merge signatures for comparison in the head node.

5.5.2 Direct P2P Comparison Technique

The P2P comparison technique allows remote debug servers to communicate their debug data between themselves so that the comparison process can be parallelised, while the global data restructure task is omitted completely. In addition, raw data is compared between programs, unlike the comparison of reduced hash signatures. [117]For simplicity, we call the program that sends data the *sender* and the program that receives and performs comparison the *receiver*.

5.5.2.1 Enabling P2P Comparison Mode

The following command can be used to trigger the direct P2P comparison mode when a comparative assertion is evaluated.

```
set p2p true
```

This command enables the use of the `EXTRACT&COMPARE` template as described in Section 5.2.1. Blockmap descriptions from both programs are converted to plain strings and are attached with the regular variable request. The debug server is also further developed to parse the request, capture the blockmap information and perform the P2P comparison tasks as described below.

5.5.2.2 P2P Comparison by Debug Servers

Upon extracting the runtime data from the monitored process, a debug server needs to perform a few tasks to carry out the P2P comparison scheme. First, the sending debug server needs to calculate the amount of data that could be sent to a receiver debug server. This amount of data is the greatest overlapping area between the two disparate blockmap descriptions, and can be computed using the implementation done in Section 5.4.1. Here we call this information a *comparable block*. After successfully computing this value, the debug server slices the data it holds and works out which debug server should receive the data. This is done using the mapping implementation as described later. Upon computing the comparable block information and the mapping information, a sender debug server performs the data slicing routine and sends the slice to the mapped receiver debug server. If the debug server is a receiver, it works out how many slices it needs to receive and computes the slicing information accordingly (i.e. the comparable_block value). It then opens a socket and waits for the slices to arrive. When a slice is successfully received, the comparison can then be done by the receiver. The overall process is summarised by the following pseudo-code.

```
input:
  num_blocks // number of blocks needed to be transferred
if sender:
  for i=1 to num_blocks:
    sliceinfo = FindSliceInfo(data, i)
    receiver = FindSliceDestination(data,overlap_info,i)
    tosend = PerformSlicing(data, slice_info)
    SliceSocketSend(receiver, tosend)
  end for
else:
  for i=1 to num_blocks:
    torecv = SliceSocketReceive()
    sliceinfo = FindSliceInfo(data, i)
```

```
    tocmp = PerformSlicing(data, slice_info)
    diffs[i] = Compare(tocmp, torecv)
  end for
```

Figure 5-18: Pseudo-code to Implement P2P Process

The result of the above routine is the array `diffs`. This array stores Boolean values which signal any difference found. The array is then transferred back to the frontend client for displaying and flagging the outcome of the overall assertion.

5.5.2.3 Building the P2P Map

The primary requirement for enabling P2P comparison is the ability to map a debug server `i` in the sender program to a debug server `j` in the receiver program. To find the mapping, a debug server needs to virtually *undo* and *redo* the decomposition process using the blockmap information provided by the frontend client.

The *undo* procedure starts with each debug server of the sender program using the local indexing information of the block and the rank ID of the process to map back that block onto the global structure. This step results in the global indexing information of that particular block. The pseudo-code below describes how the undo task is carried out.

```
input:
  rank, pid[]  // process index vector
  blocksize[]  // local array size
  datasize[]   // global array size
output:
  global_start_loc[] // location of 1st element in global array
for i=1 to rank:
  sub_proc_grid[i] = datasize[i] / blocksize[i];
end for
for i=1 to rank:
  block_loc[i] = pid[i] * sub_proc_grid[i];
end for
for i=1 to rank:
  global_start_loc[i] = block_loc[i] * blocksize[i];
end for
```

Figure 5-19: Pseudo-code to Undo Decomposition

The result of the undo task is the location of the first element of the local block on the global array. This global indexing information can then be used along with the receiver's blockmap information to identify which process from the receiver program that holds the required block. This is how we *redo* the decomposition (shown in Figure 5-20).

```
input:
  rank, datasize[]   // global array size
  global_start_loc[] // location of 1st element in global array
  bmap_grid[]        // blockmap data vector
output:
  proccess_id        // target process to send data to
for i=rank to 1:
  procces_id+=global_start_loc[i]/datasize[i]*bmap_grid[i+1];
end for
```

Figure 5-20: Pseudo-code to Redo Decomposition

The outcome of the redo task is a process rank ID of the receiver program. At this point, a sender's debug server has sufficient information including block size, and receiver process rank ID to perform a P2P data transfer.

5.5.2.4 Point-to-Point API

P2P comparison requires point-to-point (P2P) direct communication between remote debug servers from two separate programs. There are several ways to achieve this. As a proof of concept, we introduce a P2P API which makes use of the traditional TPC/IP protocol. This API provides functions for a debug server to (1) inform its network information such as host name and port number to the rest of the debug servers, (2) open a socket at the beginning of a debug section if the P2P comparison mode is enabled, (3) close it when P2P mode is disabled or when the debugger exits, and finally send and receive socket streams in non-blocking manner.

5.5.2.5 Collecting Comparison Results

In order to collect the comparison results from the processes as quickly as possible, we use collective operations similar to the broadcast and gather operations in MPI. In particular, broadcast is used to request the debug servers to retrieve debug data in parallel and communicate with their peers to perform direct P2P comparison – all responding to a single broadcast command. Collective gather operations are used to merge the P2P comparison results (i.e. in the form of Boolean values) for displaying in the head node.

This approach is quite scalable because, even though we merge the results sequentially in the head node, these are quite small, and the comparison operations (which can be time consuming) are performed in parallel.

5.5 Summary

This chapter has presented the details comprising the implementation of an assertion-based debugger. In particular, the chapter has examined various implementation aspects for enhancing an existing parallel debugger to handle different types of ad-hoc debug-time assertion. In addition, the chapter has described the development of innovative components that are specific to the debugger proposed in this study. These components include an extensible statistic framework, and two scalable comparison techniques.

Chapter 6

Case Studies

This chapter delivers three case studies to demonstrate the application of assertion-based debugging in a parallel computing environment. The case studies have been selected because they represent a class of applications that is hard to debug, namely, *highly parallel applications*. The case studies involve:

- the invocation of a large number of processes;
- the generation and transformation of large amounts of data; and
- the use of major parallel architectures to perform computationally expensive operations.

As a result, the case studies included in this chapter target a prototypical computational fluid-dynamics simulation (using the Shallow Water Equations program [177]); a molecular-dynamics simulation (based on the evaluation of the Lennard-Jones potential [178]); and a 2D photon-transport simulation (using the Monte Carlo method to solve the Boltzmann transport equation [179]). The aim of the case studies is to demonstrate that the architecture described in Chapter 4 can manage the evaluation of ad-hoc assertions by utilising the underlying parallel platforms. The case studies also highlight the efficiency of the debugging technique for automatically locating and notifying users of hard-to-spot errors. Furthermore, these case studies illustrate the expressive power of the proposed assertion templates such as general ad-hoc assertions, statistical assertions and comparative assertions, respectively.

6.1 Case Study 1: The Shallow Water Equations Program

This case study illustrates the use of general ad-hoc debug-time assertions, as introduced in Chapter 3, for finding errors in a simple but scalable scientific model, the Shallow Water Equations program [177]. This program is a simplified version of the code found in real weather models such as the Weather Research and Forecast (WRF) model [159]. The program's simplicity allows it to be applied to a small case study. As a result, this program can be used to demonstrate the effectiveness of the debugging technique without losing the reader in the detail of real code.

6.1.1 The Shallow Water Equations Program

The Shallow Water Equations program describes the motion of an incompressible fluid with a free surface, but with the constraint that the horizontal scales of motion are much larger than the vertical [177]. The equations are a favoured choice for experiments with various model structures and numerical schemes. Although a very simple representation of the atmosphere, they do include the two types of horizontal wave motion important in more realistic models: gravity waves and Rossby waves. The equations are also useful for experiments in parallelising because they include much of the structure (and so also the problems associated with data communication and synchronisation) of more complicated models.

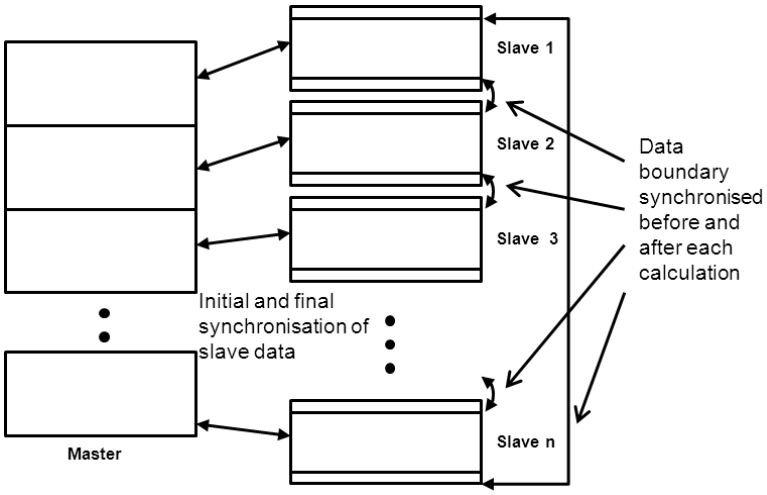


Figure 6-1: Design of the Shallow Water Equations Program

In 1991, Abramson et al. implemented an MPI program for the finite difference form of the equations [180]. In this MPI code, a master process initially distributes the key variables to each of the slaves, and the slaves then move through a sequence of steps. At

each step (a set period of time), the slaves interchange part of their state variables with their neighbours. At the end of each step, the slaves independently move forward and the whole process continues. This arrangement is shown in Figure 6-1 above.

Table 6-1: Incorrect Outputs from the Shallow Water Equations Program

Number of points in the X direction 64 Number of points in the Y direction 64 Grid spacing in the X direction 100000.00 Grid spacing in the Y direction 100000.00 Time step 90.000 Time filter parameter 0.001 Cycle number 1 Model time in days 0.00 Potential Energy 0.00 Kinetic Energy 12038.205 Total Energy 12038.205 Pot.Enstrophy 0.0000e+00	Expected Output
Number of points in the X direction 64 Number of points in the Y direction 64 Grid spacing in the X direction 100000.00 Grid spacing in the Y direction 100000.00 Time step 90.000 Time filter parameter 0.001 Cycle number 1 Model time in days 0.00 Potential Energy NaN[5] Kinetic Energy NaN Total Energy NaN Pot.Enstrophy NaN	Actual (Incorrect) Output

6.1.2 Debugging the Shallow Water Equations Program

In this section, several debug-time assertions are constructed to diagnose the erroneous behaviour of the program shown in Table 6-1. Earlier in Chapter 3, some use case scenarios for this type of assertion were discussed. Below, each of those scenarios is examined and tested with specific assertions in order to narrow down the scope of the errors and to identify the programming defects. The program is invoked by the debugger using the following command. Accordingly, one master process is denoted as `$m` and 128 slave processes are presented by process set `$a`.

[5] NaN, standing for not a number, represents an undefined or inexpressible value, often resulted from incorrect floating-point calculation.

```
invoke $m, $a[128] ./shallow
```

6.1.2.1 Assertion 1 – Checking Master State

The Shallow Water Equations program begins with the initialisation of key data structures before any time-step computation commences. The initialisation process occurs in two phases: initialising key data structures in the master process and distributing these to the compute slaves. The following assertion checks that the pressure array in the master is correctly initialised before any calculations are undertaken.

```
graph $case1_g1
  assert $m::p@"main.c":180 = 50000
end
```

Figure 6-2: Case Study 1 - Assertion for Testing Initialisation By The Master Process

To execute this assertion, the debugger sets a breakpoint at line 180 in `main.c`. When the breakpoint is reached, the debugger extracts the content of `p` (the pressure array) from `$m` (the master process) and checks that each array element equals the constant 50,000. In this example, a violation of the assertion reveals an error in the initialisation code in `init.c`, as shown below. As a result, the assertion indicates that multiple elements in `p` do not equal 50,000. Inspection consequently reveals an indexing defect; namely, only the diagonal is initialised. Rerunning the assertion after fixing the index value shows pressure array values in the master process were successfully initialised.

Incorrect code

```
for (j = 0; j < n; j++) {
      for (i = 0; i < m; i++) p[j][j] = 50000.;
}
```

Correct code

```
for (j = 0; j < n; j++) {
      for (i = 0; i < m; i++) p[j][i] = 50000.;
}
```

6.1.2.2 Assertion 2 – Checking Communication between Master and Slaves

During the second phase of the initialisation process, the pressure array `p` (and other key data structures such as velocity or energy) are distributed to slave processes using a `(block,*)` decomposition. In this phase, an error in the communication routine could result in faulty initialisation and data distribution, and could lead to later incorrect

computation by the slave processes. Verifying the successful completion of the initial decomposition tasks is therefore necessary. The following assertion checks that each slave receives the right portion before any computation proceeds.

```
blockmap case1_bm(P::V)
   define distribute(block, *)
   define data(1024, 1024)
   define grid(128, 1)
end
graph $case1_g2
   assert  case1_bm($a::p@"main.c":21)=$m::p@"main.c":43
end
```

Figure 6-3: Case Study 1 - Assertion for Testing Data Decomposition by the Master Process

As discussed, the blockmap function defines the decomposition scheme used. Using blockmap information, the debugger can rebuild the global data structure from the distributed pieces. In this example, the function `test` describes the decomposition of a two dimensional array of size 1024x1024 into blocks of rows. The parameters `P` and `V` are place holders for the set of processes (`P`) and the program variable (`V`).

The assertion checks that the decomposed array, `p`, in the slaves (`$a`) is the same as the `p` array in the master process (`$m`). It uses the function `test` to know how to recombine the pieces into a single array.

Execution of the assertion indicated that some of the slaves were incorrectly initialised. Closer investigation of the `MPI_Send` code from the master process revealed an error. In particular, when data in the `p` array is sent to the slaves, the upper and lower bounds of the inner `j` loop are incorrect. Correcting the following code in `main.c` fixed this bug.

Incorrect code

```
for (i = 1; i < proc_cnt; i++) {
   for (j = 0; j < cs; j++) {
      acopy_two_to_one(p, p_start, j);
      MPI_Send(&p_start,m,MPI_FLOAT,i,P_ROW,MPI_COMM_WORLD);}}
```

Correct code

```
for (i = 1; i < proc_cnt; i++) {
   for (j = (i-1)*cs; j < cs+((i-1)*cs); j++) {
      acopy_two_to_one(p, p_start, j);
```

```
MPI_Send(&p_start,m,MPI_FLOAT,i,P_ROW,MPI_COMM_WORLD);}}
```

6.1.2.3 Assertion 3 – Checking Slave-to-Slave Communication

Fixing the initialisation process provides the correct result for the first cycle of the simulation as shown below. However, the program still proceeds incorrectly. This problem indicates that the computational results for each time step need to be verified.

Table 6-2: Error in Slave-to-Slave Communication

Cycle number 1 Model time in days 0.00 Potential Energy 0.00 Kinetic Energy 12038.205 Total Energy 12038.205 Pot.Enstrophy 0.0000e+00 Cycle number 50 Model time in days 0.05 Potential Energy NaN Kinetic Energy NaN Total Energy NaN Pot.Enstrophy NaN	Actual (Incorrect) Output

Before and after each major computational step, each slave synchronises the edges of the data structures (that is, the ghost bands) with its immediate neighbours using the following code:

```
/* loop over latitudes calculating U,V,z,h from jstart to jend */
load_ghost(prv,nxt,my_id,jstart,jend,p,u,v,p_sync,u_sync,v_sync);
calcuvzh(jstart,jend,p,u,v,cu,cv,h,z,p_sync,u_sync,v_sync,cu_sync,
        cv_sync,h_sync,z_sync,fsdx,fsdy);
unload_ghost(prv,nxt,my_id,jstart,jend,cv,z,cv_sync,z_sync);
```

Figure 6-4: Case Study 1 - Ghost-Band Synchronisation Code

Each slave needs to communicate with two neighbours to perform the synchronisation process. In order to successfully synchronise data between slaves, given that each slave only carries a portion of the overall data structure, a ghost band is required. Accordingly, the `load_ghost` function receives a ghost band from one neighbour and sends information to the ghost band of the other neighbour. Likewise, function `unload_ghost` sends information in the other direction.

The following assertions test the ghost-band synchronization for the pressure variable `p` and its corresponding ghost band, `p_sync`, between three of the slaves. Note that a slicing operation is required to capture only the first row of the 2D array `p`.

```
graph $case1_g3
   assert $a[0]::p_sync@“main.c”:448=
          $a[1]::p[0][0..1024]@“main.c”:448
   assert $a[1]::p_sync@“main.c”:448=
          $a[2]::p[0][0..1024]@“main.c”:448
   assert $a[2]::p_sync@“main.c”:448=
          $a[3]::p[0][0..1024]@“main.c”:448
end
```

Figure 6-5: Case Study 1 - Assertions for Testing Ghost-Band Synchronisation By Slave Processes

The outcome of the assertion shows that the content of the `p_sync` array from process `$a[0]` is not the same as that of the first row of `p` array from process `$a[1]`. Function `load_ghost` contains the following code:

```
load_ghost(prv,nxt,my_id,jstart,jend,p,u,v,p_sync,u_sync,v_sync)
{   acopy_two_to_one(p, tmp_ds, 0);
    neighbour_send(prv, my_id, CALC1a, tmp_ds);
    acopy_two_to_one(u, tmp_ds, 0);
    neighbour_send(prv, my_id, CALC1b, tmp_ds);
    acopy_two_to_one(v, tmp_ds, 0);
    neighbour_send(prv, my_id, CALC1c, tmp_ds);
    neighbour_receive(nxt, my_id, CALC1b, p_sync);
    neighbour_receive(nxt, my_id, CALC1a, u_sync);
    neighbour_receive(nxt, my_id, CALC1c, v_sync);
}
```

Figure 6-6: Case Study 1 - load_ghost Function Body

For each `neighbour_send` and `neighbour_receive` function call, an ID value is passed to indicate the key data structure to be synchronised. For example, `CALCa` means pressure array `p`. The violation of the assertions shows that the `p` array was not synchronised successfully. Examining the body of the `load_ghost` routine shows that incorrect ID values were passed between the `u` and `p` arrays on the receive ends. In this example, each of the assertions is executed in parallel because the dataflow graph contains independent threads of execution.

Incorrect code
`neighbour_receive(nxt, my_id, CALC1b, p_sync);` `neighbour_receive(nxt, my_id, CALC1a, u_sync);`

Correct code

```
neighbour_receive(nxt, my_id, CALC1a, p_sync);
neighbour_receive(nxt, my_id, CALC1b, u_sync);
```

6.1.2.4 Assertion 4 – Checking Slave State across Time Steps

Fixing the bugs addressed by assertions 1, 2 and 3 allows the simulation to proceed to its conclusion with sound outputs. However, a comparison with the expected outputs reveals differences.

Table 6-3: Error in Slave State across Time Steps

Cycle number 50 Model time in days 0.05 Potential Energy 1047.34 Kinetic Energy 11499.778 Total Energy 12547.11 Pot.Enstrophy 5.357120e-29 Cycle number 100 Model time in days 0.10 Potential Energy 125.75 Kinetic Energy 11973.558 Total Energy 12099.31 Pot.Enstrophy 5.507058e-29	Expected Output
Cycle number 50 Model time in days 0.05 Potential Energy 32759.43 Kinetic Energy 11978.55 Total Energy 44737.98 Pot.Enstrophy 1.505172e-17 Cycle number 100 Model time in days 0.10 Potential Energy 39746.45 Kinetic Energy 14225.30 Total Energy 53971.75 Pot.Enstrophy 1.463373e-16	Actual (Incorrect) Output

The formulation of the Shallow Water Equations used in this case study is based on a finite different approximation. Because of that, it is possible to calculate an upper bound for the changes in any of the array values within a time step, which are limited by the time-step increment, the finite-difference step and the range of values[6]. This information can be used to set the upper bound on the difference between the old values of each of the state arrays and the new values. This bound can be implemented as part of an assertion that checks the solution when new values are calculated.

[6] We actually observed the upper bound used here in a version of the code that was working correctly, rather than calculating it from first principles. However, the latter is possible if a correct version of the code is not available.

During execution, the program maintains both current pressure values in the `pnew` array and previous pressure values in the `pold` array. The following assertion checks that the difference is less than the prescribed upper bound.

```
graph $case1_g4
   assert $a::pnew@"main.c":463 - $a::pold@"main.c":463 < A
end
```

Figure 6-7: Case Study 1 - Assertion for Validating Pressure Values at Each Time Step

Because of an error in the time-stepping code, we detected a difference that was out of bounds. The result was, again, an indexing error, as shown below.

Incorrect code
`pnew[i][j]=pold[i][j]+tdt*dpdt[j][j];`
Correct code
`pnew[i][j]=pold[i][j]+tdt*dpdt[i][j];`

6.1.2.5 Assertion 5 – Reducing Slave State

Finally, a deeper understanding of the physics of an application can be used to specify powerful assertions on the state of the system. For example, the Shallow Water Equations program conserves both mass and energy. This implies that, as the calculation proceeds, the total amount of mass in the system should only change a small amount (subject to numeric resolution), and that any divergence from this amount indicates that the program is not behaving correctly. Using this type of information, one could write an assertion to test whether mass and energy are indeed conserved. While the formulation of the Shallow Water Equations program does not directly compute the mass, another invariant unit in the program, the *total pressure* value, does. Thus an assertion that checks the value of the total pressure in the system is useful for detecting errors. The following commands (Figure 6-8) perform a sum reduction across the pressure data values from all slaves and checks whether this reduction is within a predefined error bound.

