THE DESIGN AND IMPLEMENTATION OF A PARALLEL RELATIVE DEBUGGER

by

Gregory R Watson

Bachelor of Computer Science (Honours)

A thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy

Monash University

October 2000

This thesis was produced with Microsoft® Word 2000[1]. The main text was typeset using Garamond 12 point font and 1.5-line spacing. Program code and commands use the `Courier New` font. Diagrams were produced with the Microsoft Word Picture Editor and equations with Microsoft Equation 3.0. The production copies were printed on an HP DeskJet 1120C printer.

[1] I don't claim it was easy.

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.

Gregory R Watson

ACKNOWLEDGMENTS

The author wishes to acknowledge the support of a number of people, without whom this thesis would not exist. First and foremost is the wonderful Libby Chaplin, who has been my constant companion and soul mate throughout this entire process and beyond, and who's support made this thesis possible. Next is my supervisor, Prof. David Abramson, for the remarkable effort that he has made to provide all the resources and encouragement that any PhD candidate could possibly hope for. I would also like to thank my family for their enthusiasm and understanding when, as is always the case, things don't quite go to plan. Thanks too, go to my very best friends Abigail and Su who, as partners in crime, understand what it is all about. Finally, the time I have spent walking with Jarrah, during those crisp winter mornings and balmy summer evenings, has enabled me to keep a clear head and a constant focus when I needed it most.

This thesis is dedicated to the memory of US Navy Rear Admiral Grace Brewster Hopper (9 Jan 1906 -1 Jan 1992) who is credited with having coined the term "debug", and the adage "it is always easier to ask forgiveness than it is to get permission", which has been the guiding principle in system administrator decisions ever since.

ABSTRACT

As the complexity of critical software systems increases, there is a correspondingly greater risk of software errors occurring. The trend towards global reliance on critical and complex technologies that has been apparent for many years means that the risk of serious economic or environmental damage due to software failures is also increasing. Such issues must now be considered an important factor in the life cycle of these critical systems. Program debugging is a key component of this software life cycle. While a large amount of research has been undertaken into developing debugging techniques to assist the software development process, little, if any, work has been directed specifically at the software evolution process.

This thesis presents a novel and interesting technique, known as *relative debugging*, which exploits the ability to utilise information from previous program releases as an aid in the debugging process. Relative debugging is a powerful paradigm that enables a programmer to locate and identify errors by comparing key data structures in a suspect program against that in a reference program. By observing the divergence of the data as the programs are executing, the programmer is able to make informed decisions as to the likely cause of the errors. Relative debugging is effective in these situations because the user can concentrate on *where* two related codes are producing different results, rather than being concerned with the actual values. The efficiency of the technique has been well established, and various case studies reporting the results of using relative debugging have been published.

The development of the latest version of the relative debugger, known as GUARD-2000, has required a number of significant advances in the field of debugger research. This includes a special internal data format that is independent of any particular architecture, the use of dataflow technology to control the comparison of data from executing programs and the ability to make comparisons in situations where significant transformations to data structures have occurred. This technology is built on top of a client/server architecture that supports a wide range of parallel and sequential architectures, and includes the ability to choose alternate network protocols for communication between the client and servers, and to deploy different low-level debug engines on the servers.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

On July 22, 1962 an Atlas-Agena booster was launched carrying the first U.S. Venus probe *Mariner I*. At a height of approximately 90 miles, ground controllers realised in horror that the rocket flight had become unstable and was heading off-course. The multi-million dollar rocket had to be blown up rather than risk a crash into a populated area. Extensive analysis of the incident eventually uncovered the problem. The flight plan software in a ground-based computer system monitoring the launch was missing a hyphen in a critical guidance equation.

Twenty-eight years later, on January 15, 1990 new software installed in 114 of AT&T's electronic telephone switching systems failed, allowing the switches to repeatedly propagate status messages and causing the switches to continually reset. Frustrated engineers were only able to correct the problem by drastically reducing the total message load on the network. The result was a 9-hour nationwide blackout, reportedly blocking 5 million calls and affecting 10 million customers. The problem was eventually traced to a misplaced `break` statement in a C program [54].

These two incidents, nearly three decades apart, represent only a tiny fraction of such failures; yet serve to illustrate the vulnerability of critical systems to software errors. The reason that these critical systems are so vulnerable is because such systems have critical requirements, and failure to meet these requirements can result in catastrophe. In addition, as the complexity of these systems increases, so does the complexity of the software needed to control them, and consequently there is a greater risk of software errors occurring. The trend towards global reliance on critical and complex technologies has been apparent for many years. Risk of serious economic or environmental damage due to such failures must now be considered an important factor in the life cycle of these pervasive technologies.

Myers [52] views software as a collection of information. From this perspective, software development can be regarded as a problem-solving process that involves the translation of a complex problem into its solution. This process involves translating the initial problem, through a number of intermediate steps that provide increasingly detailed representations, into in a large number of instructions that direct a computer in how to solve the problem.

Errors are introduced into the software whenever an intermediate translation does not accurately represent the problem, and so there are ample opportunities for this to occur in the development process. Once errors have been introduced, it is no longer possible to guarantee that the requirements of the system can be met. The software is unreliable.

As a consequence, considerable effort has been devoted to the task of increasing the reliability of software. Myers considers the software life cycle as a two-phase process: *design* and *testing*. The *design phase* comprises the usual processes of requirements analysis, specification, design and implementation. The main techniques for improving reliability through this design phase fall into the following categories:

- *Fault avoidance.* This is the process of reducing or eliminating the occurrence of errors in software by minimising complexity, improving communication, and identifying and removing translation errors.

- *Fault detection.* Designing software that provides a fault detection capability ensures that errors can be detected and reported as soon as they occur, so that the effect of the errors can be minimised.

- *Fault correction.* One step further is designing software with the capacity to correct the errors once they are detected, or to repair the damage that the errors may have caused.

- *Fault tolerance.* This technique refers to software that has been designed with the ability to continue to function even if errors occur.

Another major influence on the reliability of software is the *testing phase*, which involves the process of executing a program with the intention of finding errors. The testing process usually consists of designing a set of test cases according to some established objectives, executing the test cases using the software being examined and analysing the results to identify any errors. Errors that are discovered during the testing phase must then be located and corrected using a process known as *debugging*.

It is generally accepted, but not discussed in detail by Myers, that there is a third phase in addition to design and testing in the software life cycle: *maintenance* [38]. Even the most carefully designed and implemented software will contain errors, and as the faults caused by

these errors are observed, corrections to the code must be made. This is usually accomplished through incremental changes that are implemented via new releases of the software. In addition, new releases may also be made to cater for changes or extensions to user requirements, incomplete specifications, errors in the design or implementation, changes to hardware, or a combination of these. Changes such as these, made through an ongoing maintenance process, result in continuous program *growth*. Lehman first identified that program growth was related to the improvement in functional capability in 1969 [39]. It is now widely accepted that continuous growth through the initial development and ongoing maintenance of software constitutes *program evolution*, and is intrinsic to the nature of the software development process. According to Lehman, as much as 70% of the lifetime cost of software is expended after the software is first installed.

Program debugging is a key component of the software life cycle and is utilised in each phase of the process:

- *Design* – during the implementation stage of the design process, code must be validated, and as faults are identified they must be located and corrected.

- *Testing* – the testing process is designed to uncover errors that have been introduced during the design phase, but still requires a significant effort to analyse, locate and correct the errors in the code.

- *Maintenance* – as the program evolves either through enhancements to the design, as a response to the incidence of errors, or because of other environmental factors, debugging will need to be employed to ensure that the resulting software remains fault-free.

Traditional debugging techniques have been designed to aid software development by providing the tools necessary to identify, locate and correct errors in a program. To successfully debug a program, it is first necessary to gather enough information to identify an error. There are a number of methods available to do this, such as examining memory dumps, displaying the contents of memory or registers, tracing program state, or viewing event history data. This information gathering is used to establish the nature of the error, what data structures are affected, what modules or subsystems are involved, and to provide assistance in locating the source of the error. Using this information, the next step is to determine the exact cause and location of the error. In an interactive debugger, this process

might involve setting breakpoints at various locations throughout the program and observing program state at these points. In some cases it may be possible to isolate an error by altering data on the fly. Static analysis can also be used as an aid to locating and classifying errors by performing data and control flow analysis of the static program code. Once the source of the error has been identified, it is then necessary to correct the problem. This is done by proposing a solution and examining the impact that this will have on the program. The necessary modifications are then made, and the appropriate tests undertaken to ensure that error is actually fixed, and that no new errors have been introduced.

While a large amount of research has been undertaken into developing debugging techniques to assist the software development process, little, if any, work has been directed specifically at the software evolution process. In particular, the ability to utilise information from previous program releases to aid in the debugging process seems to have been largely overlooked. In 1994, Abramson and Sosič invented a technique known as *relative debugging* in response to a large effort to translate existing sequential programs to the new parallel architectures that were becoming available. Subsequently, it was realised that relative debugging applies equally well to more general program evolution. The relative debugging model they developed relies on the premise that, in many situations where software evolution takes place, key data structures are often invariant across releases. Because of this, it is possible to utilise the information content contained in prior, but correctly functioning releases to support the process of debugging a new version of the software.