Upon execution, the debugger computes a set of partial sums in parallel and then reduces these using a further sum function at the client side. The `set error` statement sets an error tolerance to 10^{-6}. In this example, the debugger executed the code for a number of cycles and confirmed that the total pressure was within bounds. As a result, it is concluded that the code was working correctly.

```
set reduce sum
```

```
set error 1e-6
graph $case1_g5
  assert $a::p@“main.c”:463 = 50000
end
```

Figure 6-8: Case Study 1 - Assertion for Testing Total Pressure Value

6.1.3 Case Study 1 - Summary

This case study presented a debugging section using a number of simple debug-time assertions to detect, locate and fix several bugs in the Shallow Water Equations program. Even though the debugged program was relatively small and simple compared to full-scale scientific applications, it nonetheless illustrated the kind of assertions proposed in this research. In any programming paradigm, certain phases comprise the overall execution of the program. Using debug-time assertions to verify the successful computation of each phase is a recommended practice. This case study exemplifies the use of ad-hoc debug-time assertions to find the errors in parallel applications.

6.2 Case Study 2: Molecular Dynamics

In this second case study, a parallel molecular-dynamics code [181] written in C/MPI is used to demonstrate the use of statistical assertions in debugging a scientific application. As explained earlier, statistical assertions explore the statistical attributes of complex computations in scientific code to verify the state of its execution at runtime. For this case study, a set of program bugs presented in Frenkel et al. [182] is replicated to illustrate the expressive power and potential of statistical assertions for finding errors.

6.2.1 Physics Background

This code uses the Lennard-Jones (LJ) potential in modelling a fluid. LJ potential is popular for investigating various liquid phenomena (such as melting points, the liquid-vapour surface and nucleation [178]), and is a fundamental simulation in molecular dynamics. The simulation consists of a 3D cube with each dimension of size L, which contains randomly positioned particles with random initial velocities. At each time step, the system computes the new positions for all particles using the interaction force between the particles and their current velocities. The interaction force is modelled by the *LJ potential* below:

$$V_{LJ} = 4\varepsilon\left\{\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right\} = \varepsilon\left\{\left(\frac{r_m}{r}\right)^{12} - 2\left(\frac{r_m}{r}\right)^{6}\right\} \quad (10)$$

where:

ε is the depth of the potential well;

σ is the finite distance at which the inter-particle potential is zero;

r is the distance between the particles; and

r_m is the distance at which the potential reaches its minimum.

Since boundary conditions are enforced, particles that leave the box on one side appear on the opposite side. The fluid's temperature inside the box is computed from the particle velocities.

6.2.2 Debugging the Simulation

6.2.2.1 Monitoring Particle Speed

Because all particles have the same mass, their kinetic energy depends only on their speed. Throughout the course of the simulation, particles interact with their neighbours and the speeds are updated accordingly. In any given fluid, the speed varies substantially, from very slow particles to very fast ones. However, this scalar value spreads according to Maxwell-Boltzmann distribution [182]. Monitoring particle speed can thus help to detect anomalies in a simulation.

In order to test this assumption, a user can assert that the histogram constructed using the `speed_array` variable is similar to a histogram generated with samples picked from the Maxwell-Boltzmann distribution. This is done with the *histogram assertion* below:

```
assign $model = randset(maxwell,49152,6,2)
set reduce histogram 100 0.0 10.0
graph $case2_g1
   assert $a::speed_array@pmd.c:28 ~ $model < 0.02
end
```

Figure 6-9: Case Study 2: Assertion for Testing Distribution of Particle-Speed Values

The first command builds a set of 49,152 samples from the Maxwell-Boltzmann distribution. This command requires two constants: the current temperature value `T` and the particle's mass `m`. The distribution function is described below:

$$f(v) = \sqrt{\frac{2}{\pi}\left(\frac{m}{kT}\right)^3} v^2 \exp\left(\frac{-mv^2}{2kT}\right) \qquad (11)$$

where k is the Boltzmann constant.

The following commands request the data obtained from both the speed_array variable and the random number set to be reduced to histograms with 100 bins and to accumulate data ranging from 0.0 to 10.0.

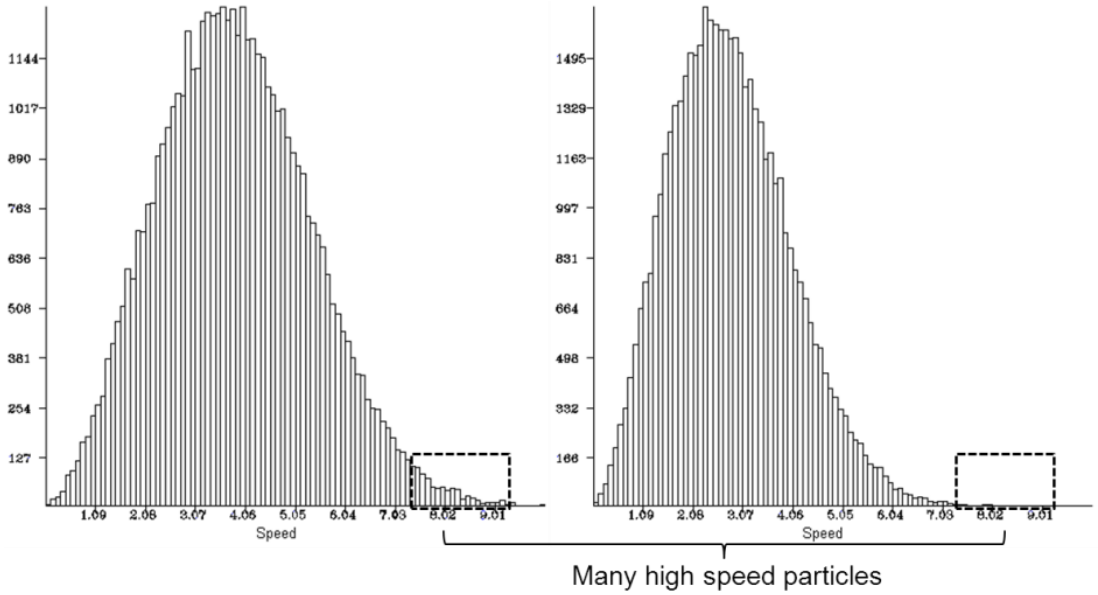


Figure 6-10: Speed Histogram Comparison

This assertion fails after several simulation cycles. Figure 6-10 reveals a large number of excessively fast particles when comparing actual results arising from bug-afflicted code and the expected behaviour. This anomaly is typically a result of using a large time step (Δt) [183]. Nevertheless, in this program, Δt is configured manually as a constant for the life of the simulation and should be sufficiently small to make this behaviour unlikely. Rather, it is suspected that Δt has been misused where the acceleration variable and velocity variable are computed. Closer investigation of the implementation of the Velocity-Verlet algorithm reveals an error (shown below) where the first Verlet's half-kick is performed. Correcting the code fixes this bug.

Incorrect code
`rv[i][a]=rv[i][a]+DeltaT*ra[i][a];`
Correct code
`rv[i][a]=rv[i][a]+DeltaTH*ra[i][a];`

6.2.2.2 Energy Conservation

The *law of energy conservation* states that the total amount of energy in an isolated system remains constant. In an LJ system, *forces* are time independent; thus the *total energy*, which is the sum of *kinetic* and *potential* energies, should stay approximately constant [178] compared to the initial provided total energy (via initial temperature). A drift of this quantity may therefore signal programming errors. To detect this, a *standard-deviation assertion* can be triggered after each simulation cycle to ensure that this quantity does not alter significantly. Standard-deviation reduction is triggered using the *set reduce* command as below.

```
set reduce stdev
graph $case2_g2
  assert $a::totEnergies@pmd.c:28 < 0.1
end
```

Figure 6-11: Case Study 2 - Assertion for Testing Conservation of Total Energy

The assertion above inserts a breakpoint at line 28 where a simulation step is just completed. When it is executed, the debugger extracts the `totEnergies` variable, which holds all the total energy values obtained so far, and performs the standard-deviation operation. After a few simulation steps, the assertion is violated, indicating that the total energy value has drifted. Because total energy is the sum of kinetic and potential energies, it is sensible to inspect the code where those variables are computed. Importantly, because a fixed temperature is kept throughout the simulation, even though kinetic and potential energies do not stay absolutely constant; they are expected to oscillate only slightly [178]. However, when monitoring the potential energy variable (`potEnergies`) for a few cycles using a similar standard-deviation assertion, we notice that this value does not oscillate, but rather increases steadily. The potential energy arises from the interactions of particles with each other. Therefore, it is necessary to inspect the segment of code where the interaction force is computed. This inspection reveals an error in the computation of the *force* variable. According to the force formula given below, the programmer has missed the `ri2` term (that is, r^2) in the code.

$$f_x(r) = \frac{48x}{r^2}\left(\frac{1}{r^{12}} = 0.5\frac{1}{r^6}\right) \qquad (12)$$

Incorrect code

```
fcVal=48*ri6*(ri6-0.5)+Duc/r1;
```

Correct code

```
fcVal=48*ri2*ri6*(ri6-0.5)+Duc/r1;
```

6.2.3 Case Study 2 – Summary

This case study has shown how scientific knowledge can be converted into ad-hoc debug-time assertions in order to monitor the progress of a target program. Even though the assertions did not directly locate the defects in the source code, they successfully highlighted the anomalies in the progress of the simulation. This information then enabled the user to scope the region of the defective code and identify the errors. In addition, this case study demonstrated the use of various statistical assertions including a standard-deviation and a histogram assertion.

6.3 Case Study 3: 2D Photon-Transport Simulation

This case study demonstrates how comparative assertions can be used to detect and analyse data variances between two code bases of the same application (that is, the classic or original reference compared to a later version).

6.3.1 Background

SPhot is a benchmarking code that describes a 2D photon-transport simulation [179]. Photons are generated in hot matter. The SPhot code is used to simulate the tracking of photons through a spherical domain that is cylindrically symmetric on a logically rectilinear 2D mesh. The code uses the Monte Carlo method to solve the Boltzmann transport equation. As a result, the particles' initial energy and direction are determined by using random numbers from appropriate distributions. Similarly, the scattering and absorption processes are also modelled by randomly sampling cross sections. Random numbers are generated using integer arithmetic.

SPhot is written entirely in Fortran77 and parallelised using both MPI and OpenMP libraries. It falls into the category of *embarrassingly parallel* applications in which "...every CPU that is employed in the computation works on a local copy of the 2D mesh (most likely stored in cache), generates its own random numbers and performs its own particle trackings" [179]. The only required communications are between the master MPI process and all other MPI tasks in order to distribute input data, update global variables and

collect various statistics. In addition, several `MPI_Barrier` calls are used for synchronisation purposes.

To show how comparative assertions can be used to debug SPhot, several random bugs are inserted into a version of the code called *sphot_buggy*. This version can be invoked and compared to a correct version called *sphot_working*.

6.3.2 Debugging SPhot Using Comparative Assertions

6.3.2.1 Error 1 - Incorrect Array Index

Below is the output of running sphot_working and sphot_buggy with 4,096 processes, respectively.

Table 6-4: Incorrect Outputs from SPhot Program

sphot_working	sphot_buggy
Total tracks = 66965828.0	Total tracks = 0.0
progTime = 2.58 sec	progTime = 0.05 sec
Sequoia Benchmark Version 1.0	Sequoia Benchmark Version 1.0
Total tracks / sec = 25989109.53	Total tracks / sec = 0.00
avg.esc.prob = 0.263122	avg.esc.prob = NaN
std dev = 0.000466	std dev = NaN

The program sphot_buggy generates a set of outputs that include NaNs (that is, Not a Number) and zero tracked particles. In contrast, the sphot_working program results in values that are both numeric and valid. This difference indicates a computing failure in some units during the course of the simulation. Using the data extracted from a reference code such as sphot_working can help in identifying the locations where some runtime values have drifted. The invocations of the two programs can be performed as follows:

```
invoke $ref[4096] ./sphot_working
invoke $sus[4096] ./sphot_buggy
```

While it is not clear exactly where the bug manifested, it is clear that the value `avg.esc.prob` was not computed correctly. Thus, an assertion comparing this key attribute (`g_ffesc`) between the two programs is useful. Because this key variable is distributed across 4,096 cores, we need a blockmap function to describe its global state.

```
blockmap case3_bm1(P::V)
  define distribute(block)
  define data(49152)
  define grid(4096)
end
graph $case3_g1
  assert case3_bm1($ref::g_ffesc@"writeout.f":121) =
         case3_bm1($sus::g_ffesc@"writeout.f":121)          (1)
end
```

Figure 6-12: Case Study 3 - Assertion 1

The above assertion acquires two sets of `g_ffesc` structures from `$ref` and `$sus` at line 121 of file `writeout.f`, and performs a comparison. The result shows differences between the datasets. Tracing the source code indicates that `g_ffesc` is computed by the following code snippet in `wroutput.f`:

```
52  trials = dble(npart)
53       do 498 i=1,12
54          wesc = wesc + enesc (i)
55          nesc = nesc + nescgp(i)
56 498   continue
57       ffesc = wesc/trials
58       g_ffesc(iRun) = ffesc
```

Therefore, the next data structure that needs to be verified is `enesc`. Accordingly, the following assertion can be used:

```
graph $case3_g2
  assert case3_bm1($ref::enesc@"wroutput.f":52) =
         case3_bm1($sus::enesc@"wroutput.f":52)             (2)
end
```

Figure 6-13: Case Study 3: Assertion 2

Again, the outcome of the assertion execution reveals differences between the two datasets. Tracing back through the code reveals that `enesc` is initialised and updated by the following lines of code in source file `execute.f`, respectively.

```
78  enesc(ijkl) = 0.0d0
461 enesc(ig) = enesc(ig) + newgt
```

Using this information, two assertions are devised to check the status of `enesc` after initialisation and the value of variable `newgt`. Because `newgt` is a scalar, another blockmap function is required to describe how multiple `newgt` values can be aggregated.

```
blockmap case3_bm2(P::V)
  define distribute(block)
  define data(4096)
  define grid(4096)
end
graph $case3_g3
  assert case3_bm2($ref::enesc@"execute.f":78) =
         case3_bm2($sus::enesc@"execute.f":78)                (3)
  assert case3_bm2($ref::newgt@"execute.f":461) =
         case3_bm2($sus::newgt@"execute.f":461)               (4)
end
```

Figure 6-14: Case Study 3: Assertions 3 and 4

The execution of the above graph indicates that the initialisation of variable `enesc` was performed successfully while the values for variable `newgt` differed. Again, the `newgt` value is assigned from `phnewgt`, which is worked out by variable `edep` as follows (in `execute.f`):

```
233 newgt  = phwgt
187 phwgt = edep / bwgt*npht
```

`edep` is a scalar computed via 2D array `efrac` and array `volcl`

```
162 edep = efrac(ir,ig) * volcl(iz)
```

The 2D array `efrac` represents the fraction of energy emitted in region `ir` of the 2D mesh. It is computed as follows:

```
91  efrac(ir,ig) = plnkut(u1,u2)*xkt4(ir)*sigtot(ir,ig)
```

Running several assertions to compare values of variables including `plnkut`, `xkt4` and `sigtot` indicates that `efrac` was computed incorrectly because of indices `u1` and `u2` (see line 91 above). As expected, the variable `u2` was miscalculated. Correcting the error eliminates the NaN results.

Incorrect code

90 u2=hnu(ig)*xtemp
Correct code
90 u2=hnu(ig + 1)*xtemp

6.3.2.2 Error 2 - Missing Parentheses

Even though the NaN results are resolved, the results of sphot_buggy still differ from the results of sphot_working (see Table 6-5 below).

Table 6-5: Sphot_buggy Missing Parentheses

sphot_working	sphot_buggy
Total tracks = 66965828.0	Total tracks = 66965828.0
progTime = 3.04 sec	progTime = 2.58 sec
Sequoia Benchmark Version 1.0	Sequoia Benchmark Version 1.0
Total tracks / sec = 22003623.22	Total tracks / sec = 25989109.53
avg.esc.prob = 989.116462	avg.esc.prob = 0.263122
std dev = 2.297236	std dev = 0.000466

Again, differences in variables such as `g_ffesc`, `enesc` and `newgt` can be examined. By running assertions (1), (2), and (3) above, we can confirm that those variables differ between the two programs. `newgt` is assigned by `phwgt`. Because `edep` is computed using `efrac` and `volcl` (as shown earlier), examining these two variables should reveal the location of the defect. Because `efrac` is a 2D array, the blockmap function `test3` below can be used by the debugger to reconstruct the global array in order to compare the two programs. Importantly, because the size of the global array `efrac` is large, a regular comparison scheme is not efficient. Therefore, we enable a point-to-point (P2P) comparison mechanism using the `set p2p` command.

```
blockmap case3_bm3(P::V)
  define distribute(block, *)
  define data(204800, 13)
  define grid(4096, 1)
end
set p2p true
graph $case3_g4
```

```
    assert case3_bm3($ref::efrac@"execute.f":92) =
           case3_bm3($sus::efrac@"execute.f":92)                (5)
    assert case3_bm3($ref::volcl@"execute.f":92) =
           case3_bm3($sus::volcl@"execute.f":92)                (6)
end
```

Figure 6-15: Case Study 3: Assertions 5 and 6

The outcomes of these two assertions suggest there is no difference between the two programs regarding these two vectors. This indicates that the value of `phwgt` is calculated incorrectly at line 187 of source file `execute.f`. Closer investigation reveals that a pair of parentheses is missing in the calculation. This defect leads to a small numerical error in computing `phwgt`. To fix this mistake, parentheses are required to force multiplication prior to division.

Incorrect code
`187  phwgt=edep/bwgt*npht`
Correct code
`187  phwgt=edep/(bwgt*npht)`

6.3.3 Case Study 3 - Summary

In our third case study, comparative assertions were successfully used to identify two programming bugs that exist in one version of the Sphot benchmark, but not in another. In the first case, several comparative assertions were used to narrow down the scope of the error by confirming the incorrect values of some key variables. In the second case, similar activities were performed on different variables. Because of the large data structures, an advanced comparison scheme – the P2P comparison method – was deployed to improve the efficiency of the debugging technique. Both of the errors found represent a class of programming error that frequently occurs when programs are modified.

6.4 Summary

The three case studies presented in this chapter use a range of ad-hoc debug-time assertions to demonstrate the power of assertion-based debugging. The case studies show how the technique can help developers to identify coding bugs ranging from array-indexing problems to incorrect values for loop bounds, and from missing terms in

computation to missing parentheses. Importantly, different properties of the target programs (such as the programming models used, the science driving the computational code, and the availability of earlier versions of the same application for referencing purposes) can be used effectively to guide the debugging process. All the case studies involved numerous processes, large amounts of data and parallel architectures.

This chapter, through the use of case studies, has shown the applicability of assertion-based debugging. The following chapter presents a performance analysis and evaluation of the debugging tool. To illustrate the scalability of the debugging technique, a Cray supercomputer is used.

Chapter 7

Performance Evaluation

This chapter evaluates and analyses the performance of the proposed debug-time assertions in debugging production codes. This chapter is organised as follows:

- Section 7.1 describes the experimental setup. It presents the configuration of the test platform, justifies the target problem size and explains the measurements used. In particular, both strong scaling and weak scaling experiments are performed to demonstrate the scalability of the proposed solutions.
- Section 7.2 to Section 7.4 present the experiments and analyse the performance results for different debug-time assertion templates including general ad-hoc assertions, statistical assertions and comparative assertions. These results demonstrate that our debug-time assertions retain acceptable performance when both the number of processing units and the problem size increase.
- Section 7.5 summarises the chapter.

7.1 Experiment Description

7.1.1 Testbed Configuration

The performance of the debug-time assertions were evaluated on "Hera", a Cray XE6 system with more than 20,000 cores. Whilst not in the petascale class, this system is sufficiently large to provide performance information that could be extrapolated to larger machines. Table 7-1 characterises the system-configuration information for the test platform. Some of the tests required two sets of processes (for example, when evaluating the comparative assertions that involve two programs) and these experiments were run using up to 10,000 processors (in pairs); the maximum available on Hera.

Table 7-1: XE6 – Hera's Configuration

	CPUs/Node	Clock (GHz)	Nodes	Total Cores
	32	2.1	96	3072
	32	2.1	308	9856
	32	2.1	228	7296
	32	2.1	4	128
	32	2.2	32	1024
	32	2.3	16	512
	12	1.9	8	96
Total			692	21984

To demonstrate that our debugging approach is applicable to large-scale scientific codes, information about data-structure sizes in WRF (a production-climate model [159]) is used. WRF models the atmosphere and performs meso-scale numerical weather prediction (NWP) [159]. In order to solve the equations of motion in the atmosphere, WRF manipulates a number of data structures that represent variables such as temperature, pressure and velocity fields. These are typically arrays with three indices representing the co-ordinates of the 3D space. When run on a parallel machine, these arrays are decomposed and distributed across the processors using techniques similar to those discussed in Section 4.3. We chose to model a recent experiment in which the WRF was run on a Cray XT5 (Jaguar at Oak Ridge National Laboratories) with 150,000 compute cores [159]. Each of the data structures of interest consisted of a `40x40x50` array of single-precision floats per processor, amounting to 320 Kbytes. Consequently,

with the 20,000 cores available to us, we generated a 6 Gbyte global structure. Using these parameters, two experiments were created for each assertion – a strong scaling experiment and a weak scaling experiment. The results were also used to compute the improvement in speed, or *speedup*. Strong scaling and weak scaling experiments are described in the following sections.

7.1.2 Strong Scaling Experiments

In strong scaling experiments, the global problem size stays unchanged while the capacity of the processing resource is increased (that is, the number of computer processors escalates). The amount of data distributed to each processor therefore reduces throughout the experiment. Such experiments can be used to determine the point at which the parallel overhead becomes insignificant and allows the computation to complete in a reasonable amount of time.

In addition, through a strong scaling test, speedup can be calculated. We set the global problem size at approximately 6 Gbytes. Because of memory constraints, we cannot perform the strong scaling experiments for a small amount of computing resources. In fact, the least number of cores required for the strong scaling test is 64. Consequently, perfect speedup is assumed for a smaller number of processing cores. Thus if a timing result for running an experiment with 64 cores is t, the timing result for 32 cores will be $2t$. In addition, if the amount of time to execute an assertion with one processing core is t_1, and the amount of time to complete the same assertion with p processing cores is t_P, the strong scaling efficiency (that is, speedup) is given as:

$$p * (t_P / t_1) * 100\% \qquad (13)$$

7.1.3 Weak Scaling Experiments

In weak scaling experiments, the problem size (or workload) assigned to each processing core stays constant; increasing the number of processing cores thus allows a larger overall problem to be solved. This type of measurement is used to justify whether the proposed tool or algorithm can cope with a large problem given a sufficient amount of processing elements. In the case of weak scaling, linear scaling is observed when the runtime stays constant while the workload increases in direct proportion to the number of processors. In addition, results from weak scaling experiments can be used to prove

that given more computing resources, debugging larger programs with our assertion-based framework is still practical.

7.1.4 Performance Metrics

In our assertion-based debugging framework, the execution of an assertion consists of a number of phases (described in Section 3.2). Of these, data reduction and data collection are the most time consuming. First, after receiving the command from the debug client, the remote debug servers obtain data from GDB and perform the requested data-reduction operation. This step proceeds in parallel because each debug server works independently. Second, through the network infrastructure, data is reduced, collected and returned to the debug client. Because each debug server performs its tasks and returns the result to the debug client independently of the others, the overall communication phase is carried out in a pipeline fashion. As a result, timing the communication phase separately is difficult and error prone. Therefore, we group both data-reduction time and data-communication time into one measurement and call it the ***server time***. Finally, the client executes post data-processing tasks and performs the final reduction sequentially. Time spent on this phase is noted as the ***client time***. The total assertion time is indicated by the ***overall assertion time***.

7.2 Evaluation of General Ad-Hoc Assertions

Chapter 3 presents several use-case scenarios for general ad-hoc assertions including assertions to (1) verify the state of the master process after the initialisation routine; (2) verify successful data decomposition performed by the master process on slave processes; (3) verify successful communication between slave processes; (4) validate slaves' states across time steps; and (5) confirm correct reduction of data across many slave processes. In this section, we examine the performance of assertion (2) and assertion (5). These are selected because their execution can be carried out in parallel.

First, assertion (2) checks each element in the global dataset (which has been decomposed across many slave processes) against a constant. The assertion can thus be evaluated at the backend where each debug server will perform the comparison in parallel. The server only reports if differences are found. Importantly, comparison results collected from the debug servers need to be combined to reconstruct the global view of the assertion outcome. A blockmap function is provided for this assertion. We refer to

this assertion in the discussions below as a *simple compare assertion*. Second, assertion (5) verifies that the average value of all elements in the global structure is within certain bounds. This assertion can be evaluated in parallel because each debug server goes through the data elements in their part of the global structure and perform the reduction using the average function. Sub-average results are collected and passed to the debug client and are reduced once more to determine the average value of the global dataset. No blockmapping is needed for this assertion. We refer to this assertion as the *average assertion* in the following subsections. The script for running these two assertions is given below[7] (see Figure 7-1). As explained above, we conduct both strong scaling and weak scaling experiments.

```
blockmap bm(P::V)
  define distribute(block, *)
  define data(16384,50000)
  define grid(8192,1)
end
invoke $a[8192] ./simple
set reduce average
graph $g
  assert bm($a::array@"simple.c":35)=10 #simple compare assertion
  assert $a::array@"simple.c":35 = 5    #average assertion
end
```

Figure 7-1: Script for Testing General Ad-hoc Assertions

7.2.1 Strong Scaling Results

In Figure 7-2 and Figure 7-3, we measure the elapsed time for the *average assertion* and the *simple compare assertion*. We show these times on a log scale. Note that for a small number of processors, the overall assertion time is around 100 seconds. This is mostly the cost for each debug server to query and process large amount of raw runtime data (that is around 96 Mbytes given 64 processors) obtained from GDB instances. For small number of processors, the overall assertion time is strongly dominated by the server time, as shown in the figures.

[7] The assertion given is setup for 8,192 cores. The number of cores can be changed using the *invoke* commands and the *grid* definition of the *blockmap* function can be updated accordingly. The average reduction mode can be switched on and off using the *set reduce average* command.

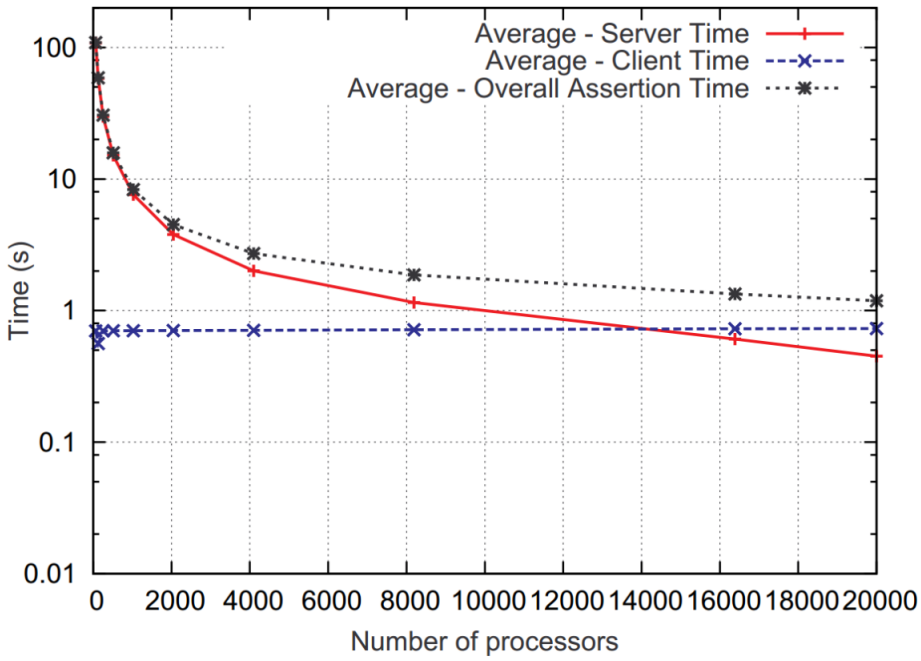


Figure 7-2: Average Assertion -Strong Scaling Results

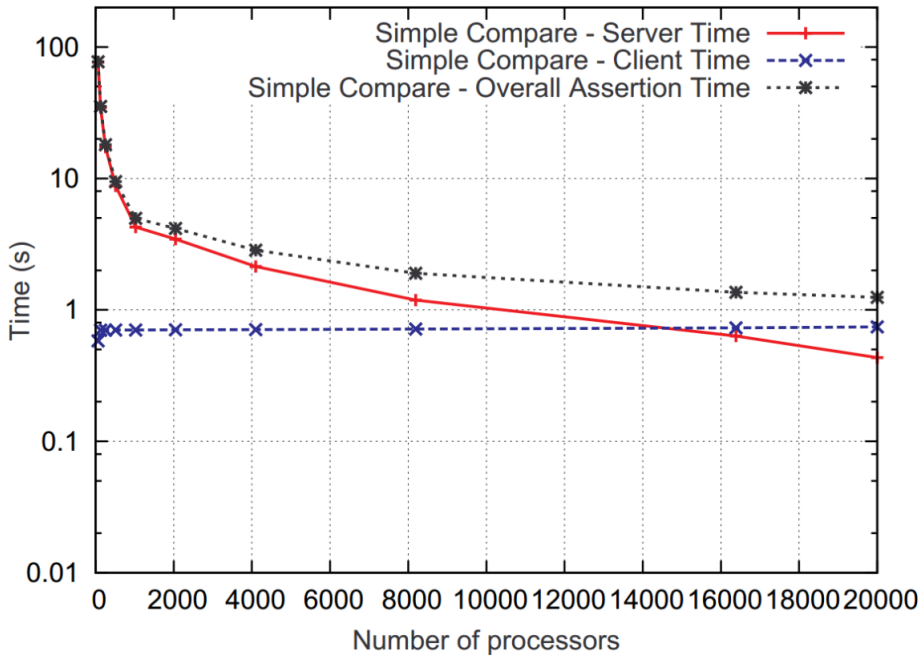


Figure 7-3: Simple Compare Assertion - Strong Scaling Results

Both assertions generate similar performance trends in which the server times fall as the number of processors increases. This is expected because more processes indicate that less data is processed by each backend debug server. On the other hand, the client times increase as processor numbers escalate, even though the increases are small and the client time curves seem flat. The client performs tasks sequentially and the workload is proportional to the number of partial results collected from the debug servers. Therefore, more debug servers indicate a longer client time. Actual timing data can be

found in Table 7-2 below and in *Appendix B – Performance Data*. More importantly, because the client times in both figures are around one second and keep increasing, they become significant at 14,000 cores. This is most likely the communication start-up latency of the MRNet middleware, which is tuned for large-scale network communication and uses socket connections between nodes. This limits the speedup (shown in Figure 7-4).

Another important note is that the average assertion achieves better speedup than the simple compare assertion (shown in Figure 7 4). This is because the simple compare assertion needs to rebuild the global view of the result while the average assertion only needs to compute the final average value using the sub-averages collected from the debug servers. Despite these caveats, the debugging tool achieves reasonable speedup overall, and more importantly, reduces the assertion execution time to the order of seconds, making it feasible as an interactive debugging aid.

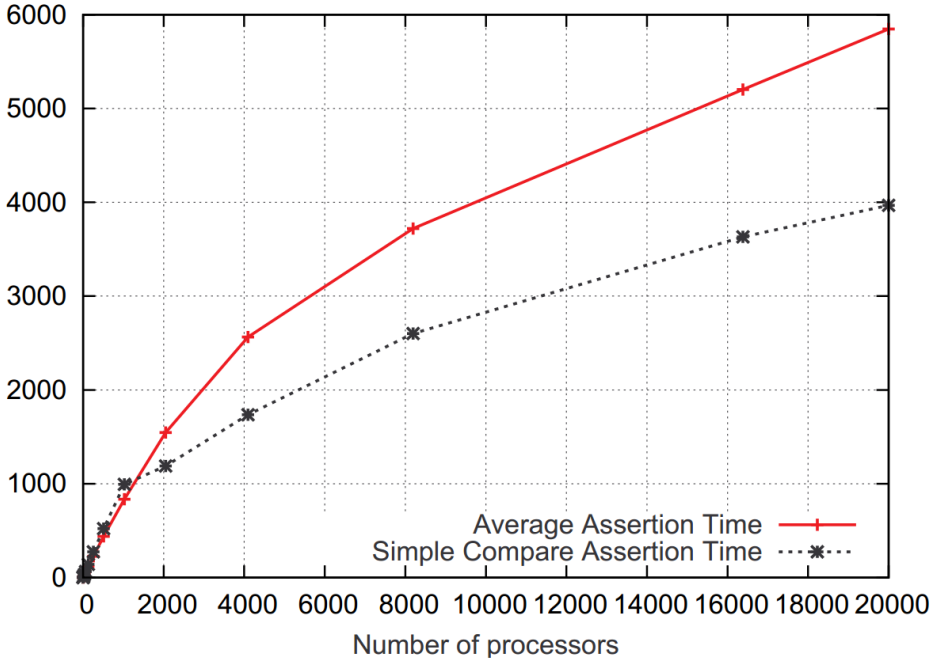


Figure 7-4: General Ad-hoc Assertions - Speedup Against #Processors

Table 7-2: General Ad-hoc Assertion - Raw Strong Scaling Results

Number of Cores	Average Assertion	Simple Compare Assertion
64	108.6052	77.2796
128	58.69077	35.30655
256	30.55389	18.02769
512	15.80661	9.470287

1024	8.31842	4.974198
2048	4.496775	4.158439
4096	2.711667	2.849367
8192	1.868058	1.901147
16384	1.335841	1.362211
20000	1.188727	1.246494

7.2.2 Weak Scaling Results

In this experiment, when we double the number of processors (that is, more data is involved), the assertion evaluation time rises slowly, as shown in Figure 7-5. For both assertions, the server times remain fairly constant in this experiment. In contrast, the client times increase. As indicated above, the client time for the simple compare assertion grows faster than that of the average assertion and this is shown at around 8,000 cores. In addition, for the average assertion, the curve starts to level out at 8,000 cores. This implies that given more computing resources, conducting an assertion with a simple reduction function (such as the average function) on large data structures is still practical. With a simple compare assertion, a bottleneck arises due to the reconstruction of the global structure. However, at around 20,000 cores, handling 6 Gbytes of data takes less than 1.4 second.

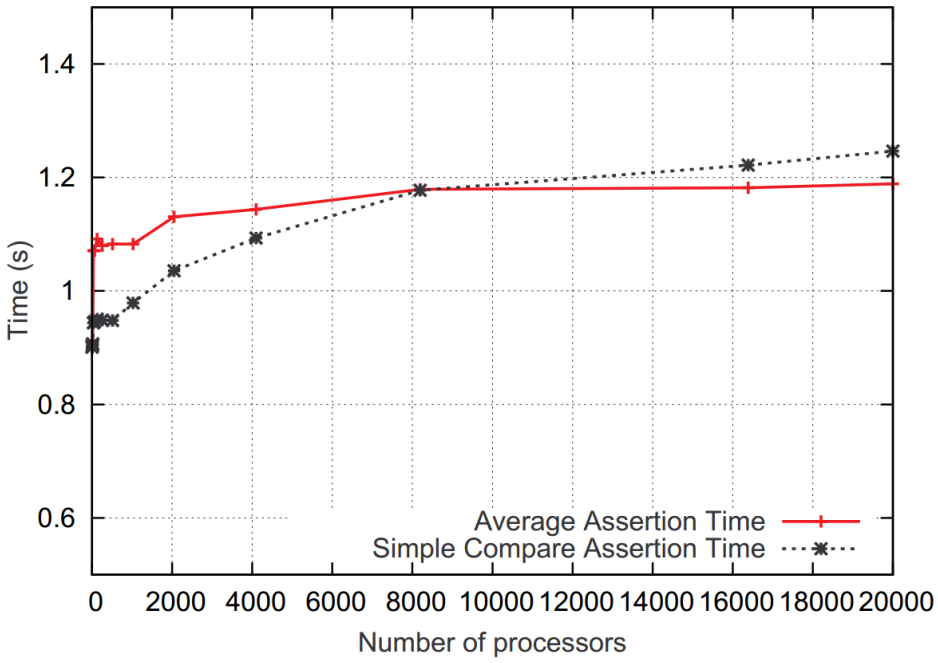


Figure 7-5: General Ad-hoc Assertions - Weak Scaling Results

7.3 Evaluation of Statistical Assertions

In this section, three statistical assertions are evaluated using similar experiments. The first assertion computes the standard-deviation value given an array of floating-point numbers, and compares it to a constant. More importantly, the standard-deviation function is built within the debugger and can be called directly by the debug servers. We call this assertion the *built-in stdev assertion*. The second assertion performs the same computation as the built-in stdev assertion. However, the standard-deviation function is implemented by user code and it is only loaded and invoked by the debug servers dynamically at runtime. This technique can be conducted using the split-phase operation (described in detail in Chapter 4). We refer to this assertion later in the text as the *user-defined stdev assertion*. The last assertion constructs a histogram from debug data and compares it to a *user-defined data model*. We call this assertion the *histogram assertion*. The script used to define the three assertions is given in Figure 7-6[8].

```
invoke $a[8192] ./stat
set reduce stdev                          # built-in stdev mode
#register $TEST/assertion/my_stdev       # user-defined stdev mode
#create $model  # histogram assertion
#assign $model = randset(guassian,49152)
#set reduce histogram 1000 0.0 0.1
graph $g
   assert $a::array@“stat.c”:35 = 0.1 #built-in stdev
   assert my_stdev($a::array@“stat.c”:35)=0.1 # user-defined stdev
   # histogram assertion
   assert $a::array@“stat.c”:35 ~ $model = 0.1
end
```

Figure 7-6: Script for Testing Statistical Assertions

7.3.1 Strong Scaling Results

Figure 7-7 through to Figure 7-9 present the elapsed time for executing the three statistical assertions on a log scale. The results show that the overall assertion times are dominated by the server times for all three assertions. As the number of processors

[8] No blockmap function is required for statistical assertions. Built-in reduction functions such as *stdev* or *histogram* can be switched on and off using the *set reduce* command. The user-defined reduction function is used through the *register* command.

increases, this measure falls as expected. The client times grow as the number of processors increase, because a number of tasks are performed sequentially by the debug client. However, in the built-in stdev and user-defined stdev assertions, client times are relatively small and barely affect the overall assertion times, while client times increase noticeably in the histogram assertion. This is due to the cost of constructing two histograms and comparing them sequentially. Therefore, at around 512 cores, for histogram-assertion curves, the server time diverges from the overall assertion time. Further, the client time becomes larger than the server time at around 4,000 cores. This results in the overall assertion time for the histogram assertion levelling out instead of reducing further, even with more processors added. Consequently, the built-in stdev and user-defined stdev assertions deliver better speedup compared to the histogram assertion, as indicated in Figure 7-10. The overall histogram-construction procedure can be further tuned to improve the speedup for the histogram assertion. Nevertheless, the debug tool achieves a good speedup overall (Figure 7-10), and more importantly, reduces the assertion execution time down to an order of seconds, indicating its feasibility as an interactive debugging aid.