Relative debugging is a powerful paradigm that enables a programmer to locate and identify errors by comparing a suspect program against a reference code. By observing the divergence of key data structures as the programs are executing, the programmer is able to make informed decisions as to the likely cause. This becomes a particularly valuable tool when a program is rewritten in order to operate on another computer platform (a technique known as *porting*). Relative debugging is effective in these situations 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. The efficiency of the technique is now well established, and various case studies reporting the results of using relative debugging have been published [5,2,3,69].

The concept of relative debugging is both language and machine independent. A programmer can compare data structures without concern for the implementation, and thus

attention can be focussed on the cause of the errors rather than implementation details. As a result, the effective implementation of a relative debugger requires architecture features that differ from traditional debuggers. A significant requirement is the need to employ a client/server model because the system must support the concurrent execution and control of multiple processes, potentially on different machines. In addition to this, the debugger client must operate in a multi-threaded manner in order to receive and process data from a number of independently executing programs. Another significant requirement is due to the likelihood that the programs might be running on completely different architectures. This means that the debugger must provide some mechanism to isolate architectural characteristics like word size, address type and byte ordering to enable the comparisons to be made. Finally, the debugger must also be able to deal with situations where data decomposition or transformation techniques have been applied in the process of porting code from one architecture to another. During the porting process it might be necessary to change the underlying data structures to better suite the target language or machine architecture. For example, one version of a program suited to a vector processor might require long single dimensioned arrays, whereas a cache based multiprocessor version might require two dimensioned arrays for efficient execution. This means that the relative debugger must be able to reverse these transformations seamlessly in order that data structures can be compared

While Abramson and Sosič's original implementation of the relative debugging technique addressed a number of these issues, it also had some severe limitations. The client/server debug engine was based on Sosič's earlier "dynascope" work [63]. While this was designed to provide rudimentary architecture independence, like most debuggers it was also highly machine specific and so supported only a very small number of architectures. More importantly, the comparison algorithm employed was simplistic and only single threaded, so was seriously limited in its ability to deal with the asynchronous matching behaviour apparent when debugging parallel programs. In fact, while the original implementation was designed to aid in debugging parallel codes, the best it could deal with was a parallel program running on a single process. The other major problem with the implementation was that it provided no mechanism to deal with data structures that had been decomposed or transformed in the porting process, something that is normally very common. This limited its applicability to situations where only very simple transformations had been applied, such as swapping array indices.

The latest version of the GUARD debugger, known as GUARD-2000 [69a], addresses each of the limitations of the original implementation and meets all the requirements for a successful relative debugger. The development of GUARD-2000 has required a number of significant advances in the field of debugger research, and which form the basis of this thesis. These advances consist of a number of key architectural features that have been added to the debugger, including the following:

- A special data format that is independent of any particular architecture has been developed. An architecture independent format (AIF) library provides a set of routines that can be used to store and manipulate data in a completely machine independent fashion.

- The debugger utilises a client/server architecture that includes a switchable protocol interface and pluggable low-level debug engines. By supporting a range of protocols, the debugger is able to take advantage of protocol features that suit a particular environment. Low-level debug support is also the main factor that limits a debugger's portability. Allowing different debug engines to be deployed ensures that the debugger can support a wide variety of parallel and sequential architectures.

- A dataflow engine has been introduced to automate the control and comparison of data from the programs being debugged. Dataflow is particularly suited to this environment for a number of reasons. The execution of a dataflow engine occurs in an inherently multi-threaded manner, which addresses the requirement outlined above. In addition, the implicit matching logic of the dataflow model provides an ideal framework to support the asynchronous extraction of data from concurrently executing programs.

- The ability to transform data structures between different versions of a program is another novel innovation. A transformation algebra has been developed in order to address situations where a data decomposition or other transformations have been applied. Once the actual transformation has been described using this algebra, the debugger is then able to perform a seamless comparison of data.

- The ability to manage data decomposition and transformations enables the debugger to provide support for distributed and shared memory parallel architectures, as well as a wide range of data parallel languages.

The realisation of the relative debugging technology has been a complex process involving a lengthy development cycle. Yet the current implementation of the debugger combines all these requirements together into an impressive package using a layered and modular architecture. Novel features have been employed, such as the ability to choose alternate network protocols for communication between the client and servers, and to deploy different low-level debug engines on the servers. GUARD-2000 is now placed as the first fully implemented prototype of the relative debugging paradigm.

The remainder of this document is devoted to presenting the key contributions the author has made to the technique of relative debugging. A brief overview and guide to reading this document follows.