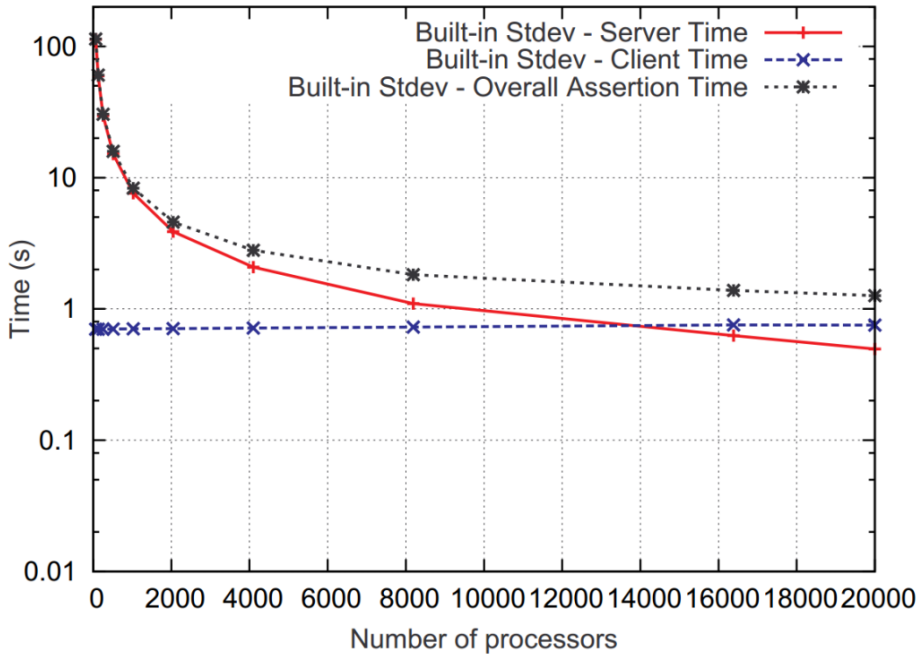


Figure 7-7: Built-in Stdev Assertion - Strong Scaling Results

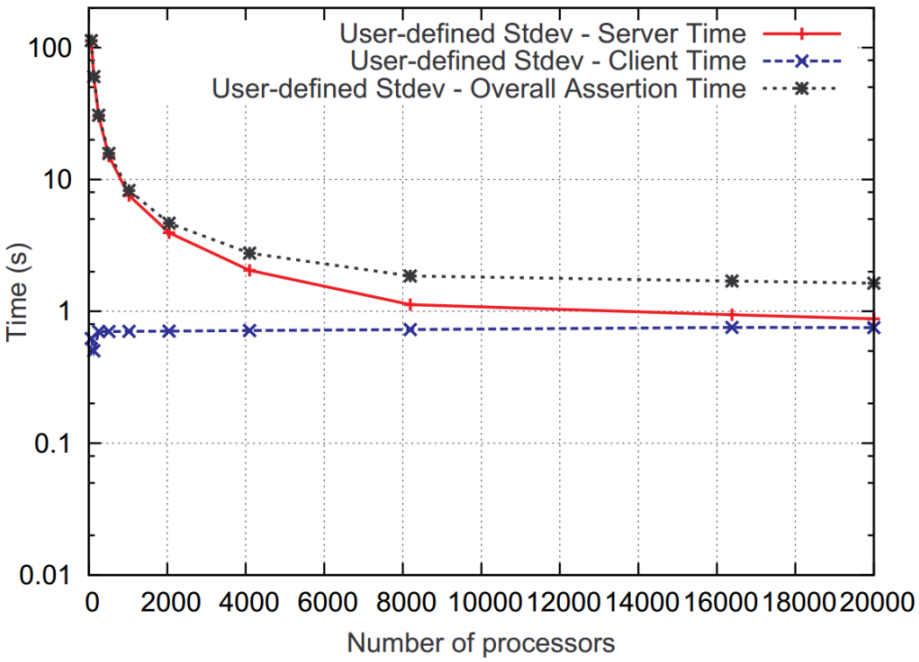


Figure 7-8: User-defined Stdev Assertion - Strong Scaling Results

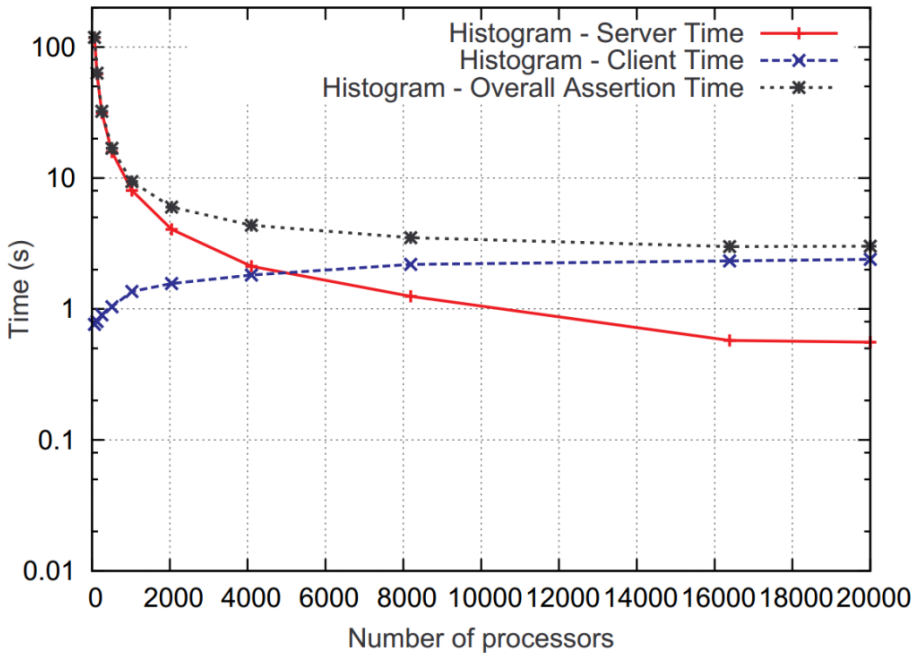


Figure 7-9: Histogram Assertion - Strong Scaling Results

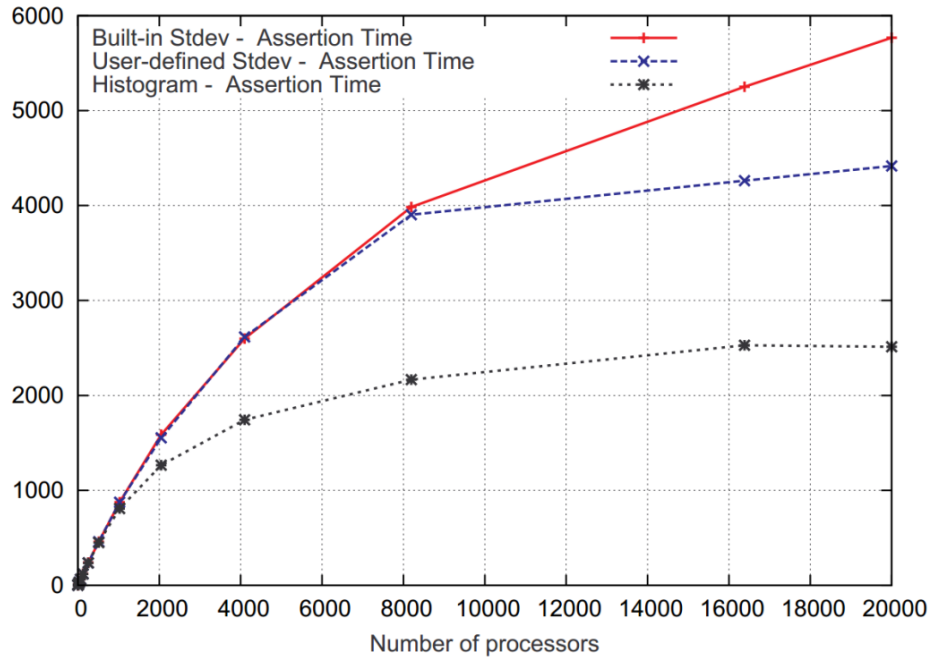


Figure 7-10: Statistical Assertions - Speedup Against #Processors

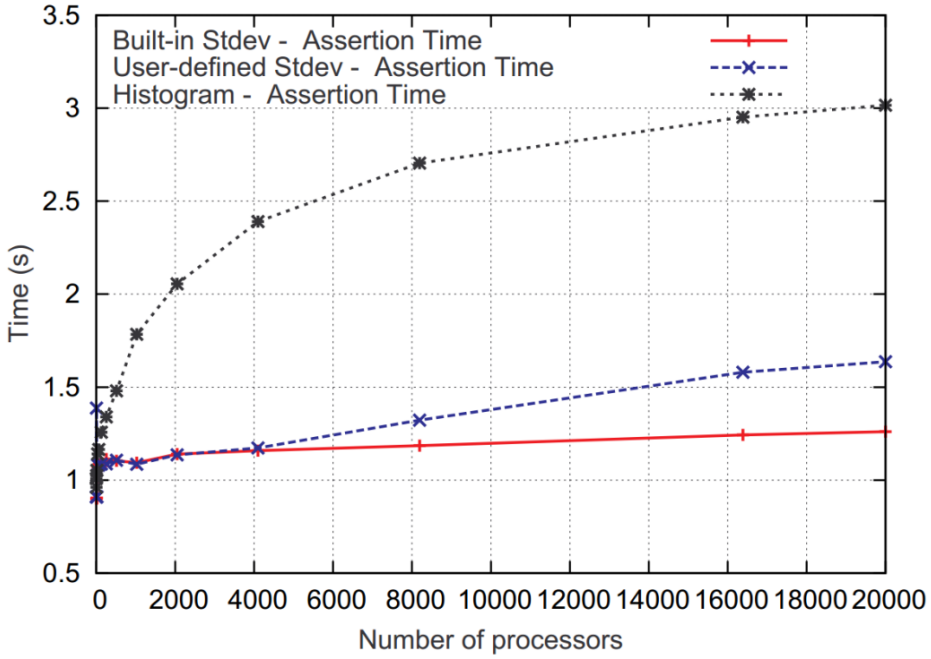


Figure 7-11: Statistical Assertions - Weak Scaling Results

7.3.3 Built-in Function vs User-Defined Function

The difference between using a built-in function and using a user-defined function for a statistical assertion is how the function is invoked at runtime. While a built-in function can be called directly, a debug server needs to load a shared-library object at runtime to invoke a user-defined function. This leads to a bottleneck because many debug servers may try to access a centralised shared-library object simultaneously.

Figure 7-12 compares the scaling performance between the two approaches when invoking a different number of processors. As the number of processors doubles, the time required to execute a user-defined stdev assertion becomes bigger compared to that of the built-in stdev assertion. This observation verifies that in executing a user-defined stdev assertion, more processors try to access a shared object and consequently slow down the computation of partial standard-deviation values at the backend. Nevertheless, the difference is relatively small (even though it affects the speedup of the user-defined stdev assertion (depicted in Figure 7-10)). Despite this limitation, at 20,000 processors, both assertions can be completed in less than two seconds.

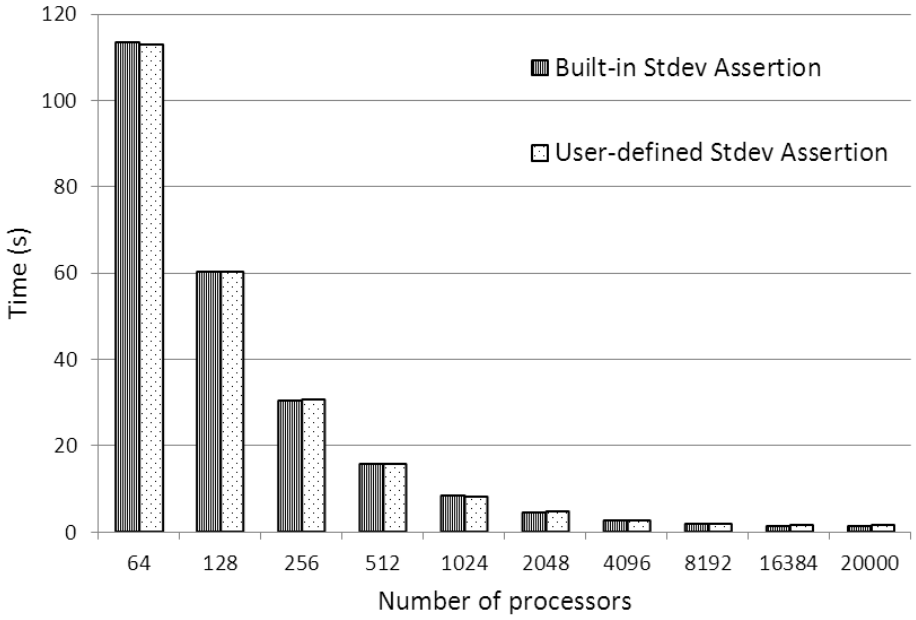


Figure 7-12: Built-in Stdev Assertion v.s User-defined Stdev Assertion

To conclude, even though the current implementation of the split-phase operation and the use of a shared-library object allow dynamic extensions of the statistical functions, these limit the performance of assertions such as the user-defined stdev assertion. Nonetheless, the implementation still returns an acceptable performance and demonstrates that the technique can be used as a practical debugging aid.

7.4 Evaluation of Comparative Assertions

Here we evaluate and analyse the performance of comparative assertions. As discussed in previous chapters, executing comparative assertions that involve large data structures on large parallel machines requires using either the hash-based comparison scheme or the

direct P2P comparison scheme. Figure 7-13 shows a script for running a comparative assertion with either comparison scheme across a number of different compute cores[9]. First, we present the performance results for the hash-based comparison scheme and then deliver the performance results for the P2P comparison scheme. We then compare the performance between the two. This comparison is conducted given two different scenarios: the best scenario where the numbers of processes invoked by the reference program and the suspect program are the same (such as the assertion in Figure 7-13) and the general scenario where the suspect program invokes twice the number of processes of the reference program.

```
blockmap bm(P::V)
  define distribute(block, *)
  define data(16384,50000)
  define grid(8192,1)
end
invoke $a[8192] ./comp1
invoke $b[8192] ./comp2
#set reduce fnvhash
set p2p true
graph $g
  assert bm($a::array@"comp1.c":35)=bm($b::array@"comp2.c":35)
end
```

Figure 7-13: Comparative Assertion – Script for Testing Best Scenario

To fully characterise the performance of each approach, the various activities that contribute to the overall elapsed time are presented in the following table.

Table 7-3: Time Breakdown for Hash-based Scheme and P2P Scheme

P2P Scheme	Server time = GDB+$P2P_{Map}$+$P2P_{Comm}$+Comp+$Client_{Comm}$
	Client time = Blockmap + Result_Display
Hash-based Scheme	Server time = GDB + Hashing + $Client_{Comm}$
	Client time = Blockmap*2 + Comp + Result_Display

where:

GDB: Time for collecting runtime data from GDB

$P2P_{Map}$: P2P mapping + preparing sub-blocks

[9] The assertion given is setup for 8,192 cores. The P2P mode and the hash-based mode can be switched on and off using the *set reduce* and *set p2p* commands.

$P2P_{Comm}$: Communication between debug servers

$Client_{Comm}$: Communication time between servers and client

Hashing: Data hashing time at servers

Comp: Data Comparison time

Blockmap: Data reconstruction time in client

Result_Display: Time for displaying assertion result

Because a comparative assertion requires two sets of processes, these experiments were run with up to 10,000 processors (in pairs). This is half of the maximum size of the test machine and allowed us to debug a suspect program against a reference program of the same size.

7.4.1 Evaluation of the Hash-based Comparison Scheme

In Figure 7-14 and Figure 7-15, we measure the elapsed time for evaluating comparative assertions using the hash-based comparison method. We also sketch three major timing components: the *overall assertion time*, the *server time* and the *client time*. Details of the activities that comprise these measures can be found in Table 7-3 above.

7.4.1.1 Strong Scaling Results

In this experiment, we keep the total data-structure size constant at 320 Kbytes*10,000 processors; around 3 Gbytes. In a strongly scaled experiment, as the number of processors is increased, the amount of data per core decreases.

Results in Figure 7-14 reveal that when the number of cores increases, the server time, as well as the overall assertion time, decreases. This is not surprising because a larger processing resource enables more data to be processed (hashed) in parallel. However, because the server time comprises not only the hashing time by debug servers (Hashing) but also the hash-signatures transfer time ($Client_{Comm}$), the greater number of debug servers implies more data needs to be transferred to the debug client. This results in the server time decelerating its decrease after around 4,000 cores. Furthermore, the client time also increases proportionally with the number of processors. This is because the hashing method requires the rebuilding of the global data structures from both programs and also includes the serial comparison time that doubles as the number of processors doubles. These factors limit the speedup for the hash-based technique after 4,000 cores, as depicted in Figure 7-19.

7.4.1.2 Weak Scaling Results

In the weak scaling experiment, the amount of data assigned to each processor remains constant and thus the total amount of data grows as the number of processors increases. Starting at 320 Kbytes for one processor, the total memory increases to 3 Gbytes at 10,000 processors.

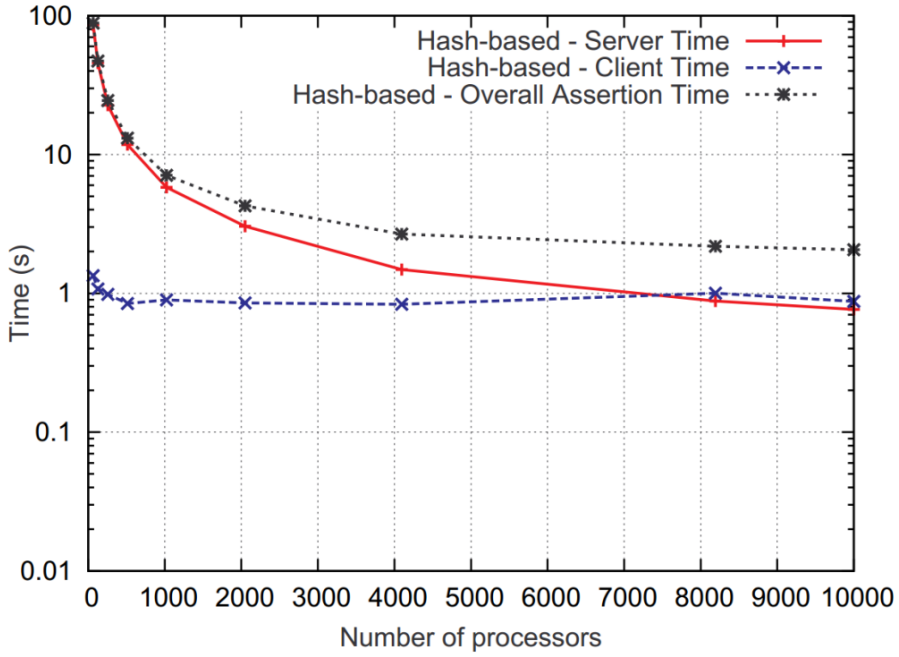


Figure 7-14: Hash-based Comparison Scheme - Strong Scaling Results

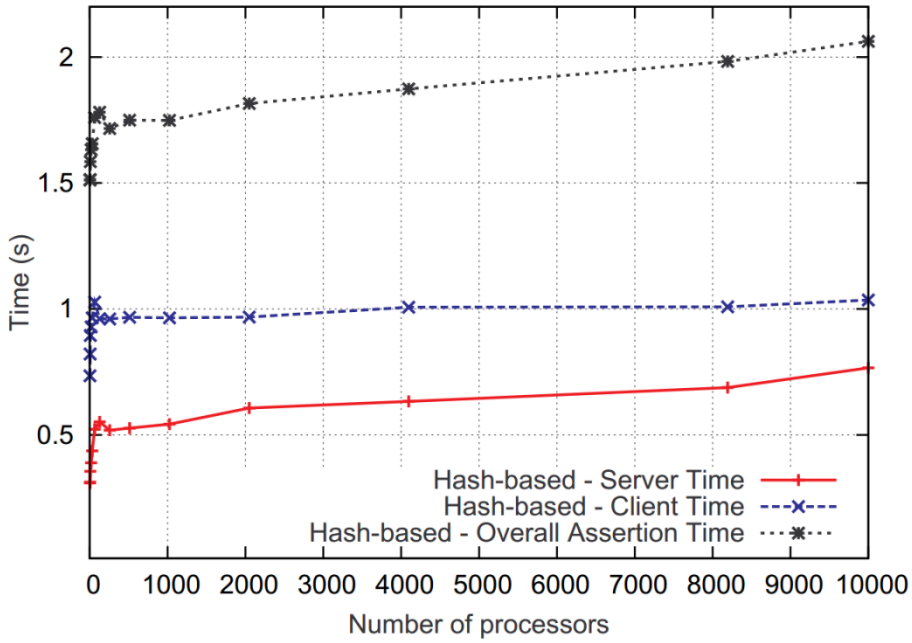


Figure 7-15: Hash-based Comparison - Weak Scaling Results

Because the amount of data assigned to each debug server is small (and thus hashing is fast), the server time is always smaller than the client time and the overall assertion time

increases slowly as we double the number of processors (shown in Figure 7-15). The client time, although dominating the overall assertion time, increases slowly as well. The server time seems to have the stronger increasing rate because more hash signatures are transferred to the debug client. However, this is linear growth (proportional to the number of processors); therefore, we estimate that server time will only catch up with the client time at around 100,000 processors, which should be less than two seconds. The result of the weak scaling experiment thus indicates that the hash-based comparison scheme is sustainable for problems and machines of petascale type.

7.4.2 Evaluation of the P2P Comparison Scheme

7.4.2.1 Strong Scaling Results

Figure 7-16 reveals that as the number of cores increases, the server time and the overall assertion time reduce for the P2P comparison method. This is similar to the results achieved in the evaluation of the hash-based comparison scheme. The reduction in time occurs because of the parallelism gained through having more debug servers working on a global problem with a constant size. However, at around 4,000 cores, the server time decreases with the lower rate. Two potential reasons can be identified for this observation. First, because P2P communication is currently implemented with a basic socket protocol, more debug servers try to communicate with each other results in a higher network-traffic load and thus limits the parallelism gained. Second, a greater number of comparison results are collected and transferred to the debug client. That may also slow down the overall server time. However, we still gain good speedup at 20,000 processors, as shown in Figure 7-19.

7.4.2.2 Weak Scaling Results

The result of the weak scaling experiment shows that the P2P method incurs increasing overall assertion times (Figure 7-17). This is the result of having both the server time and client time increase as the number of processors increases. However, at 4,000 cores, the server time starts to level out while the client time continues to slowly increase. Again, weak scaling implies that a constant amount of data is assigned to each debug server and that extra sequential jobs are assigned to the debug client. In other words, the same amount of data is transferred between two debug servers but an increasing amount of P2P comparison results is collected and processed by the client. Nevertheless, while the weak scaling experiment result shows linear time complexity for the P2P comparison

method, the increasing rate is relatively low and at 20,000 processors, the assertion takes less than two seconds to process.

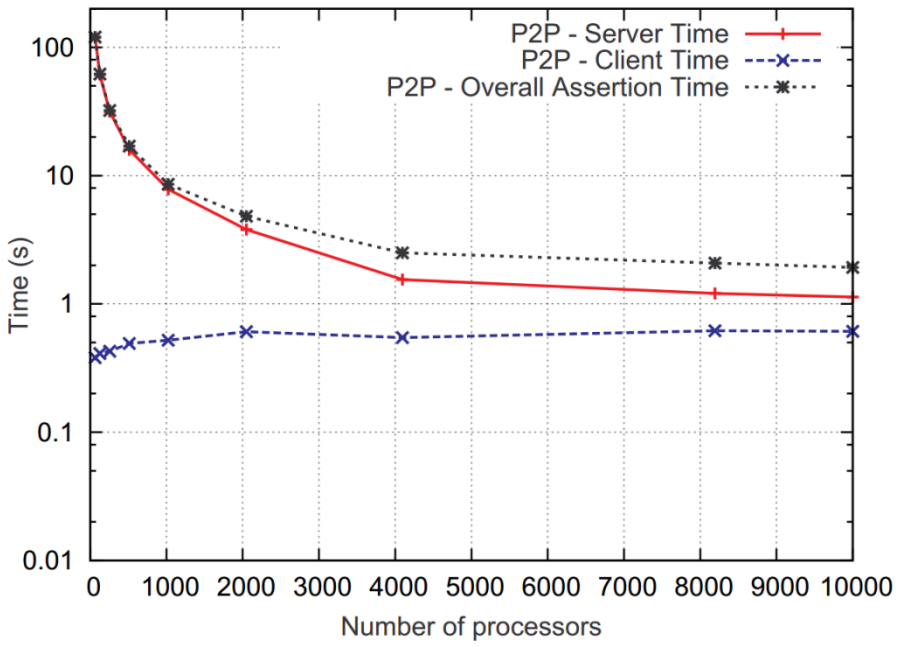


Figure 7-16: P2P Comparison Scheme - Strong Scaling Results

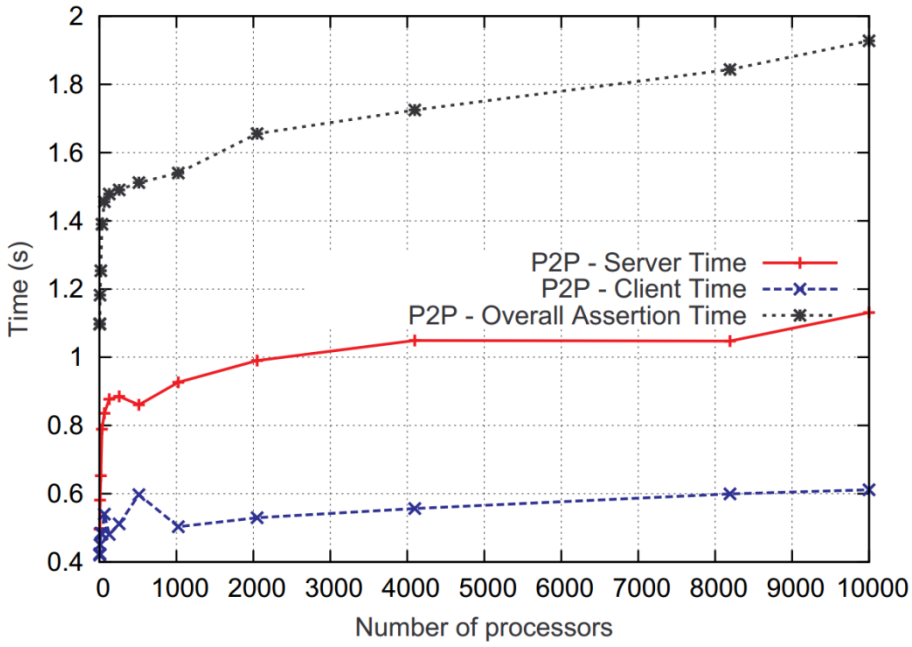


Figure 7-17: P2P Comparison Scheme - Weak Scaling Results

7.4.3 P2P Comparison vs Hash-based Comparison

We now assess the performance of the P2P comparison technique against the hash-based comparison technique. This assessment allows us to see the scalability of both solutions and reveals when it is more suitable to use a particular technique over the other. The

results from strong scaling and weak scaling experiments with both techniques are combined and shown in Figure 7-18 through to Figure 7-20.

In Figure 7-18, the P2P overall assertion time was initially much longer than that of the hash-based overall assertion time. When only a small number of debug servers are invoked in the execution of the comparative assertion, it seems the time needed to hash a large amount of data is less than the time required for transferring the data using a socket connection. However, at 4,000 cores, the P2P scheme becomes faster than the hash-based scheme. This is because of two reasons. First, in the P2P scheme, the amount of data transferred between two debug servers becomes insignificant. Second, the amount of data that needs to be communicated back to the client (and processed) in the hash-based scheme overtakes the amount in the P2P scheme. Figure 7-19 shows the results as speedup curves and highlights the improved scalability of the P2P scheme over the hash-based scheme.

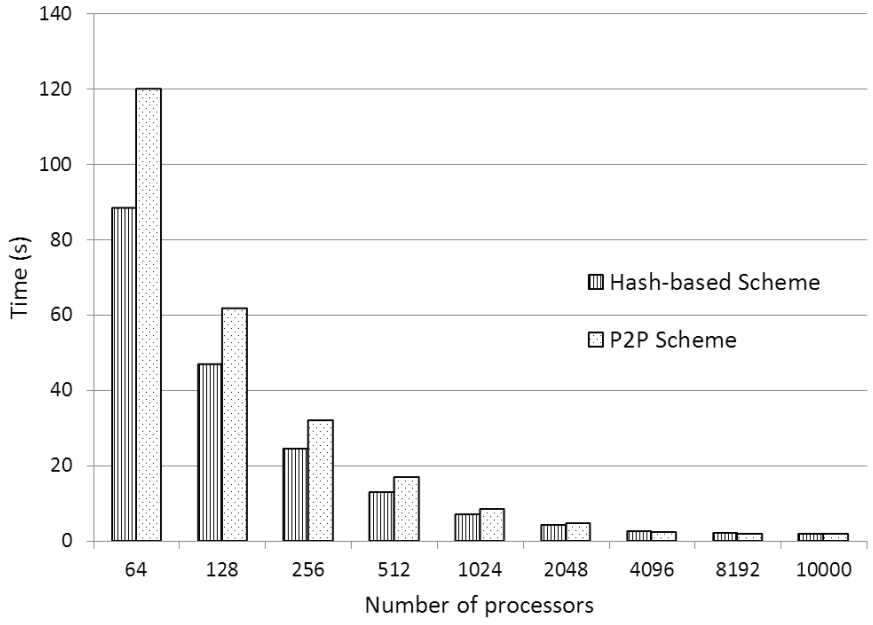


Figure 7-18: Overall Comparative Assertion Times for Best Scenario

Figure 7-20 compares the weak scaling behaviours of the two solutions. Here, both hashing and P2P methods incur increasing overall assertion times as the number of processors increases. However, because the amount of data distributed to each debug server remains unchanged, the P2P overall assertion time was always smaller than that of the hash-based overall assertion time. This confirms that the combination of the communication time between client and servers ($Client_{Comm}$) and the client processing time for hashing is more significant (compared to the P2P method) in the overall

assertion time. Nevertheless, both methods produce acceptable execution times for runs with 20,000 processors (10,000 processors for each program), which indicates their potential as practical debugging aids.

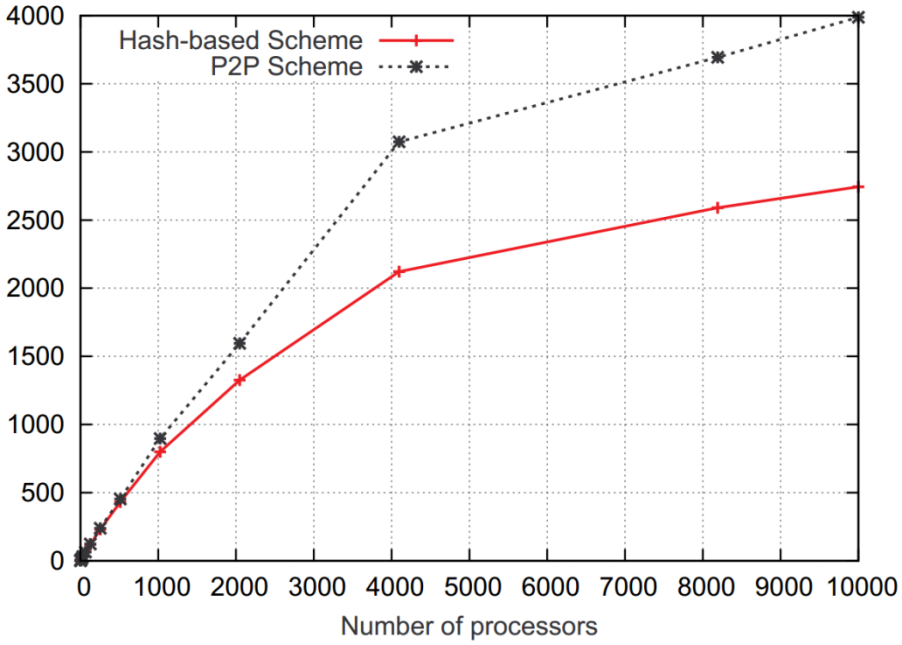


Figure 7-19: Comparative Assertion - Speedup Against #Processors

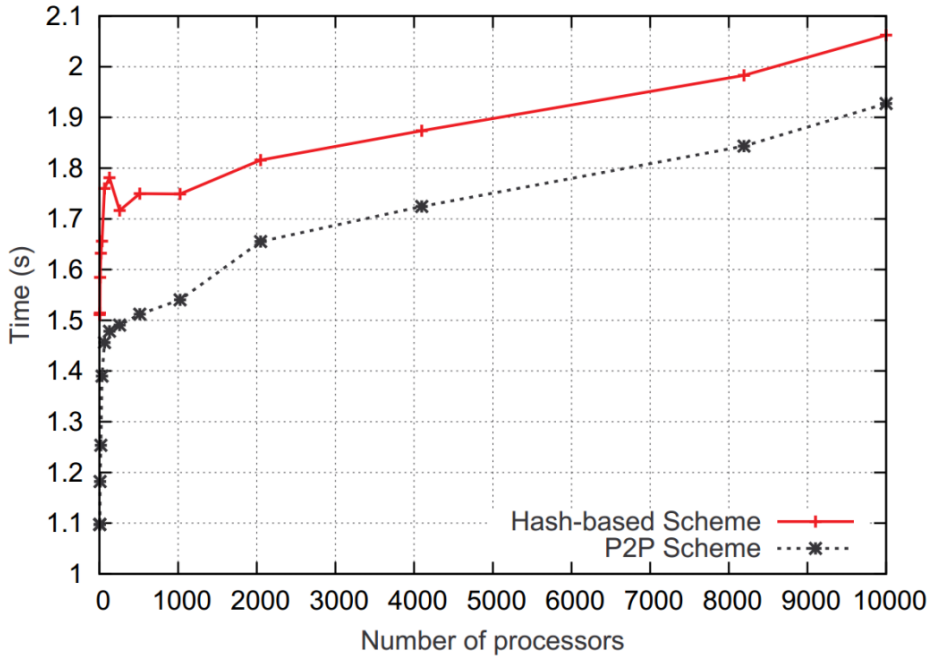


Figure 7-20: Comparative Assertions - Weak Scaling Results

7.4.4 Performance for Comparative Assertions under the General Scenario

As discussed in Section 3.2, parallelising the comparison phase when the two programs use the same decomposition strategy, and the same number of processors, is relatively straight forwards because processes communicate in pairs. However, when two programs use different decomposition schemes and a different number of processing cores, this becomes more complex. It is common for developers to scale up the program until abnormal behaviour is noticed. Then the abnormal run is compared to the last successful run in order to determine where, and how, the incorrect results were introduced. Furthermore, it is common to scale the number of processors as a power of two. In this research, we wish to compare a program running `p` (working) and `2p` (failing) processes[10]. In this case, every debug server communicates with two others. In Figure 7-22 and Figure 7-23, the `X axis` depicts the value of `p`. For example, an `X axis` value of 5,000 means that 5,000 debug servers are communicating with 10,000 others; a total of 15,000 debug servers. Figure 7-21 shows the script used for this experiment. Two blockmap functions (`bm1` and `bm2`) are required because of the different number of processes used for program `comp1` and program `comp2`.

```
blockmap bm1(P::V)
  define distribute(block, *)
  define data(16384,50000)
  define grid(8192,1)
end
blockmap bm2(P::V)
  define distribute(block, *)
  define data(16384,50000)
  define grid(4096,1)
end
invoke $a[8192] ./comp1
invoke $b[4096] ./comp2
#set reduce fnvhash
set p2p true
graph $g
```

[10] We denote the general scenario for comparative assertions as p-2p to indicate that p processes from one program is compared with 2p processes from the other program.

```
  assert bm($a::array@"comp1.c":35)=bm($b::array@"comp2.c":35)
end
```

Figure 7-21: Comparative Assertion – Script for Testing General Scenario

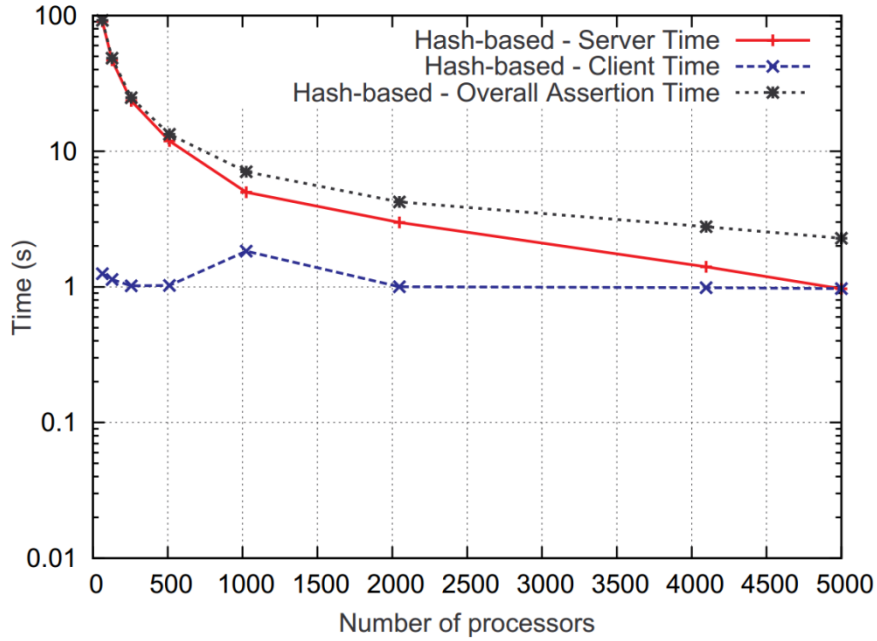


Figure 7-22: p-2p Strong Scaling Performance for the Hash-based Scheme

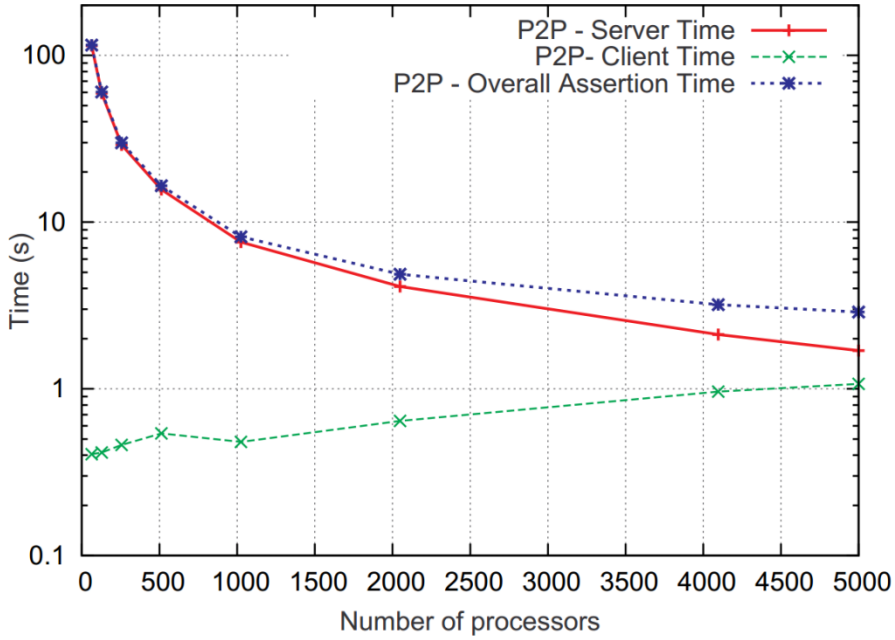


Figure 7-23: p-2p Strong Scaling Performance for the P2P Scheme

Figure 7-22 and Figure 7-23 present the strong scaling performance. Notably, the performance for P2P method shown here is lower than that of the experiment performed in Section 7.4.2 because every server has to communicate (serially) with two others. Overall, the assertion times for both the hash-based scheme and P2P scheme fall as the number of processors increases; this is as expected in a strong scaling experiment.

Importantly, because each debug server executes the P2P data-transfer task in a blocking manner, if a debug server X needs to transfer data to two other servers Y_1 and Y_2, Y_2 only starts receiving data after X has finished transferring data to Y_1. This is the current shortfall of the P2P technique. Under the general scenario (Figure 7-25), the hash-based scheme thus has the better performance. Even with this shortfall and using 5,000 servers, both techniques still complete the assertions in less than three seconds. As a result, we still see a speedup occurring from 2,000 servers for both solutions (Figure 7-24).

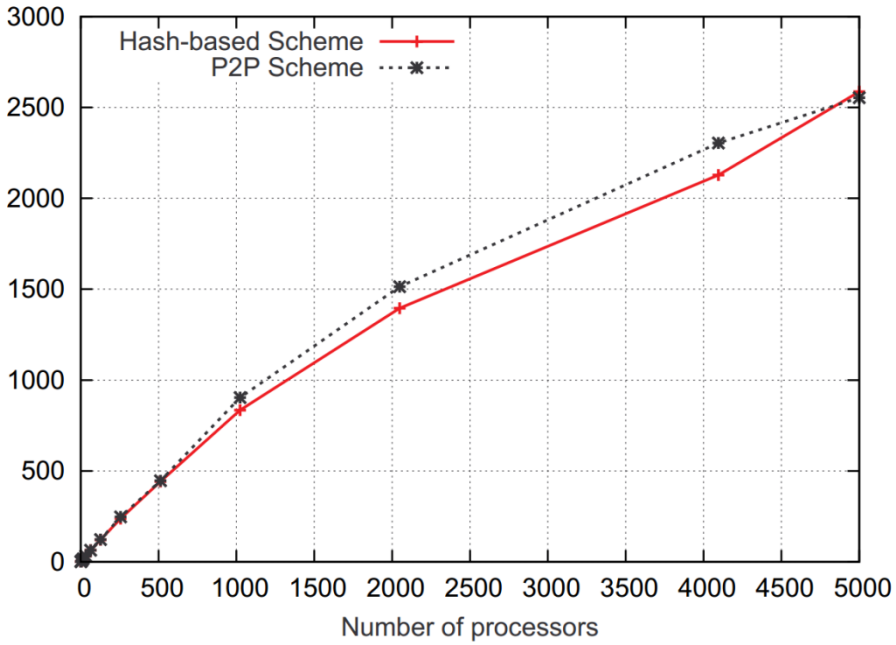


Figure 7-24: p-2p - Speedup Gain

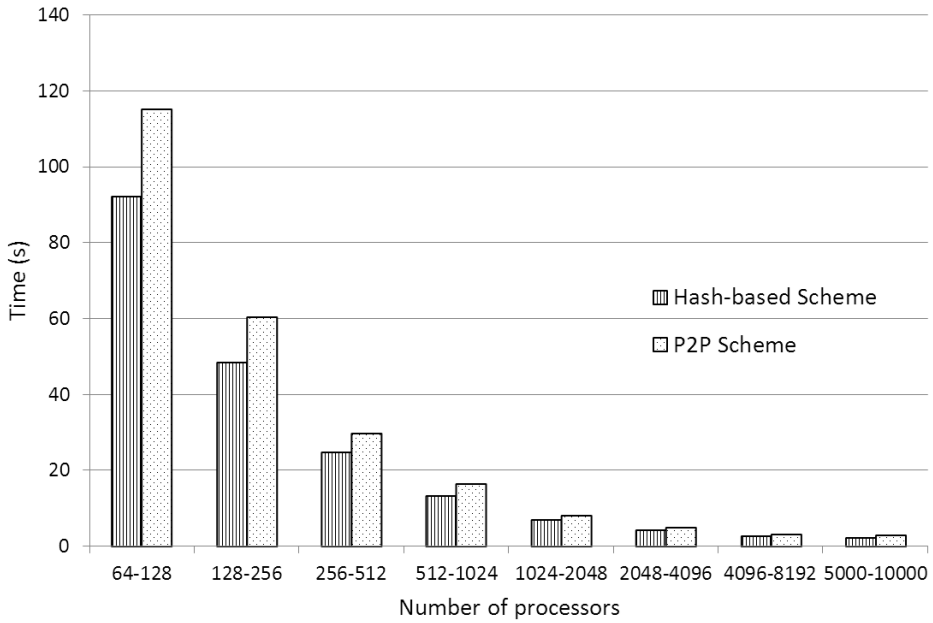


Figure 7-25: Overall Comparative Assertion Times under the General Scenario

7.5 Summary

This chapter delivered performance results for the different types of ad-hoc debug-time assertions discussed in this research. In order to assess the performance, both strong scaling and weak scaling tests were conducted on large data structures running on a supercomputer with around 20,000 compute cores. For some advanced assertions such as the histogram assertion, a bottleneck was identified where a single client process had to perform multiple tasks sequentially; this limited the speedup at a few thousand cores. These limitations were identified and can be targeted for future work (discussed in Chapter 8). Nevertheless, good speedups were achieved overall and the executions of our ad-hoc debug-time assertions returned acceptable response times. The results indicate that ad-hoc debug-time assertions are useful and practical debugging aids.

Chapter 8

Conclusion and Future Directions

This final chapter concludes the dissertation with a summary of achievements. Based on the experience gained throughout the course of this research, potential directions for future research into the assertion-based debugging domain are also identified and proposed.

8.1 Research Summary

The complexity in debugging programs that execute across very large parallel machines is a major problem in developing high-performance applications. Although progress has been made in the debugging domain (for instance, improvements in the user interface to better present runtime data), it is still cumbersome for developers to isolate the source of a software fault.

We see two main sources of challenge in the current approaches to debugging parallel applications. First, the state of a typical scientific application can be extremely large, making it infeasible for users to examine individual cells of, typically, large multi-dimensional floating-point structures. It is not feasible to print out complete arrays to decide if a program is computing the correct results. We referred to this as a cognitive challenge because programmers cannot integrate the data effectively into a mental model of correct execution. One potential solution for this challenge is to perform sophisticated data-processing operations in order to reduce large datasets into meaningful debugging information. However, such operations are often expensive to execute on a single processor. This leads to the second challenge in parallel debugging. Although debuggers are used to debug parallel programs, they generally do not make use of the underlying parallel platform to improve their own performance. Thus, debuggers are typically restricted to techniques that can be executed sequentially on a frontend node in a reasonable amount of time.

This research envisages a set of powerful debugging primitives that diminish the cognitive challenge. On a single processor, these would be too expensive to run sequentially. However, by using the inherent parallel-processing capabilities of a supercomputer (which is usually idle when a program is suspended at a breakpoint), a range of more powerful debugging operations become feasible. To enhance the debugging process, we employ assertions as the debugging directive.

Assertions, which have been available in programming languages for some time, allow a programmer to test assumptions regarding code behaviour. These constructs enable a programmer to insert a predicate at a particular location within a program. These predicates are then evaluated at runtime and an error is captured if any predicate is violated. With key attributes such as error detection, error isolation and error notification, assertions have been used extensively for evaluating invariants, checking

input parameters and enhancing program correctness and quality. However, the use of traditional assertions can be expensive. For example, the programmer must manually create assertions before the code is compiled. Programming assertions are, therefore, inflexible in that they can only be added and removed through the recompilation of the entire program. In addition, assertions can be computationally expensive because extra computation is required to validate them at runtime. As a result, assertions are more useful if they can be constructed outside the program source code and can be invoked solely at debug time to provide developers with the ability to detect any unexpected behaviour of the program.

The data-centric debugging method discussed and developed in this research couples assertions with existing debugging techniques to enhance the debugging process. The method provides a user with flexible assertions that can be altered to suit a dynamic environment. Ad-hoc debug-time assertions can be created and modified on-the-fly at runtime and they can be selectively enabled or disabled.

The debugging method presented in this research also provides powerful, yet flexible, templates that developers can use to create different sorts of assertions. For example, a user can specify conditions for correct data initialisation or data distribution between sets of processes. Users can also verify the correct result of a complex computation by making statements about the statistical attributes of a certain dataset. In addition, where a program is enhanced or ported to work on a larger-scale problem or to run on a more advanced computing platform, a user can construct assertions to verify the correct execution of the new code by comparing its runtime data with the data extracted from the working version. To generalise the usage of these debug-time assertions, this research categorised them into three templates: general ad-hoc assertions, statistical assertions and comparative assertions. Each of these assertion templates provides users with different mechanisms to assert various conditions throughout the execution of the target program.

In relation to implementation, supporting assertions of the types discussed in this research requires a novel architecture in which advanced data-reduction and data-comparison algorithms are integrated. The execution of a debug-time assertion can be typically divided into three phases: data collection, data processing and data reasoning. Conducting these phases becomes more complex when data is decomposed and distributed across many processing units. More importantly, we want to execute the assertions in parallel, making assertions over large data structures feasible. Hence, several

innovative components and techniques were introduced including the blockmap decomposition algebra, the hash-based and P2P-comparison techniques and the split-phase data-reduction mechanism.