Chapter 2 provides a brief history of debugging and introduces the reader to some of the main debugging techniques that have been developed over this time. Further, it shows where relative debugging and in particular the GUARD debugger fits into this evolution. Finally, the chapter focuses on the architecture of conventional debuggers and discusses why these are not suitable for implementing relative debugging.

Chapter 3 provides a detailed introduction and analysis of the relative debugging technique. The chapter then examines the implications and challenges that have had to be overcome to extended the paradigm to parallel computers.

Chapter 4 describes the key architecture features of a parallel relative debugger, and describes how an innovative design has been used to fully realise the relative debugging technology.

Chapter 5 presents a formalisation of the techniques that are used to describe the mapping and transformation operations that are employed by developers when codes are ported to parallel machines. This unique and powerful formalism underpins the data transformation features that are provided by the debugger.

Chapter 6 provides details of the novel techniques used to implement the key architectural features of the debugger, and identifies the challenges that have had to be overcome to during this process.

Chapter 7 describes three case studies of programs debugged using GUARD. These case studies illustrate the power of relative debugging, and how our implementation can be applied to all three major parallel software paradigms: data parallel, shared memory and distributed memory. In addition these case studies have allowed us to test the debugger on a variety of hardware and operating system platforms, thus illustrating the portability of the debugger itself.

Chapter 8 provides an analysis of some future directions for improvements to the debugger and for research into relative debugging technology.

Chapter 2

EXISTING DEBUGGING TECHONOLOGY

> "If you don't see the bug where you're looking, then you're looking in the wrong place." – The Second Law of Debugging [44].

This chapter serves a number of purposes. First, it provides a brief overview of debugger history in the last 30 years, and introduces some of the technologies that have predominantly influenced the development of debuggers during this time. The second aim is to provide the reader with a clear picture of where the GUARD debugger, and in particular the technique of relative debugging, fits into the evolutionary history of debuggers and debugging technology. Finally, the chapter focuses on the architecture of conventional debuggers and discusses why these are not suitable for implementing relative debugging. In particular, it highlights how traditional debuggers have been designed to facilitate the program development process, but do not necessarily address the requirements of program evolution.

WHAT IS DEBUGGING?

The definition of "debug" in the IEEE/ANSI Standard Glossary of Software Engineering Terminology [32] is given as "to detect, locate, and correct faults in a computer program". Myers [52] defines debugging as "the activity of diagnosing the precise nature of a known error and then correcting the error." The important distinction is that *debugging* is different from *testing*. Although it may be used as part of the testing process, debugging is also used at other stages in the software development life cycle. The particular debugging technique adopted will depend largely on the number of errors and the complexity of the program. One systematic approach commonly used is to address each error in turn using the following steps.

1. *Gather information.* This step is used to establish the nature of the error, the behaviour that is being observed, data structures likely to be involved, and a broad indication of the error location, such as the relevant module or subsystem. Information gathering is also sometimes referred to as *monitoring*.

2. *Analyse and locate.* Once sufficient information about the error has been obtained, it must be analysed to determine the cause of the error. This is generally a deductive process that

relies on a range of investigative techniques such as static program analysis, direct code inspection, interactive examination of program state, or a combination of these.

3. *Correct the error.* After the cause of the error has been identified, a solution must be proposed. The impact of the solution must be evaluated and once the changes have been made, the program tested to ensure that the error has actually been fixed and that no new errors have been introduced.

Most debugger research is devoted to the information gathering, analysis and location steps. Some literature distinguishes information gathering from debugging, but like McDowell and Helmbold [49], the author considers it an integral part of the debugging process.

SEQUENTIAL DEBUGGERS

Since early computer systems were almost exclusively sequential architectures, it is not surprising that programming languages and development environments also reflected the sequential nature of these machines. Debuggers were no different, and the early debugging tools were predominantly designed with a single thread of control in mind.

As early as 1975, a large number of debugging systems had been developed for a wide variety of architectures, operating systems and languages. Starting with tools designed to analyse memory dumps, debugging technology developed at a rapid pace, so that even by this time it was well recognised that debugging methods could be categorised into a few basic types: static analysis, event-based, interactive and post-mortem [47]. The debuggers themselves often implemented a combination of these techniques. Some researchers also distinguished debuggers as either static or dynamic [51], where static debuggers operated primarily on source code and dynamic debuggers operated on executing programs. A brief overview of these techniques follows.

STATIC ANALYSIS

This type of debugging method generally operates by performing a flow analysis of the static program code in much the same way as optimising compilers [68]. Such analysis usually consists of:

- *control flow analysis* – analysing the flow of control through the program in order to identify unexpected transfers of control, as well as presenting information about the structure and calling sequence of the program;

- *data flow analysis* – examining the use of variables within a program to detect errors such as references to uninitialised variables and variables that may have indeterminate values; and

- *inter-procedural flow analysis* – analysing control and data flow across procedure boundaries in order to examine the procedural structure, and identify data usage both within and across procedure calls.

EVENT-BASED

Another debugging technique is the event-action model. Here, interactions between the debugger and the program being debugged are viewed in terms of events. This type of debugger works by allowing a series of conditions to be defined. The occurrence of an event that satisfies the condition will cause an associated action to be triggered [35]. Typically, actions might alter the state of the program, display the contents of a variable or perform some other user-defined operation.

Another type of event-based debugger is one that utilises an event history to record information about the execution of a program [11]. This history can then be compared with the expected patterns of behaviour to isolate errors in the program. Event histories can also be used to replay the execution of the program to examine the state of a suspicious operation in more detail.

INTERACTIVE

By far the most common type of dynamic debuggers are interactive debuggers, although these are sometimes referred to as "breakpoint" debuggers [49]. Interactive debuggers are characterised by some form of user interface (generally a command line or a graphical user interface) that allows the user to issue commands that operate on an executing program. This provides a mechanism for controlling the execution of the program through the insertion of breakpoints, and then examining the state of the program once a breakpoint has been reached.

Interactive debuggers are generally employed using an iterative process, referred to as *cyclical debugging* [49]. This is a process familiar to many software developers, where a program is repeatedly stopped during its execution in order to examine the state. The programmer might begin by setting a breakpoint at a location that is suspicious or related to the error in some way. The program is then executed and will eventually reach the breakpoint. The state

of the program can then be examined and, based on the information obtained, the execution can be continued to another location. If execution has proceeded too far, then the program must be restarted in order to stop at some earlier point.

POST-MORTEM

The remaining debugging technique is that of post-mortem debugging. Here, the operation of an executing program is captured through the use of program traces or log files. The debugger is then able to replay the execution sequence of the program by consulting the trace information. This allows the user to repeatedly examine the program state in order to determine the cause of an error. More sophisticated debuggers may also provide animation of the program behaviour [36].

THE PARALLEL PROBLEM

Debugging is a difficult task because it requires understanding the software being debugged. Although attempts have been made to automate the debugging process, it still remains largely a job for humans. Advances in software engineering, such as information hiding, abstract data types and modularisation have simplified the identification and location of errors in sequential programs, however significant challenges still remain when debugging parallel programs.

By definition, parallel programs consist of many simultaneously executing processes. While some of these processes may synchronise for long enough to exchange messages through an inter-process communication mechanism, they must operate asynchronously in order to take advantage of the parallel architecture. In general, this asynchronous operation results in nondeterministic behaviour, and is the main reason that parallel programs are more difficult to develop and debug.

Unfortunately for debugger designers, the traditional debugging process does not always translate well to the parallel environment. The cyclical debugging technique may not prove effective because the error may be dependent on a particular order of process execution or a race condition that is affected by external factors. Further, the act of debugging the process itself may effect its timing and hence whether the error is apparent or not, an occurrence referred to as the *probe effect* [25]. In an attempt to overcome these problems, techniques such as event-based debugging or static analysis are often employed, with varying degrees of success.

As well as dealing with the nondeterminisim of the programs, a parallel debugger must also be able to manage the complex nature of multiple simultaneous executing processes and the resulting large volumes of information that can be generated. In addition, most parallel programs decompose their data, which may in turn be physically distributed across the different processes. This also raises problems for the developer, because the specific data decomposition used may be complex, so it may be difficult to identify where data is located and the particular point that errors are being introduced.

PARALLEL/DISTRIBUTED DEBUGGING

In the early 1980's, concurrent programming was beginning to be widely accepted and parallel computers were becoming commercially available. The first debuggers available for parallel systems were interactive serial debuggers that had been modified to handle multiple processes, such as "dbxtool" [8]. Some debuggers were also available for specialised concurrent languages such as the "defence" debugger for concurrent Euclid [70] and the "YODA" debugger for Ada [37]. By the late 1980's and early 1990's a wide range of debuggers for parallel and distributed architectures had been developed. McDowell and Helmbold [49] categorised these debuggers as follows.

EXTENSION OF TRADITIONAL TECHNIQUES

This refers to the technique of using a collection of sequential interactive debuggers, one per process to debug a parallel program. This technique raises a number of difficulties, including how the output from multiple debuggers can be displayed in a coherent fashion, and how the multiple debuggers are coordinated and controlled. While these debuggers tend to be easy to build, they are most restricted by the probe effect. Many debuggers in this category are (or were) commercially available, such as the "dbxtool" debugger mentioned above, as well as debuggers such as "CXdb" [20], "TotalView" [22], "HP/DDE" [27] , "pdbx" [30] and many others.