Finally, this thesis also delivered a series of case studies that illustrated the expressive power of the assertion-based method in debugging highly parallel, data-intensive code. The case studies included a computational fluid-dynamics simulation using the Shallow Water Equations [177], a molecular-dynamics simulation [178] and a 2D photon-transport simulation using the Monte Carlo method to solve the Boltzmann transport equation. These programs were selected because debugging them with traditional debugging techniques is ineffective and expensive. These case studies highlight the efficiency of the assertion-based debugging method for automatically locating and notifying users of hard-to-spot errors. The performance was also evaluated to confirm the scalability of the technique.

8.2 Future Work

This section highlights future potential research directions within the assertion-based debugging domain.

8.2.1 Extending the Debugging Syntax

First, the current assertion syntax could be extended. The syntax currently uses a declarative notation with arrays as first-class objects. In addition, the syntax relies on the programmer making sensible choices about the location from which data will be extracted. However, a more powerful set of semantics would support assertion statements in debugging expressions of greater complexity and sophistication.

Second, the use of breakpoints in assertion statements limits the use of the technique in finding timing errors that are a common cause for failure in task-parallel programs such as the MPI programming paradigm. This leads to phenomena such as the probe effect as discussed in Chapter 2 [34, 72]. This is the situation where a parallel debugger masks out non-deterministic behaviours of a concurrent program while trying to synchronously control the parallel processes. Further investigation into the deployment of debug servers and the methods used to retrieve runtime data are required to address this issue.

8.2.2 Data Decomposition

As discussed in Section 4.3.1, the current mechanism to deal with the distribution of global data structures is limited to regular decomposition schemes. Future work could explore irregular data-decomposition techniques and devise a better data-mapping algorithm that works for arbitrary data-decomposition schemes.

8.2.3 Integration with Other Tools and Approaches

No single debugging technique can address all problems. Hence, we envisage the integration of the ideas presented in this thesis with other tools and approaches. To this end, we are leveraging the Eclipse Parallel Tools Plugin (PTP) [78] project with our current debugging primitives. PTP is a framework in which multiple tools can sit side-by-side in a powerful Integrated Development Environment (IDE). PTP supports the efficient invocation of parallel programs from the Eclipse environment. It can therefore help in improving the process to invoke debug servers. Having previously integrated Guard into Eclipse PTP, we feel this would be an effective framework for further experiments in debugging.

8.2.4 Expanding the Statistic-Reduction Framework

Our debugging framework provides a range of statistics to be used in constructing assertions when massive amount of data is produced by the program at runtime. However, the current implementation can be further enhanced to support more effective statistical operations.

First, fitting data onto a given function or model (linear or non linear) is an important procedure in any data-patterning algorithm. It is useful for users to identify the pattern generated by their data at debug time in order to spot any abnormal segment of data in a massive dataset.

Second, spotting outliers is a crucial part of debugging, regardless of the exact methodology used. However, when data structures become large and complex, finding values that stand out from the rest becomes increasingly difficult. Modern data-mining techniques allow users to quickly cluster data values and identify outliers.

Finally, the current implementation for statistical assertions provides a collection of basic statistical functions and allows users to code their own reduction routines. This open

implementation is useful because different scientific applications hold different statistical attributes. However, the current template is too simple to support more complex statistical routines, such as curve fitting or data clustering. In future, the current statistical framework could be equipped with more sophisticated data-mining algorithms. The split-phase reduction template would also be enhanced accordingly.

8.2.5 Performance Tuning

The debugger developed and evaluated in this thesis is a research prototype and serves as a proof-of-concept for the data-centric debugging methodology discussed in this thesis. While returning acceptable and promising performance, the implementation can be improved and the overall performance can be further tuned.

One area for improvement is the P2P API which uses simple socket communication to perform data transfer between backend debug servers. Further, the communication is conducted in a synchronous manner. In the future, more sophisticated communication techniques will be explored to remove this bottleneck at the backend when the P2P comparison mode is enabled.

Another area for improvement is the split-phase reduction operation. While the overall reduction can be partially parallelised using the parallel phase, the aggregation phase is executed sequentially by the frontend. This creates a bottleneck when the number of processes reaches millions or tens of millions. A potential solution is to perform the aggregation task across the communication layer. This indicates that a communication network tree such as MRNet can be used, not only for broadcasting debug commands and collecting runtime data, but also for aggregating the reduction results. This solution elevates the sequential-aggregation task currently performed by the frontend and thereby improves the overall performance of the technique.

8.2.6 Supporting GPUs and GPU Debugging

As briefly mentioned in Chapter 2, GPUs has become an increasing valuable resource in developing and executing high throughput, computationally intensive scientific applications. In addition, recent supercomputers contain both multi-core CPUs and GPGPUs (or GPUs). As future work, we aim to design techniques that allow our debugger to reason about data that is decomposed across the complex memory hierarchy of GPU based platform, develop new techniques that efficiently access distributed data,

and produce techniques that control applications running on both conventional multi-core CPUs and GPUs.

To sum up, the outcomes of this research suggest a novel debugging paradigm. The use of assertions as an integral part of a data-centric debugging method creates a much-needed debugging technique, and one that is relevant to highly parallel, data-intensive code. The results justify the continuation of research in this area to cement the technique and to provide support to researchers and developers working with high-performance computers.

References

[1] J. J. Dongarra and D. W. Walker, "The Quest for Petascale Computing", *IEEE Computing in Science and Engineering,* vol. 3, pp. 32-39, 2001.

[2] T. Dunning and R. Pennington, "Meeting the Challenge of Petascale Computing", National Center for Supercomputing Applications, Amsterdam2008.

[3] J. Vetter, "Software Development Tools for Petascale Computing", Oak Ridge National Laboratory and Georgia Tech, Washington, DC2007.

[4] NovoSpark Corporation. (2010, 10/04/2010). *NovoSpark Visualizer*. Available: http://www.novospark.com/

[5] paraview.org. (2012, 05/02/2012). *ParaView*. Available: www.paraview.org

[6] C. Viravan, "Enhancing Debugging Technology", Doctor of Philosophy, Purdue University, 1994.

[7] Free Software Foundation Inc. (2008, 15/01/2009). *GDB: The GNU Project Debugger*. Available: http://www.gnu.org/software/gdb/

[8] R. M. Stallman, *Debugging with GDB – The GNU Source Level Debugger*. Free Software Foundation, 2002.

[9] M. Auguston, C. Jeffery, and S. Underwood, "A Framework for Automatic Debugging", in *Automated Software Engineering*, 2002, pp. 217-222.

[10] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski, "Algorithmic debugging with assertions", in *Meta-programming in logic programming*, ed: MIT Press, 1989, pp. 501 - 521.

[11] G. Puebla, F. Bueno, and M. Hermenegildo, "A Framework for Assertion-based Debugging in Constraint Logic Programming", *Lecture Notes in Computer Science,* vol. 1520, p. 472, 1998.

[12] H. Lieberman, "The Debugging Scandal and What to Do About It", *Communications of the ACM,* vol. 40, 1997.

[13] K. Araki, Z. Furukawa, and J. Cheng, "A General Framework for Debugging", *IEEE Software,* vol. 8, pp. 14-20, 1991.

[14] A. Zeller, *Why programs fail: A guide to systematic debugging*. Elsevier, 2006.

[15] B.-d. Yoon and O. N. Garcia, "Cognitive Activities and Support in Debugging", in *Fourth Annual Symposium on Human Interaction with Complex Systems*, Dayton, USA, 1998, pp. 160 - 169.

[16] I. Vessey, "Expertise in debugging computer programs: A process analysis", *International Journal of Man-Machines Studies,* vol. 23, pp. 459–494, 1985.

[17] T. G. Moher and P. R. Wilson, "Guest Editors' Introduction: Offsetting Human Limits with Debugging Technology", *IEEE Software,* vol. 8, pp. 11-13, 1991.

[18] J. D. Gould, "Some psychological evidence on how people debug computer program", *International Journal of Man-Machine Studies,* vol. 7, pp. 151–170, 1975.

[19] Wikipedia. (2009, 25/03/2009). *debug (command)*. Available: http://en.wikipedia.org/wiki/Debug_(command)

[20] Wikipedia. (2009, 25/03/2004). *Turbo Debugger*. Available: http://en.wikipedia.org/wiki/Turbo_Debugger

[21] Free Software Foundation (FSF). (2008, 14/03/2009). *DDD - Data Display Debugger - GNU Project - Free Software Foundation (FSF)*. Available: http://www.gnu.org/software/ddd/

[22] Sun Microsystems, "Title", to appear in *Forte Developer 6 update 2*.

[23] B. Andersson, S. Blair-Chappell, and R. Mueller-Albrecht, "Intel® Debugger for Linux", Intel Corporation2010.

[24] H. Agrawal, "Towards Automatic Debugging of Computer Programs", Doctor of Philosophy, Software Engineering Research Center, Purdue University, 1991.

[25] G. R. Watson, "The Design and Implementation of a Parallel Relative Debugger", Doctor of Philosophy Dotoral, Faculty of Information Technology, Monash University, Melbourne, 2000.

[26] C. Fry, "Programming on an Already Full Brain", *Communications of the ACM,* vol. 40, 1997.

[27] Free Software Foundation Inc. (2012, 20/04/2011). *GCC, the GNU Compiler Collection*. Available: http://gcc.gnu.org/

[28] W. Teitelman and L. Masinter, "The InterLisp Programming Environment", *Computer Jounal,* vol. 14, 1981.

[29] eclipse.org. (2012). *Eclipse*. Available: http://www.eclipse.org/

[30] NetBeans.org. (2012). *NerBeans*. Available: http://www.netbeans.org/

[31] Microsoft. (2012). *.NET Framework Developer Center*. Available: http://msdn.microsoft.com/en-us/netframework/

[32] B. Lazzerini and L. Lopriore, "Abstraction mechanisms for event control in program debugging", *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 15, pp. 890-901, 1989.

[33] J. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, and D. Stemple, "The Ariadne Debugger: Scalable Application of Event-Based Abstraction", *Workshop on Parallel & Distributed Debugging,* pp. 85 - 95, 1993.

[34] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs", *ACM Computing Surveys,* vol. 21, pp. 593 - 622, 1989.

[35] H. Katsoff, "Sdb: a symbolic debugger", *Unix Programmer's Manual,* 1979.

[36] B. Beander, "Vax DEBUG: an interactive, symbolic, multilingual debugger", in *Proceedings of the symposium on High-level debugging*, 1983.

[37] CenterLine Software Inc., "CodeCenter User's Guide", CenterLine Software Inc., Cambridge, MA1992.

[38] T. A. Cargill, "The feel of Pi", in *UNIX*. vol. 2, ed Philadelphia, PA, USA: W. B. Saunders Company, 1990.

[39] R. M. Balzer, "EXDAMS: extendable debugging and monitoring system", in *Spring Joint Computer Conference*, 1969.

[40] H. Agrawal, R. A. DeMillo, and E. H. Spaord, "An Execution Backtracking Approach to Program Debugging", *IEEE Software,* pp. 21-26, 1991.

[41] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: a syntax-directed programming environment", *Communications of the ACM,* vol. 24, pp. 563-573, 1981.

[42] S. I. Feldman and C. B. Brown, "Igor: a system for program debugging via reversible execution", in *Proceedings of the Workshop on Parallel and Distributed Debugging*, Madison, WI, 1988.

[43] J. E. Archer Jr and R. Conway, "User recovery and reversal in interactive systems", *ACM Transactions on Programming Languages and Systems,* vol. 16, pp. 1-19, 1984.

[44] T. Shimomura and S. Isoda, "VIPS: a visual debugger for list structures", in *Proceedings of Computer Software and Applications Conference*, 1990, pp. 530-537.

[45] S. Isoda, T. Shimomura, and Y. Ono, "VIPS: A Visual Debugger", *IEEE Software,* vol. 4, pp. 8-19, 1987.

[46] T. G. Moher, "Provide: a process visualization and debugging environment", *IEEE Transactions on Software Engineering,* vol. 14, pp. 849-857, 1988.

[47] I. Foster, *Designing and Building Parallel Programs*. Addison Wesley, 1995.

[48] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[49] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture : a hardware/software approach*. Morgan Kaufmann Publishers, 1999.

[50] J. A. Fisher and B. R. Rau, "Instruction-level parallel processing", *Science,* vol. 253, pp. 1233-1241, 1991.

[51] G. S. Almasi and A. Gottlieb, *Highly parallel computing*. Benjamin/Cummings Pub. Co., 1994.

[52] H. Sutter and J. Larus, "Software and the Concurrency Revolution", *Queue - Multiprocessors,* vol. 3, 2005.

[53] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", in *AFIPS Conference Proceedings*, 1967, pp. 483–485.

[54] J. L. Hennessy, D. A. Patterson, and D. Goldberg, *Computer Architecture: A Quantitative Approach*, 2nd ed. San Francisco, USA. Morgan Kaufmann Publishers, 1996.

[55] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal of Research and Development,* vol. 11, p. 25, 1967.

[56] NVIDIA Corporation. (2012, 05/10/2012). *What is GPU Computing?* Available: http://www.nvidia.com/object/what-is-gpu-computing.html

[57] A. Lee. (2010, 05/10/2012). *GPU Parallelizable Methods*. Available: http://www.oxford-man.ox.ac.uk/gpuss/simd.html

[58] W. Stevens, *Unix network programming*, 2nd ed. Upper Saddle River, 1998.

[59] D. P. Helmbold and C. E. McDowell, "Race Detection -- Ten Years Later", in *Debugging and Performance Tuning for Parallel Computing Systems*, M. L. Simmons, A. H. Hayes, J. S. Brown, and D. A. Reed, Eds., ed Los Alamitos: IEEE Computer Society Press, 1996.

[60] A. Burns and G. Davies, *Concurrent Programming*. Addison-Wesley, 1993.

[61] K.-C. Tai, "Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs", in *International Conference on Parallel Processing (ICPP)*, North Carolina, USA, 1994, pp. 69-72.

[62] mpi-forum.org. (2012). *MPI: A Message-Passing Interface Standard*. Available: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[63] A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jiang, and V. Sunderam, *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA, USA. The MIT Press, 1994.

[64] H. Richardson, "High Performance Fortran: history, overview and current developments", Thinking Machines Corporation1996.

[65] The Open MPI Project. (2012). *Open MPI: Open Source High Performance Computing*. Available: http://www.open-mpi.org/

[66] Argonne National Laboratory. (2012). *MPICH-A Portable Implementation of MPI*. Available: http://www.mcs.anl.gov/research/projects/mpi/mpich1-old/

[67] Argonne National Laboratory. (2012). *MPICH2*. Available: http://www.mcs.anl.gov/research/projects/mpich2/

[68] lam-mpi.org. (2009). *LAM/MPI Parallel Computing*. Available: http://www.lam-mpi.org/

[69] OpenMP.org. (2011, 20/04/2012). *The OpenMP® API specification for parallel programming*. Available: http://openmp.org

[70] The Berkeley UPC Project. (2012). *Berkeley UPC - Unified Parallel C*. Available: http://upc.lbl.gov/

[71] Pacific Northwest National Laboratory. (2012). *Global Arrays Toolkit*. Available: http://www.emsl.pnl.gov/docs/global/

[72] J. Gait, "A debugger for concurrent programs", *SOFTWARE - PRACTICE AND EXPERIENCE,* vol. 15, pp. 539-554, 1985.

[73] Dolphin Interconnect Solutions Inc. (2007, 03/04/2009). *TotalView Debugger: A comprehensive debugging solution for demanding multi-core applications*. Available: http://www.totalviewtech.com/pdf/TotalViewDebug.pdf

[74] Allinea, "Allinea DDT – A revolution in debugging", ed, 2009.

[75] D. Cheng and R. Hood, "A Portable Debugger for Parallel and Distributed Programs", presented at the Conference on High Performance Networking and Computing, Washington, D.C., United States 1994.

[76] Compaq Computer Corporation. (2000, 06/04/2009). *Ladebug Debugger Manual*. Available: http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/V51_HTML/LADEBUG/TITLE.HTM

[77] C.-P. Chen, "The Parallel Debugging Architecture in the Intel Debugger", *Lecture Notes In Computer Science,* vol. 2763, pp. 444-451, 2003.

[78] eclipse.org. (2009). *PTP - Parallel Tools Platform*. Available: http://www.eclipse.org/ptp/

[79] IBM. (2010, 20/09/2010). *Chapter 10. Using the pdbx debugger*. Available: http://publib.boulder.ibm.com/infocenter/zos/v1r12/index.jsp?topic=%2Fcom.ibm.zos.r12.fomp100%2Fipezou00154.htm

[80] C.-C. Lin and R. J. LeBlanc, "Event-based Debugging of Object/Action Programs", *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on parallel and distributed debugging,* vol. 24, pp. 23 - 34, 1989.

[81] Sun Microsystems. (2003, 06/04/2009). *Prism™ 7.0 Software User's Guide*. Available: http://crc.nd.edu/information/prismug.pdf

[82] M. Oberhuber and R. Wism¨uller, "DETOP – An Interactive Debugger for PowerPC Based Multicomputers", in *Parallel Programming and Applications*, ed: IOS Press, 1995, pp. 170-183.

[83] D. Callahan and J. Sublok, "Static Analysis of Low-level Synchronization", *Workshop on Parallel & Distributed Debugging,* pp. 100 - 111, 1989.

[84] R. Hood, K. Kennedy, and J. Mellor-Crummey, "Parallel Program Debugging with On-the-fly Anomaly Detection", in *Supercomputing*, New York, USA, 1990, pp. 74-81.

[85] National Institute of Standard and Technology. (2006, 02/05/2012). *NIST DataPlot*. Available: http://www.itl.nist.gov/div898/software/dataplot/

[86] C. Zhang, S. Callaghan, T. Jordan, R. K. Kalia, A. Nakano, and P. Vashishta, "ParaViz: A Spatially Decomposed Parallel Visualization Algorithm Using Hierarchical Visibility Ordering", *International Journal of Computational Science,* vol. 1, pp. 407-421, 2007.

[87] ScienceGL Inc. (2012, 02/05/2012). *Advanced 3D Graphics for 3D, 4D and multiple dimension data sets. Visualization solutions for industry, business, healthcare and education*. Available: http://www.sciencegl.com/

[88] Kitware Inc. (2012, 02/05/2012). *VTK - The Visualization Toolkit*. Available: http://www.vtk.org/

[89] Andor Technology PLC. (2012, 02/09/2009). *Bitplane: Imaris Scientific 3D/4D Image Processing & Analysis Software*. Available: http://www.bitplane.com/

[90] C. Manning, "Traveler: The Apiary Observatory", in *European Conference on Object-Oriented Programming*, Paris, France, 1987.

[91] K. Paul, Harter Jr, D. Heimbigner, and R. King, "IDD: An Interactive Distributed Debugger", in *ICDCS*, Denver, Colorado, 1985, pp. 498-506.

[92] A. Hough and J. E. Cuny, "Belvedere: Prototype of a Pattern Oriented Debugger for Highly Parallel Computation", in *ICPP*, University Park, PA, 1987, pp. 735-738.

[93] L. Feng, T. Han, M. Kwiatkowska, and D. Parker, "Learning-based Compositional Verification for Synchronous Probabilistic Systems", *Lecture Notes In Computer Science,* vol. 6996, pp. 511-521, 2011.

[94] Z. Han, "Automatic Comparison of Execution Histories in the Debugging of Distributed Applications", Master of Mathematics in Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1999.

[95] F. Baiardi, N. D. Francesco, E. Matteoli, S. Stefanini, and G. Vaglini, "Development of a debugger for a concurrent language", *ACM SIGSOFT Software Engineering Notes,* vol. 8, pp. 98 - 106, 1983.

[96] R. E. Fairley, "ALADDIN: Assembly language assertion driven debugging interpreter", *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 5, pp. 426- 428, 1979.

[97] E. Shapiro, *Algorithmic Program Debugging*. Cambridge, Massachusetts, USA. MIT Press, 1983.

[98] W. L. Johnson, *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufmann, 1986.

[99] J. W. Lloyd, "Declarative error diagnosis", *New Generation Computing,* vol. 5, pp. 133-154, 1987.

[100] C. J. Fidge, "Partial orders for parallel debugging", *Workshop on Parallel & Distributed Debugging,* pp. 183 - 194, 1988.

[101] W. R. Murray, *Automatic Program Debugging for Intelligent Tutoring System*. San Francisco, CA, USA. Morgan Kaufmann, 1989.

[102] M. Ducassé, "A Pragmatic Survey of Automated Debugging", *Lecture Notes In Computer Science,* vol. 749, pp. 1 - 15, 1993.

[103] M. Auguston, "FORMAN - Program Formal Annotation Language", in *Fifth Israel Conference on Computer Systems and Software Engineering*, 1991.

[104] S. F. Siegel and T. K. Zirkel, "Collective Assertions", in *Verification, model checking, and abstract interpretation (VMCAI)*, Austin, Texas, 2011.

[105] C.-N. Wen, S.-H. Chou, T.-F. Chen, and T.-J. Lin, "RunAssert: A Non-Intrusive Run-Time Assertion for Parallel Programs Debugging", presented at the Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, 2010.

[106] D. Schwartz-Narbonne, F. Liu, T. Pondicherry, D. August, and S. Malik, "Parallel assertions for debugging parallel programs", presented at the Formal Methods and Models for Codesign (MEMOCODE), Cambridge, 2011.

[107] F. Gioachin and L. V. Kal´e, "Dynamic High-Level Scripting in Parallel Applications", presented at the International Symposium on Parallel&Distributed Processing, Roma, 2009.

[108] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "AccMon: Automatically Detecting Memoryrelated Bugs via Program Counterbased Invariants", *37th International Symposium on Microarchitecture (MICRO),* pp. 269-280, 2004.

[109] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution", *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 27, 2001.

[110] S. Hangal and M. S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection", in *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, 2002.

[111] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3S: Debugging Deployed Distributed Systems", presented at the Networked Systems Design and Implementation - NSDI, 2008.

[112] X. Liu, W. Lin, A. Pan, and Z. Zhang, "WiDS Checker: Combating Bugs in Distributed Systems", *NSDI,* 2007.

[113] Q. Gao, F. Qin, and D. K. Panda, "DMTracker: Finding Bugs in Large-Scale Parallel Programs by Detecting Anomaly in Data Movements", presented at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Reno-Tahoe, USA, 2007.

[114] G. Bronevetsky, I. Laguma, S. bagchi, B. R. d. Supinski, D. H. Ahn, and M. Schulz, "AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks", presented at the International Conference on Dependable Systems & Networks (DSN), Chicago, USA, 2010.

[115] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken, "Statistical Debugging of Sampled Programs", presented at the Neural Information Processing Systems (NIPS 2003), Vancouver, British Columbia, 2003.

[116] D. Abramson and R. Sosic, "Relative Debugging Using Multiple Program Versions", in *8 th International Symposium on Languages for Intensional Programming*, Sydney, 1995, pp. 3-5.

[117] D. Abramson and R. Sosic, "A Debugging and Testing Tool for Supporting Software Evolution", *Journal of Automated Software Engineering,* vol. 3, pp. 369-390, 1996.

[118] D. Abramson, I. Foster, J. Michalakes, and R. Sosic, "Relative debugging and its application to the development of large numerical models", presented at the Conference on High Performance Networking and Computing, San Diego, California, 1995.

[119] G. Watson and D. Abramson, "Relative Debugging for Data-Parallel Programs: A ZPL Case Study", *IEEE Concurrency,* vol. 8, pp. 42-52, 2000.

[120] G. Watson and D. Abramson, "Parallel Relative Debugging for Distributed Memory Applications: A Case Study", presented at the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, 2001.

[121] C. Gottbrath, "Automation Assisted Debugging on the Cray with TotalView", in *Cray User Group Proceedings*, 2011.

[122] F. Wang, Q. Zheng, H. An, and G. Chen, "A Parallel and Distributed Debugger Implemented with Java", in *Technology of Object-Oriented Languages and Systems*, Nanjing , China, 1999, pp. 342-346.

[123] J. C. Cunha, J. Lourenco, and T. Antao, "A Distributed Debugging Tool for a Parallel Software Engineering Environment", in *European Parallel Tools Meeting*, Onera, France, 1996.

[124] S. S. Lumetta and D. E. Culler. (1996, 06/04/2009). *Mantis User's Guide, Version 1.0*. Available: http://www.eecs.berkeley.edu/Research/Projects/CS/parallel/castle/mantis/mantis.tr.html

[125] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden, "A New Approach to Parallel Debugger Architecture", *Lecture Notes In Computer Science,* vol. 2367, pp. 139-149, 2002.

[126] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden, "Extending a traditional debugger to debug massively parallel applications", *Journal of Parallel and Distributed Computing,* vol. 64, pp. 617-628, 2004.

[127] Intel Corporation, "Title", to appear in.

[128] K. Lindekugel, A. DiGirolamo, and D. Stanzione, "A scalable Framework for Offline Parallel Debugging", presented at the Parallel Programming & Cluster Computing Workshop, Oklahoma, 2008.

[129] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, and R. Title, "A Scalable Debugger for Massively Parallel Message-Passing Programs", *IEEE Parallel & Distributed Technology: Systems & Technology,* vol. 2, pp. 50-56, 1994.

[130] M. N. Dinh, D. Abramson, D. Kurniawan, C. Jin, B. Moench, and L. DeRose, "Assertion based parallel debugging", in *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, Newport Beach, California, 2011, pp. 63-72.

[131] A. Turing, "On checking a large routine", in *Report of a conference on high speed automatic calculating machines*, ed: Cambridge, 1949, pp. 67-69.

[132] E. W. Dijkstra, *Structured programming*, 1969.

[133] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM,* vol. 12, pp. 576 - 580, 1969.

[134] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*, 2 ed. Boston. Kluwer Academic Publishers, 2005.

[135] P. Horan, "Eiffel Assertions and the External Structure of Classes and Objects", *Journal of Object Technology,* vol. 1, pp. 105-118, 2002.

[136] NUnit.org. (2002, 05/10/2012). *Nunit V2.0 - 3 October, 2002*. Available: http://www.nunit.org/NUnit-V2.0-ReadMe.pdf

[137] R. Sosic and D. Abramson, "Guard: A relative debugger", *Software Practice and Experience,* vol. 27, pp. 94-115, 1997.

[138] K. Thearling. (2009, 02/12/2009). *An Introduction to Data Mining*. Available: http://www.thearling.com/text/dmwhite/dmwhite.htm

[139] K. J. Cios, R. W. Swiniarski, W. Pedrycz, and L. A. Kurgan, *Data Mining: A knowledge Discovery Approach*. Springer, 2007.

[140] J. E. Gentle, *Random Number Generation and Monte Carlo Methods*, 2 ed. New York. Springer, 2003.

[141] D. Abramson, R. Sosic, and G. Watson, "Implementation Techniques for a Parallel Relative Debugger", presented at the International Conference on Parallel Architectures and Compilation Techniques, Boston, Massachusetts, USA, 1996.

[142] G. Watson and D. Abramson, "The Architecture of a Parallel Relative Debugger", presented at the 13th International Conference on Parallel and Distributed Computing Systems, 2000.

[143] D. Abramson, R. Finkel, D. Kurniawan, V. Kowalenko, and G. Watson, "Parallel Relative Debugging with Dynamic Data Structures", *Communications of the ACM,* vol. 39 pp. 69-77, 1996.

[144] G. Watson and D. Abramson, "Relative Debugging for Parallel Systems", *Proceedings of PCW 97,* pp. 25-26, 1997.

[145] D. Abramson, T. Ho, C. Chu, and W. Goscinski. (2008). *Eclipse Guard: Relative Debugging in the Eclipse Framework*. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.128.5814&rep=rep1&type=pdf

[146] D. Abramson and G. Edgan, "The RMIT Data Flow Computer: A Hybrid Architecture", *The Computer Journal,* vol. 33, pp. 230-240, 1990.

[147] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-based Multicast/Reduction Network for Scalable Tools", in *Proceedings of SC*, Phoenix, AZ, 2003.

[148] J. Chao, D. Abramson, M. N. Dinh, A. Gontarek, B. Moench, and L. Derose, "A Scalable Parallel Debugging Library with Pluggable Communication Protocols", presented at the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Ottawa, Canada, 2012.

[149] D. H. Ahn, B. R. d. Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, "Scalable Temporal Order Analysis for Large Scale Debugging", presented at the Supercomputing (SC), Portland, 2009.

[150] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. d. Supinski, M. Legendre, B. P. Miller, and B. Liblit, "Lessons learned at 208K: Towards Debugging Millions of Cores", presented at the ACM / IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, Austin, Texas 2008.

[151] A. N. A. D. Malony, A. Morris, D. C. Arnold, and B. P. Miller, "A Framework for Scalable, Parallel Performance Monitoring using TAU and MRNet", presented at the IEEE Cluster, Tsukuba, Japan, 2008.

[152] P. Pacheco, *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc, 1996.

[153] A. Petitet. (1995, 20/12/2009). *Block Cyclic Data Distribution*. Available: http://www.netlib.org/utk/papers/scalapack/node8.html

[154] NASA Advanced Supercomputing Division. (2009, 20/12/2009). *NAS Parallel Benchmarks*. Available: http://www.nas.nasa.gov/Resources/Software/npb.html

[155] ScaLAPACK Project. (2000, 21/12/2009). *The ScaLAPACK Project*. Available: http://www.netlib.org/scalapack/index.html

[156] R. H. Landau, Manuel José Páez, and C. C. Bordeianu, *Computational Physics Problem Solving with Computers*, 2nd ed. Weinheim. WILEY-VCH Verlag GmbH & Co KGaA, 2007.

[157] A. Nakano, R. K. Kalia, K.-i. Nomura, A. Sharma, P. Vashishta, F. Shimojo, A. C. T. v. Duin, W. A. Goddard, R. Biswas, and D. Srivastava, "A divide-and-conquer/cellular-decomposition framework for million-to-billion atom simulations of chemical reactions", *Computational Materials Science,* vol. 38, pp. 642–652, 2007.

[158] S. Ogata, T. J. Campbell, R. K. Kalia, A. Nakano, P. Vashishta, and S. Vemparala, "Scalable and portable implementation of the fast multipole method on parallel computers", *Computer Physics Communications,* vol. 135, pp. 445–461, 2003.

[159] J. Michalakes, J. Hacker, R. Loft, M. O.McCracken, A. Snavely, and N.-l. J. Wright, "WRF Nature Run", presented at the High Performance Networking and Computing, Nevada, 2007.

[160] T. F. Chan, G. H. Golub, and R. J. Leveque, "Updating Formulae and a Pairwise Algorithm for Computing Sample Variances", Department of Computer Science, Stanford University1979.

[161] T. F. Chan, G. H. Golub, and R. J. LeVeque, "Algorithms for computing the sample variance: analysis and recommendations", *The American Statistician,* vol. 37, pp. 242-247, 1983.

[162] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", presented at the Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004.

[163] B. Mulvey. (2006, 26/03/2009). *Hash Functions*. Available: http://bretm.home.comcast.net/~bretm/hash/

[164] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "Cryptographic Hash Functions: A Survey", 1995.

[165] P. Wolper and D. Leroy, "Reliable Hashing without Collision Detection", *Lecture Notes in Computer Science,* vol. 697, pp. 59-70, 1993.

[166] I. Mironov, "Hash Functions: Theory, attacks, and applications", Microsoft Research, Silicon Valley Campus2005.

[167] G. D. Knott, "Hashing Functions", *Computer Jounal,* vol. 18, pp. 265-278, 1972.

[168] M. Molina, S. Niccolini, and N. G. Duffield., "A Comparative Experimental Study of Hash Functions Applied to Packet Sampling", presented at the International Teletraffic Congress (ITC-19), 2005.

[169] A. Partow. (2009, 25/07/2009). *General Purpose Hash Function Algorithms*. Available: http://www.partow.net/programming/hashfunctions/

[170] A. Tridgell, "Efficient Algorithms for Sorting and Synchronization", Doctor of Philosophy, The Australian National University, Canberra, 1999.

[171] ETNUS. (2003, 29/10/2010). *Discovering TotalView*. Available: http://www.jaist.ac.jp/iscenter-new/mpc/old-machines/altix3700/opt/toolworks/totalview.6.3.0-1/doc/html/user_guide/intro.html

[172] Python Software Foundation. (2011, 18/07/2011). *Python Programming Language*. Available: http://www.python.org/

[173] Yale University. (1998). *Chi-Square Goodness of Fit Test*. Available: http://www.stat.yale.edu/Courses/1997-98/101/chigf.htm

[174] N. Gagunashvili, "CHI2 TEST FOR THE COMPARISON OF WEIGHTED AND UNWEIGHTED HISTOGRAMS", *Nuclear Instruments and Methods in Physics Research,* pp. 287-296, 2009.

[175] N. Gagunashvili, "Goodness of fit tests for weighted histograms", *Nuclear Instruments and Methods in Physics Research,* pp. 439-445, 2008.

[176] Free Software Foundation Inc. (2009, 25/11/2010). *The plotutils Package*. Available: http://www.gnu.org/software/plotutils/

[177] D. A. Randall, "The Shallow Water Equations", Colorado State University, Fort Collins2006.

[178] J. J. Nicolas, K. E. Gubbins, W. B. Streett, and D. J. Tildesley, "Equation of state for the Lennard-Jones fluid", *Molecular Physics,* vol. 37, pp. 1429-1454, 1979.

[179] ASC - Advanced Simulation and Computing. (2009, 16/10/2011). *ASC Sequoia Benchmark Codes*. Available: https://asc.llnl.gov/sequoia/benchmarks/#sphot

[180] D. Abramson, M. Dix, and P. Whiting, "A Study of the Shallow Water Equations on Various Parallel Architectures", presented at the 14th Australian Computer Science Conference, Sydney, 1991.

[181] CACS. (2011, 16/01/2011). *High Performance Computing and Simulations*. Available: http://cacs.usc.edu/education/cs653.html

[182] D. Frenkel and B. Smit, *Understanding Molecular Simulations: From Algorithms to Applications*, 2 ed. Elsevier Science & Technology, 2002.

[183] W. Wieser. (2008). *Simple molecular dynamics simulation of a Lennard-Jones fluid.* Available: http://www.triplespark.net/sim/ljfluid/

[184] E. Damiani, V. Liberali, and A. G. B. Tettamanzi, "Evolutionary Design of Hashing Function Circuits Using an FPGA", in *International Conference on Evolvable Systems: From Biology to Hardware*, 1998, pp. 36-46.

[185] C. Estebanez, J. C. Hernandez-Castro, A. Ribagorda, and P. Isasi, "Evolving hash functions by means of genetic programming", in *The 8th Annual Conference on Genetic and Evolutionary Computation*, Seattle, Washington, USA, 2006, pp. 1861-1862.

[186] L. C. Noll. (2010, 27/09/2009). *Fowler / Noll / Vo (FNV) Hash*. Available: http://isthe.com/chongo/tech/comp/fnv/

[187] X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions", *Advances in Cryptology – EUROCRYPT 2005,* vol. 3494, pp. 19-35, 2005.

[188] D. Jovanovic and P. Janicic, "Logical analysis of Hash Functions", *FroCoS : frontiers of combining systems,* vol. 3717, pp. 200-215, 2005.

Appendix A

Evaluation of Hash Functions

A.1 Motivation

Hashing has been used in many applications in computing. For example, in network transmission, hash functions can be used to map a large collection of messages into a small set of message digests in order to perform error detection, by appending the digest to the message during transmission. This research use hash functions to improve the performance of a debugging technique. Chapter 4 introduces a novel comparison scheme that uses a hash function to reduce amount of debug data so that comparison between two large datasets can be performed faster. However, while appealing in term of performance, reducing large data segments into small hash signatures leads to the potential of generating false results (that is hash collisions). As a result, choosing a good hash function becomes a critical task in implementing the hash-based comparison scheme as detailed in Chapter 4.

A.2 Experiment Configuration

Over the years, a number of good hash functions and hashing algorithms have been devised for various purposes [163, 165, 184-186]. Each hash function has a set of properties and designed goals. Some particular properties suit data comparison purposes and are listed here. In particular, for data comparison purpose, a hash function should:

- minimise the number of collisions. This is important because a high collision rate indicates that arrays have the same contents when they actually do not, and will confuse the debugging process.
- distribute signatures uniformly.
- have an *avalanche effect*. This attribute ensures that the output varies widely (e.g., half the bits changed) for a small change in input (e.g., changing a single bit). This is important because data may be skewed and only use a small part of the potential

range of input values. Having the avalanche effect will allow us to detect changes even when data is skewed in this way.

- minimise the size of the signature.
- detect permutations on data order within a structure. This is important because index permutation is a common error when code is parallelised and evolved.

Earlier work has studied these properties on a range of hash functions [163, 166, 187, 188], and has found that general hash functions such as the Bob Jenkins' hash function [164], FNV hash functions [186], the SHA hash family, or the MD hash family [164] exhibit the above properties [163, 164]. These hash functions are tested with specific hashing evaluation techniques, as discussed in [165-167].

In spite of these results, we concern that even a small collision probability might make the proposed hash-based comparison scheme ineffective. Accordingly, the probability of collisions in the AP hash function, FNV hash function and Bob Jenkins hash function are evaluated to confirm their suitability for our comparison scheme. Nevertheless, the following tests do not evaluate hash functions based on criteria such as the avalanche property, the uniform distribution property, and the correlation performance. They just aim to confirm that for various scenarios where certain types of debug data are collected, hashed and compared, the collision probability is low enough to make hashing a sensible choice for conducting relative debugging method.

To match real world data conditions, different criteria for selecting test data are applied. First, a range of different data types including character string, signed/unsigned integer and floating points are considered. Second, data elements are arranged in array with different patterns such as random, skewed (either increase or decrease incrementally in values), and permuted. Finally, the total amount of data used in the evaluation is around 46 Gbytes. Such data is decomposed into blocks of different sizes (ranging from 1 Kbyte up to 1 Mbyte). Each block is hashed individually to generate a 32 bits hash signature. The table below summaries the general configuration of the tests.

All test harnesses are programmed in C and runs on a GNU/Linux 64 bit system. We run several test cases, each with a different combination of data type, amount of data per signature, data generation method, and hash functions, as listed below. We compare arrays of hash signatures and record the number of collisions (if found) to determine the collision rate for specific scenario.

Table A-1: Configuration for Hash Function Evaluation

Data Types	Character (char) Unsigned Integer (unsigned int) Signed Integer (int) Single Precision Float(float) Double Precision Float(double)
Total Amount of Data	46 Gbytes
Hash Functions	Bob Jenkins Hash Function (BJHash) FNV Hash Functions (FNVHash) AP Hash Function (APHash)
Hash Signature Size	32 bits (unsigned int)
Amount of Data per Signature	1 Kbyte – 10 Kbytes – 100 Kbytes – 1024 Kbytes
Data Generation	Random /Shuffled / Permuted Skewed (increase/decrease incrementally in values)

A.3 Results

Results for different test cases are shown in the following tables. Each table presents test runs with a certain amount of data per signature (between 1 Kbyte and 1Mbyte) and records the collision rate for different combination of parameters.

A.3.1 Sub Result 1

Amount of Data per Signature: 1 Kbyte
Total Number of Signatures: 48,234,496

Table A-2: Collision Found for BJHash, FNVHash, APHash

Data Gen / Data Type	Random	Skewed	Shuffled	Permuted
Character	0	0	0	0
Unsigned Integer	0	0	0	0
Signed Integer	0	0	0	0
Single Float	0	0	0	0
Double Float	0	0	0	0

A.3.2 Sub Result 2

Amount of Data per Signature: 10 Kbytes

Total Number of Signatures: 4,823,450

Table A-3: Collision Found for BJHash, FNVHash, APHash

Data Type \ Data Gen	Random	Skewed	Shuffled	Permuted
Character	0	0	0	0
Unsigned Integer	0	0	0	0
Signed Integer	0	0	0	0
Single Float	0	0	0	0
Double Float	0	0	0	0

A.3.3 Sub Result 3

Amount of Data per Signature: 100 Kbytes

Total Number of Signatures: 482,345

Table A-4: Collision Found for BJHash, FNVHash, APHash

Data Type \ Data Gen	Random	Skewed	Shuffled	Permuted
Character	0	0	0	0
Unsigned Integer	0	0	0	0
Signed Integer	0	0	0	0
Single Float	0	0	0	0
Double Float	0	0	0	0

A.3.4 Sub Result 4

Amount of Data per Signature: 1024 Kbytes

Total Number of Signatures: 47,104

Table A-5: Collision Found for BJHash, FNVHash, APHash

Data Type \ Data Gen	Random	Skewed	Shuffled	Permuted

Character	0	0	0	0
Unsigned Integer	0	0	0	0
Signed Integer	0	0	0	0
Single Float	0	0	0	0
Double Float	0	0	0	0

A.4 Conclusion

In this section, we evaluate three hash functions including AP hash function (APHash), Bob Jenkins hash function (BJHash) and FNV hash function (FNVHash), on their suitability for our hash-based comparison scheme. The evaluation process uses a combination of different criteria including different types of data, different data block sizes, and different methods in generating large datasets (randomly, skewed data, permuted data, shuffled data). Importantly, the results of the tests show no collision at all, regardless of the data type for the ways data can be generated. These results confirm that using hash functions such as AP, FNV or Bob Jenkins to reduce runtime data for comparative debugging purposes is acceptable.

Appendix B

Performance Data

B.1 Description

Chapter 7 evaluates a number of debug-time assertions with different number of processes ranging from 64 to 20,000. Each experiment is repeated 10 times and the average values are used to graph the performance curves. In this section, average timing data from those experiments are presented.