EVENT-BASED DEBUGGING

This debugging technique is an extension of sequential event-based debugging. These debuggers generally maintain a history of events (although the definition of *event* varies) and allow the user to browse the history, replay the execution or simulate the environment in order to debug the execution of individual processes. In the parallel environment, events may also include messages or inter-process communication. By providing a deterministic replay of events, these debuggers can also minimise the impact of the debugger on the

execution of the parallel program. Typical debuggers in this category are "HDL" [11], "Clouds" [40] and "Ariadne" [19].

STATIC ANALYSIS TECHNIQUES

For parallel programs, static analysis can be used to detect various types of errors, including synchronisation errors such as deadlock and other timing problems, and data-usage errors such as the simultaneous access of shared variables. McDowell and Helmbold suggest that static analysis debuggers cover two distinct areas: (1) applying dataflow analysis techniques to parallel programs, and (2) determining if two statements in a parallel program can be executed in parallel [49]. This latter area applies to debuggers that perform a data dependency analysis on the target program to determine the location of schedule-dependencies [15]. Static analysis debuggers do not suffer from the probe effect at all, since the program is never executed. Recent debuggers that employ these techniques are "DETOP" [71] and "PTOOL" [15].

OTHER DEVELOPMENTS

Two major advances in debugging technology have been in the use of graphical display systems, and in the use of integrated development environments (IDEs). The introduction of GUIs in the 1980's was quickly exploited by debugger designers, since a multi-window environment provided an ideal mechanism for enhancing the display of key information such as breakpoint location and variable data. Traditional parallel debuggers could also take advantage of the GUI for the control and coordination of multiple processes. The GUI also allowed novel visualisation techniques to be introduced, such as viewing complex data structures, time-process diagrams and the animation of program execution and data manipulation. Although integrated development environments have been available for considerable time, the combination of IDE and GUI has seen widespread proliferation of these systems.

In 1997, the Parallel Tools Consortium sponsored the High Performance Debugging Forum. The aim of the Forum was to define a set of standards relevant to debugging tools for high-performance computing (HPC) systems. In November 1998, Version 1 of the HPD Standard was announced [28]. Until this standard was established, there had been no published standards or definitions of debugger user interfaces or functionality. The consequence of this has been that debugger implementations differ significantly. With the standard now in place, users can be confident that debuggers will provide at least a base level of consistency and functionality across a wide range of HPC platforms.

DEBUGGER IMPLEMENTATION

In the last decade, the number and types of debuggers has increased substantially. In one non-exhaustive literature survey conducted by the author, it was found that well over 100 parallel and sequential debuggers have been developed between 1990 and 1999 (See Appendix F). Many of these debuggers combine a number of the techniques described above, and significant efforts have been made to extend these techniques to better support parallel environments. For example, the "Panorama" debugging environment [48] combines traditional debugging features with a post-mortem debugger into a portable and extensible package. A few debuggers have developed substantially new technologies, such as the "RAID" debugger [13], which employs probabilistic reasoning, heuristic debugging knowledge and structural analysis to automate the debugging process.

Early debuggers only required the ability to examine memory and processor state information that had been saved to a dump file. The level of sophistication has increased significantly in more recent debuggers. Many debug tools, particularly interactive debuggers, provide features such as the ability to set breakpoints, to single step programs, to read and write the memory locations and registers, and to trap program exceptions and memory accesses. At a minimum these features need to be supported by the host operating system since they generally operate outside the memory protection schemes of most systems, or require a transfer of control to the debugger such as when a breakpoint is encountered. On a number of systems some or all of these features are also supported at the hardware level.

Most debuggers today exist as stand-alone applications that operate in their own address space. In a multi-user protected-memory operating environment, a debugger must be able to both gain access to and control the program being debugged. For interactive debuggers, this is usually achieved by attaching to the program in some operating system specific manner. Event-based debuggers usually require a special library to be linked with the target program or statements inserted into the source code, although some debuggers are able to monitor hardware busses passively. Static analysis debuggers, of course, do not require any special support at all since the program is never actually executed.

Typically, the operating system will provide the debugger designer with the necessary debugging tools. In the case of the UNIX operating system for example, the `ptrace()` system call is available for just this purpose. In order to use `ptrace()` the process to be debugged must first be "stopped" by sending it a signal. The debugger is then able to take

control of, and perform debug operations on the process by issuing `ptrace()` calls with the appropriate arguments.