B.2 Performance Data for Simple Assertions

B.2.1 Simple Compare Assertion

1. Strong Scaling Data

Cores	Server Time	Client Time	Overall Time
64	76.69857	0.580876	77.2796
128	34.60539	0.701065	35.30655
256	17.32625	0.701352	18.02769
512	8.768456	0.701747	9.470287
1024	4.271405	0.702636	4.974198
2048	3.450121	0.704754	4.158439
4096	2.141321	0.707882	2.849367
8192	1.185369	0.715575	1.901147
16384	0.63278	0.729084	1.362211
20000	0.433089	0.741006	1.246494

2. Weak Scaling Data

Cores	Server Time	Client Time	Overall Time
2	0.199931	0.700974	0.901054
4	0.201424	0.701018	0.902533

Cores	Server Time	Client Time	Overall Time
8	0.204154	0.701045	0.905316
16	0.205546	0.70097	0.906668
32	0.242685	0.700943	0.943719
64	0.244699	0.701051	0.945846
128	0.249174	0.70112	0.950387
256	0.24725	0.701377	0.948731
512	0.246055	0.701812	0.94797
1024	0.275982	0.702746	0.978824
2048	0.330935	0.704547	1.035669
4096	0.386355	0.707925	1.093344
8192	0.462485	0.714836	1.177538
16384	0.491512	0.729878	1.221678
20000	0.433089	0.741006	1.246494

B.2.1 Average Reduction Assertion

1. Strong Scaling Data

Cores	Server Time	Client Time	Overall Time
64	107.904	0.700952	108.6052
128	58.12959	0.560932	58.69077
256	29.85258	0.701249	30.55389
512	15.10485	0.701703	15.80661
1024	7.615765	0.702596	8.31842
2048	3.792202	0.704446	4.496775
4096	2.003297	0.708307	2.711667
8192	1.152574	0.715415	1.868058
16384	0.607215	0.728068	1.335841
20000	0.450006	0.730864	1.188727

2. Weak Scaling Data

Cores	Server Time	Client Time	Overall Time
2	0.198539	0.700872	0.899486
4	0.200521	0.70091	0.901571

Cores	Server Time	Client Time	Overall Time
8	0.201277	0.700915	0.902294
16	0.203297	0.700901	0.904277
32	0.369727	0.701089	1.070937
64	0.37003	0.700929	1.071009
128	0.390566	0.701071	1.091695
256	0.378488	0.701329	1.079883
512	0.38128	0.701419	1.082747
1024	0.380049	0.702654	1.082754
2048	0.426137	0.704467	1.130726
4096	0.435288	0.708276	1.14363
8192	0.449016	0.729639	1.179009
16384	0.450006	0.730864	1.18198
20000	0.473239	0.715295	1.188727

B.3 Performance Data for Statistical Assertions

B.3.1 User-defined Standard-deviation Assertion

1. Strong Scaling Data

Cores	Server Time	Client Time	Overall Time
64	112.8654	0.70103	113.5666
128	59.62044	0.701247	60.32185
256	29.68465	0.701655	30.38636
512	15.13144	0.702457	15.83395
1024	7.606061	0.704111	8.310236
2048	3.871351	0.707551	4.578966
4096	2.083324	0.713837	2.797225
8192	1.09692	0.727079	1.824069
16384	0.626931	0.754234	1.384023
20000	0.494993	0.75222	1.260173

2. Weak Scaling Data

Cores	Server Time	Client Time	Overall Time
2	0.198539	0.700872	0.899486
4	0.200521	0.70091	0.901571
8	0.201277	0.700915	0.902294
16	0.203297	0.700901	0.904277
32	0.369727	0.701089	1.070937
64	0.384089	0.70101	1.085145
128	0.375729	0.701219	1.077073
256	0.411431	0.701634	1.113108
512	0.401566	0.702396	1.10401
1024	0.389216	0.704075	1.093347
2048	0.432499	0.707172	1.139732
4096	0.442124	0.714021	1.158768
8192	0.457867	0.726889	1.184896
16384	0.489854	0.752763	1.242705
20000	0.494993	0.75222	1.260173

B.3.2 Built-in Standard-deviation Assertion

1. Strong Scaling Data

Cores	Server Time	Client Time	Overall Time
64	112.2959	0.622025	112.9181
128	59.68156	0.502182	60.18395
256	30.01947	0.702593	30.72214
512	15.03716	0.703358	15.74058
1024	7.547136	0.705083	8.252285
2048	3.943698	0.708295	4.652056
4096	2.047301	0.715342	2.762709
8192	1.123522	0.727653	1.851246
16384	0.940411	0.75463	1.69513
20000	0.877738	0.75281	1.636239

2. Weak Scaling Data

Cores	Server Time	Client Time	Overall Time
2	0.206457	0.702422	0.908961
4	0.514297	0.873354	1.387722
8	0.207579	0.702116	0.909783
16	0.207916	0.702014	0.910015
32	0.379142	0.702244	1.08148
64	0.378289	0.701918	1.080262
128	0.384649	0.70212	1.086831
256	0.384698	0.702501	1.087252
512	0.403352	0.703273	1.106741
1024	0.380944	0.704877	1.085876
2048	0.427519	0.708278	1.135855
4096	0.457947	0.71508	1.173263
8192	0.593338	0.728416	1.321885
16384	0.825751	0.754115	1.579951
20000	0.877738	0.75281	1.636239

B.3.3 Histogram Assertion

1. Strong Scaling Data

Cores	Server Time	Client Time	Overall Time
64	117.665	0.762278	118.4279
128	62.36676	0.801777	63.16924
256	31.39203	0.899573	32.29212
512	15.81369	1.04093	16.85505
1024	8.020563	1.359212	9.379942
2048	4.056895	1.562611	5.990642
4096	2.113958	1.817606	4.349991
8192	1.247792	2.187341	3.499302
16384	0.57499	2.329447	2.998678
20000	0.558246	2.392003	3.015986

2. Weak Scaling Data

Cores	Server Time	Client Time	Overall Time
2	0.230775	0.77724	1.008069
4	0.237806	0.723654	0.961869
8	0.293672	0.730223	1.024352
16	0.319976	0.731973	1.052561
32	0.397085	0.743072	1.140746
64	0.401033	0.763289	1.164723
128	0.439186	0.818064	1.257875
256	0.445993	0.893548	1.339833
512	0.435928	1.043918	1.480354
1024	0.422265	1.361124	1.784004
2048	0.469995	1.540764	2.055433
4096	0.507845	1.862222	2.389635
8192	0.525951	2.164217	2.704189
16384	0.57499	2.307907	2.952078
20000	0.558246	2.392003	3.015986

B.4 Performance Data for Comparative Assertions

B.4.1 Hash-based Comparison Technique

1. Strong Scaling Data – Best Case

Cores	Server Time	Client Time	Overall Time
64	85.6746718	1.3419232	88.393117
128	45.6143972	1.0733164	47.030933
256	22.6932732	0.9812358	24.490688
512	11.7876268	0.844069	13.116158
1024	5.8024686	0.8961062	7.0749262
2048	3.0451118	0.8534924	4.2643986
4096	1.4816572	0.8327598	2.6668472
8192	0.8795924	1.0011644	2.1836142
10000	0.766081	0.873452	2.06205

2. Strong Scaling Data – General Case

Cores	Server Time	Client Time	Overall Time
64-128	90.6355165	1.2514505	92.12767
128-256	46.8354578	1.1338306	48.412506
256-512	23.509113	1.0185772	24.76865
512-1024	11.8727618	1.0238772	13.337759
1024-2048	4.9868288	1.8346076	7.0631326
2048-4096	2.98213238	1.001566	4.226736
4096-8192	1.4039966	0.9843506	2.7706076
5000-10000	0.96763792	0.9712146	2.2795852

3. Weak Scaling Data – Best Case

Cores	Server Time	Client Time	Overall Time
2	0.30955	0.734639	1.511269
4	0.312488	0.820631	1.514105
8	0.355561	0.8946744	1.584494
16	0.389469	0.9286674	1.632259
32	0.436728	0.9646158	1.655936
64	0.522579	1.0267608	1.760092
128	0.551944	0.9612224	1.781111
256	0.51857	0.960479	1.716404
512	0.527203	0.9666662	1.749693
1024	0.54235	0.9646732	1.748823
2048	0.606454	0.9674822	1.815708
4096	0.633078	1.0066633	1.873525
8192	0.688187	1.008575	1.982873
10000	0.766081	1.035452	2.06205

B.4.2 P2P Comparison Technique

1. Strong Scaling Data – Best Case

Cores	Server Time	Client Time	Overall Time
64	119.068177	0.3804534	120.081927
128	61.1678724	0.4104896	61.9048382
256	31.5200932	0.427945	32.189358
512	15.9119684	0.492563	16.9578312
1024	7.8259244	0.5211068	8.5632728
2048	3.7986652	0.6066472	4.8165732
4096	1.548723	0.5470354	2.4997622
8192	1.20649528	0.6170072	2.0806194
10000	1.130886	0.6115656	1.9272332

2. Strong Scaling Data – General Case

Cores	Server Time	Client Time	Overall Time
64-128	114.308348	0.4055586	115.194341
128-256	59.385933	0.4154094	60.322139
256-512	29.0686042	0.460825	29.8497482
512-1024	15.6635958	0.5404002	16.5242786
1024-2048	7.5924024	0.4806136	8.1532838
2048-4096	4.1092116	0.641269	4.8710636
4096-8192	2.1174852	0.96095	3.19899
5000-10000	1.6940812	1.0720048	2.8866926

3. Weak Scaling Data – Best Case

Cores	Server Time	Client Time	Overall Time
2	0.496285	0.4203424	1.097216
4	0.4970662	0.4243256	1.0979788
8	0.5812606	0.4503174	1.1821478
16	0.6527444	0.4845442	1.2536922
32	0.789098	0.4821562	1.389986
64	0.8356962	0.5403312	1.4560634

Cores	Server Time	Client Time	Overall Time
128	0.8773382	0.480688	1.4784882
256	0.885538	0.5113828	1.4904758
512	0.860429	0.5972668	1.5120258
1024	0.9263168	0.5031452	1.5402216
2048	0.9904614	0.5298104	1.6551998
4096	1.0492712	0.5563952	1.7244596
8192	1.0477114	0.5994768	1.8436008
10000	1.130886	0.6115656	1.9272332

Appendix C

Survey of Parallel/Distributed Debuggers: 1969 - 2012

Name	Technology	Language	Authors	Date
	interactive, concurrent language	ECSP	De Francesco N., Latella D., Vaglini G., Baiardi F.	1983
defence	source-level interactive, concurrent	concurrent Euclid	Weber, JC.	1983
	event-driven multiprocess		Smith ET.	1984
CBUG	distributed, GUI	C	Gait J.	1985
dbxtool	GUI, multiple-process	C, Pascal, Fortran	Adams E., Muchnick SS.	1985
HARD		Ada	Di Maio A., Ceri S., Reghizzi SC.	1985
IDD	GUI, assertions, distributed	C, Modula2	Harter PK Jr., Heimbigner DM., King R.	1985
RADAR	event-based, replay		LeBlanc RJ., Robbins AD.	1985
TSL		Ada	Hembold D., Luckham D.	1985
YODA		Ada	LeDoux CH., Parker DS.	1985
DISDEB	interactive high-level, event-driven	Mara	Lazzerini B., Prete CA.	1986

Name	Technology	Language	Authors	Date
EBES	behaviour specification		Chien NH.	1986
Meglos		C	Gaglianello RD., Katseff HP.	1986
pdbx		C, Fortran, Pascal		1986
Pi	distributed, object-oriented	C, C++	Cargill TA.	1986
PTOOL			Randy Allen	1986
Belvedere	pattern oriented, animated	Simple Simon	Hough AA., Cuny JE.	1987
Bugnet	real time distributed debugging	C, Modula2	Jones SH., Barkan RH., Wittie LD.	1987
DI	interactive debugging interpreter	IF1, IF2/SISAL	Skedzielewski SK., Yates RK., Oldehoeft RR.	1987
Instant Replay			LeBlanc TJ., Mellor-Crummey JM.	1987
Jade			Joyce J., Lomow G., Slind K., Unger B.	1987
Pilgrim	Distributed	CLU	Cooper R.	1987
	windows, client/server	DADO	Mills RC., Woodbury L., Maguire GQ Jr.	1988
DECON	Concurrent	C, Fortran	Wei Min Pan., Jackson V.	1988
MacBug	GUI		Bemmerl T., Erl N., Hansen O.	1988
mtdbx	GUI, real-time views of multitasking synchronization primitives	Fortran	Griffin JH., Wasserman HJ., McGavran LP.	1988
ndb	Parallel, GUI, stack	CrOS NCUBE	Flower J., Williams	1988

Name	Technology	Language	Authors	Date
	frame		R.	
ParaScope	parallel programming environment	Fortran	Callahan CD., Cooper KD., Hood RT., Kennedy K., Torczon L.	1988
Parasight	high-level abstractions	C	Aral Z., Gertner I., Schaffer G.	1988
PPD	distributed breakpoints	C	MillerBP., Choi J.-D.	1988
Recap			Pan DZ., Linton MA.	1988
Voyeur	application-specific graphical views	Poker, Fortran	Bailey ML., Socha D., Notkin D.	1988
	Parallel		Griffin J., Hiromoto R.	1989
	data path debugging		Hseush W., Kaiser GE.	1989
	integrated tools	Fortran	Appelbe WF., McDowell CE.	1989
Agora	replay, user-defined synchronisation primitives	Agora	Forin A.	1989
Amoeba	Distributed	Amoeba	Elshoff IJP.	1989
DPD	Distributed	REM	Side RS., Shoja GC.	1989
HDB	Checksums		Cheng DY.	1989
MAD	debugs in parallel		Rubin RV., Rudolph L., Zernik D.	1989
Moviola	Visualisation, parallel		LeBlanc, Fowler, Mellor-Crummey	1989
Pdeb	shared memory parallel		Zifrony D., Averbuch A.	1989
	Distributed	Modula-2,	Scholten J., Jansen	1990

Name	Technology	Language	Authors	Date
		C/TUMULT	PG., Posthuma J.	
DB	Distributed		Dahem J-H, Lenga R.	1990
	sequential view		Cohn R.	1991
bdb	Library	Cray	Young B.	1991
CodeVision	distributed, client/server, multiple user interfaces	SGI	Chang AM., Karlton PL., Ciemiewicz DM.	1991
CPEM	graphical, multi-process	C	Bullinger H-J.	1991
DESK	distributed, object-oriented, heterogeneous	ESP	Khanna A.	1991
IPD	interactive, parallel		Intel Corporation	1991
ldb	Parallel	Fortran/UNICOS	Brown JS.	1991
mdb	debug library	Cedar	Emrath P., Marsolf B.	1991
MPD	event-action	ParaC	Ponamgi MK., Hseush W., Kaiser GE	1991
Observer	debugger for object-oriented, distributed programs		Jamrozik H., Roisin C., Santana M.	1991
Paragraph	post-mortem, visualization		Heath MT., Etheridge JA.	1991
Prism	Distributed		Thinking Machines	1991
SIMGER	language level simulator	EDAM	Chaumette S., Counilh MC.	1991
	parallel programming environment		Chaumette S., Counilh MC., Roman J., Vauquelin	1992

Name	Technology	Language	Authors	Date
			B., Charrier P.	
EREBUS	distributed	Estelle	Hurfin M., Plouzeau N., Raynal M.	1992
HyperDEBU	multiwindow, parallel	Fleng	Tanaka H., Tatemura J.	1992
NeD	debug server, programmable network interface		Maybee P.	1992
TOPSYS	parallel system tools	MMK	Bemmerl T.	1992
	layered distributed program debugger		Zhou W.	1993
	object, thread		Gunaseelan L., LeBlanc RJ Jr.	1993
	distributed		Scholten H., Posthuma J.	1993
Ariadne	event- and state-based debugging		Kundu J., Cuny JE.	1993
Conductor	simultaneous breakpoints		Baek Y., Jin S.	1993
Ddbx-LPP	distributed, GUI		Fernandez MG., Ghosh S.	1993
DPDP	distributed, event-action model	C	Zaki M., El-Nahas MY., Allam HA.	1993
IMPROV	debugging views, visualization		Kohl JA., Casavant T.	1993
Panorama	retargetable, extensible		May J., Berman F.	1993
PDG	process-level debugger for concurrent programs, animating, hierarchical	GRAPE	Caerts C., Lauwereins R., Peperstraete JA.	1993

Name	Technology	Language	Authors	Date
	graphical representations			
Source	distributed	ParMod	Weininger A.	1993
	integrated, hierarchical tool environment	TOPSYS	Bode A.	1994
	data-parallel, performance		Van Dongen V., Hurteau G., Singh A., Reiher E., Hum H.	1994
ADAT	automated, very high level	Ada	Lopes AV., Heller RS., Feldman MB.	1994
AIMS	instrumentation and monitoring		Yan JC.	1994
DETOP	simple GUI, static and dynamic parallel codes		Oberhuber M., Wismuller P.	1994
DPD	dynamic rollback, replay, GUI	REM	Side RS., Shoja GC.	1994
HP DDE	event-based, retargetable debugger	C, C++, Fortran, Pascal	Iyengar AK., Grzesik TS., Ho-Gibson VJ., Hoover TA., Vasta JR.	1994
LPdbx	distributed, iconic interface		Sorel PE., Fernandez MG., Ghosh S.	1994
mdb	semantic race detection, replay	C, Fortran/PVM	Damodaran-Kamal SK., Francioni JM.	1994
Node Prism	parallel message-passing, scalable expression, execution, and interpretation		Sistare S., Allen D., Bowker R., Jourdenais K., Simons J., Title R.	1994

Name	Technology	Language	Authors	Date
p2d2	parallel, client/server	HPF/MPI, PVM	Hood R., Cheng D.	1994
PPPE	integrated parallel programming tools	PARMACS/MPI	Cownie J., Dunlop A., Hellberg S. Hey AJG., Pritchard D.	1994
	parallel debugger with adaptively replayable lock		Miei T., Takahashi N.	1995
Annai	integrated tool environment	HPF/MPI	Clemencon C., Decker KM., Deshpande VR., Endo A., Fritscher J., Lorenzo PAR., Masuda N., Muller A., Ruhl R., Sawyer W., Wylie BJN., Zimmermann F.	1995
EDL	Event-Based Behavior Abstraction		Bates PC.	1995
GOLD	visualization-based environment		Sharnowski JL., Cheng BHC.	1995
GUARD	relative debugger, visualisation	C, Fotran, ZPL/MPI	Abramson D., Watson G., Sosic R.	1995
PDT	race detection, deterministic replay	Annai	Clemencon C., Fritscher J., Meehan MJ., Ruhl R.	1995
	parallel visualizing debugger		Oyanagi S., Kubota K., Kawakura Y.	1996
	parallel debugger	Parallaxis	Braunl T., Keller H., Stippa J.	1996
BUSTER	integrated parallel debugger	PVM	Jianxin X., Dingxing W., Weimin Z.,	1996

Name	Technology	Language	Authors	Date
			Meiming S.	
dbxR	replay	PVM	Miei T., Takahashi N.	1996
DDB	distributed, replay		Sienkiewicz J., Radhakrishnan T.	1996
Mantis	GUI, interactive, visualisation engine	C, C++, Fortran, Split-C	Steven S. Lumetta and David E, Culler	1996
	replay debugger	A-NETL	Baba T., Furuya Y., Yoshinaga T.	1997
	multilingual distributed debugger		Olivier PA.	1997
	data parallel, run-time dependence analysis, performance		Rajamony R., Cox AL.	1997
Aardvark	single control and data views	HPF	LaFrance-Linden DCP.	1997
deet	machine-independent, graphical, programmable, distributed, extensible, and small	C, Java	Hanson DR., Kern JL.	1997
EPPP	data parallel performance debugger		Singh A., Van Dongen V.	1997
Kemari	programming environment	HPF/MPI	Kamachi T., Muller A., Ruhl R., Seo Y., Suehiro K., Tamura M.	1997
	process tracing,	CHILL	Paik EH., Byun YJ.,	1998

Name	Technology	Language	Authors	Date
	break-point setting, process monitoring, error localization, and error fixing		Chung YS., Lee BS.	
DDBG	interfacing to software engineering environment, graphical programming language a testing and debugging tool, meta-breakpoint, macrostep execution	C, GRAPNEL /PVM	Cunha JC., Lourenco J., Antao TR., Kacsuk P.	1998
net-dbx	java-based debugger	MPI	Neophytou N., Evripidou P.	1998
ParaDebug	graphical view mapping	ParaC/MPI	Gi-Won O., Dong-Hae C., Suk-Han Y.	1998
PDBG	process-based distributed debugger	DAMS	Cunha JC., Medeiros P., Lourenco J., Duarte V., Vieira J., Moscao B., Pereira D., Vaz R.	1998
PSUITE	graphical array-data visualizer		Fujii H., Shibata T., Yoshioka H., Ishikawa K., Endo A., Nakatomi T.	1998
UniVIEW	event trace based, heterogeneous, client/server	RPC	Young-Ae S., Eun-Jung L., Chang-Soon Pk.	1998
Xunify	instrumentation system and a		Lumpp JE Jr., Sivakumar K., Diaz	1998

Name	Technology	Language	Authors	Date
	performance evaluation tool		C., Griffioen JN.	
	reduced intrusion, cooperative debugging	CHILL, C, C++	Sato N., Wanvik DH., Botnevik H., Borsting T., Stromme JE.	1999
DCDB	java front end		Feng W., Qilong Z., Hong A., Guoliang C.	1999
DeHiFo	HPF debugger	HPF	Brezany P., Grabner S., Sowa K., Wismuller R.	1999
MPVisualizer	trace/replay mechanism, GUI, visualization engine		Claudio AP., Cunha JD., Carmo MB.	1999
Umpire	profiling	MPI	Jeffrey S. Vetter, Bronis R. de Supinski	2000
LadeBug	interactive, command-line, tree-based network	C, C++, Fortran, COBOL, and Ada	Susanne M. Balle, Bevin R. Brett, Chih-Ping Chen, David LaFrance-Linden	2002
CDB	GUI, interactive	PVM, MPI, Java	Xingfu Wu, Qingping Chen, Xian-He Sun	2002
idb	GUI, tree-based network	C, C++, Fortran, Pthreads, OpenMP, MPI	Intel Corporation	2003
	data-replay, reverse execution, checkpointing			2005
PDB	event-based, interactive		Rashid Mehmood, Jon Crowcroft,	2005

Name	Technology	Language	Authors	Date
			Steven Hand and Steven Smith	
VDT	interactive, visualisation		Lami Kaya Mohammad Al-Qudah	2006
XMPI	GUI, visualisation	LAM/MPI	Indiana University	2006
Thread Checker	Data race-detection, thread		Intel	2006
STAT	stack trace analysis and visualisation, MRNet layer	MPI, C, C++, Fortran	D. C Arnold, D. H Ahn, B. R Supinski, G. Lee, B. P Miller, and M. Schulz	2007
DPS (Dynamic Parallel Schedules)	flow graph based, message race detection		Ali Al-Shabibi, Sebastian Gerlach, Roger D. Hersch, Basile Schaeli	2007
GDBase	offline event analysis, post-mortem	MPI, C, C++, Fortran	Karl Lindekugel, Anthony DiGirolamo, and Dan Stanzione	2008
Marmot	profiling	Hybrid MPI-OpenMP	Tobias Hilbrich, Matthias S. Müller, Bettina Krammer	2009
padb	profiling commnadline	MPI	Ashley Pittman	2010
CharmDebug	record-replay, interactive	Charm++	University of Illinois	2010
AutomaDeD	automated debugging, offline analysis, statistical verification	MPI, C, C++, Fortran	G. Bronevetsky, I. Laguma, S. bagchi, B. R. d. Supinski, D. H. Ahn, and M. Schulz	2010

Name	Technology	Language	Authors	Date
DDT	GUI, visualisation engine, memory debugging	MPI, OpenMP, C, C++, Fortran	Allinea Inc.	2010
TotalView	GUI, visualization engine, memory debugging	MPI, OpenMP, C, C++, Fortran	Rogue Wave Software	2010
Eclipse PTP	IDE, interactive, visualisation	Fortran , C/C++, Java, MPI, OpenMP and UPC	Eclipse Foundation	2010
Visual Studio 2010	IDE, interactive, visualisation, task based parallel debugging	Visual C++, C#, Visual Basic, .NET Platform	Microsoft	2010
pgdbg	GUI, thread, stack trace, visualisation	Fortran, C/C++, MPI, OpenMP, hybrid MPI-OpenMP	The Portland Group	2010