Because of their special requirements, debuggers tend to be particularly dependent on specific operating systems and architectures. Some attempts have been made to develop more portable debuggers, such as "ldb" [55], "gdb" [66] and "p2d2" [16], the first of which appeared in the early 1990's. The approach taken by all these implementations is to isolate the machine dependent code from the portable code. In the case of "ldb", this is achieved by defining machine independent classes that describe important abstractions in "ldb". Machine dependencies are then defined as subtypes of these classes. "Gdb" takes a simpler approach by isolating the machine dependencies to separate source files that are managed using pre-processor directives when the debugger is built. The "p2d2" debugger [16] has been designed with portability as a key objective, and accomplishes this using a client/server architecture. The "p2d2" client, which provides a uniform debugger user interface, contains no machine dependent code. All implementation dependencies are isolated to the servers, which in turn employ "gdb" to provide the low-level debugger functionality. The portability of "gdb" ensures that the debug servers are available for a wide range of architectures.

SOFTWARE DEVELOPMENT V. SOFTWARE EVOLUTION

Current debuggers provide good support for the software development process. Static debuggers provide the developer with a preventative mechanism for locating and determining problems before they are encountered. Event-based debuggers, particularly those that are equipped with sophisticated visualisation features, aid the developer in analysing the behaviour of programs through the use of time-process diagrams or the animation of program activity. Interactive debuggers incorporate features to identify and determine the location of errors, to monitor the operation of a program as it is executing, and to examine the state of a program that has stopped or failed. Some debuggers combine two or more of these capabilities.

Unfortunately, these debuggers are not as effective in supporting the software evolution process. In most cases, where incremental changes are made to existing software or where a program is ported from one architecture to another, a working reference program is available. Ideally the developer should be able to take advantage of such a reference program in order to assist in locating errors in the new code. A small number of existing debuggers do go some way to meeting these needs. For example, the III programming

environment [46] provides an integrated development and debugging environment that allows the developer to make incremental changes to executing programs. However none of these systems allow the developer to make direct comparisons with a reference code.

RELATIVE DEBUGGING

The technique of relative debugging was first introduced in 1994 as a result of a large effort to port existing sequential and vector codes to the new parallel computers that were becoming widely available by this time. Unfortunately, the porting process tends to be a difficult and specialised one, as is reflected in the lack of high quality automated parallelising systems available even today. However, it is often possible to use information from these known, working codes in order to simplify the porting and subsequent debugging process. Relative debugging is the first technology that has been specifically designed to aid software evolution by allowing the developer to make such comparisons.

The design of a relative debugger has required the development of a number of technologies that are not available in traditional parallel (or sequential) debuggers. These technologies are necessary for the following reasons:

- Typically a sequential program will be executed on a different architecture to a parallel version of the same code, which means that even simple data comparisons can be problematic if the architectures have different byte orderings, word sizes or floating point representations.

- The sequential and parallel programs will in most situations, be located on separate machines. This implies that the debugger must be able to control two or more programs, each of which may possibly be running on a remote computer system.

- In order to be useful to the developer, the comparison process must be reasonably automated. Since the programs are operating asynchronously, this requires a sophisticated control mechanism to manage the extraction of data at arbitrary points in the execution of each program.

- A data structure may undergo significant changes in the sequential-to-parallel porting process through data decomposition or other transformations. In order to compare such a data structure with the original version, some mechanism to replicate and/or undo these changes must be available to the debugger.

While some debuggers may address a few of these issues (such as "p2d2" which operates in a client/server manner) we know of no debugger that deals with them all in an integrated fashion that would support the process of relative debugging.

Abramson and Sosič developed an early proof-of-concept version of GUARD in 1995 [4]. This version provided a simple interactive framework that was limited to debugging two separate images on a local machine. In particular, the comparison-matching logic was a simplistic design that prevented the relative debugging technique from being used in a truly parallel environment. Since then, the original program has been completely redesigned to incorporate a number of new technologies developed by the author. Client/server technology has been employed to expand traditional interactive debugging techniques to parallel architectures. The implementation now utilises novel and unique methods to address the heterogeneity issue, provides a dataflow control mechanism to manage the complex control requirements, and provides techniques for the decomposition and transformation of data.

In terms of the overall debugger genealogy, GUARD evidently falls into the category of the parallel interactive debugger. However it has some significant improvements over many of the "standard" debuggers of this type. Unlike many of these debuggers, GUARD is now capable of controlling multiple independent *programs* rather than just a single parallel program. In addition, through the use of its internal dataflow engine it is highly programmable and can perform automatic data matching. Finally, it has the ability to perform powerful transformations on user data while a program is executing, a capability that is not available in any other debugger.

Future versions of GUARD could clearly take advantage of other debugging techniques. In particular, static analysis could be used effectively to automate the creation of assertions through data dependency analysis. Also, the ability to maintain an event history could easily be used to streamline the process of data comparison. The use of a GUI could provide a great deal of scope for improving user interaction and data visualisation. These issues will be discussed in more detail in Chapter 8.

Chapter 3

RELATIVE DEBUGGING

In 1994, Abramson and Sosič [4] developed the technique of relative debugging to assist in the porting of sequential programs to parallel architectures. Relative debugging, like most good ideas, is deceptively simple: utilise the availability of a known, correct version of a program to aid the debugging of later releases of the software. However, it turns out that the technique is somewhat more difficult to implement. Abramson and Sosič were able to produce a working debugger, but this was restricted to debugging sequential programs, and only provided limited support for changes that result from software engineering practices or when software is ported to parallel systems. Since then the author has significantly advanced the technology in a number of key areas to address these shortcomings. The work undertaken to date includes:

- formalising some common transformations that occur to programs as a result of software engineering and when programs are parallelised, and developing techniques for supporting these transformations;

- solving the complex control issues that arise when applying relative debugging to parallel programs; and

- building support for debugging programs on remote machines and addressing the heterogeneity issues that this raises.

This chapter comprises three parts. First, the technique of relative debugging is described in detail. In particular, key aspects of relative debugging are examined, including the use of imperative and declarative comparisons, the use of comparison tolerances when defining assertions, how differences can be reported and how the technique can be applied to parallel programs. The chapter then identifies the transformations that arise from common software engineering practices in program evolution, and how these must be addressed in order to support relative debugging. These transformations include data parallel decomposition, variable promotion and loop fusion, shape transformation, index permutation and array slicing. The remainder of the chapter then describes a range of issues that must be overcome when designing a practical parallel relative debugger.

RELATIVE DEBUGGING

Software evolution is now widely recognised as a process that involves the initial software development followed by ongoing maintenance releases that accommodate enhancements and corrections to the code, and changes to the operating environment [39]. Debugging is a key part of this evolutionary process, but few if any debuggers have been designed to take advantage of the inherent informational content available in previous software releases. In many situations such as after minor corrections have been made, or when programs are ported from one architecture to another, key data structures within a program remain invariant between releases. Because of this, it is often possible to compare data structures between one release and the next as an aid to identifying the cause and location of introduced errors. In these situations, relative debugging provides a powerful technique for locating such errors quickly. Relative debugging is the first debugging technology to exploit this capability.

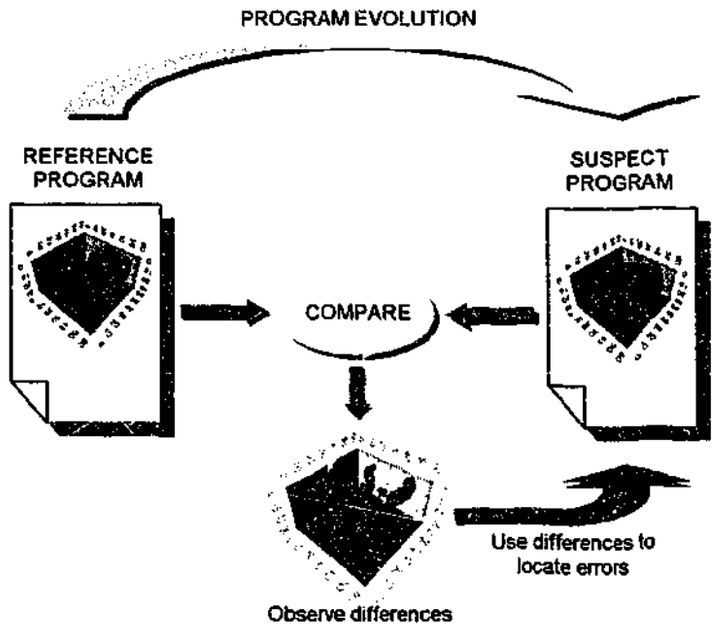


Figure 3.1: Relative Debugging

Relative debugging is a high level technique that allows data in a *reference* program to be compared to that of a *suspect* program [5,64], as shown in Figure 3.1. By observing the differences in key data structures, important clues about the nature and location of the errors can be obtained. Various case studies reporting the results of using relative debugging have now been published [3,4]. An important feature of relative debugging is that it is both

language and machine independent. This allows a user to compare data structures without concern for program complexities, and thus attention can be focussed on the cause of the errors rather than implementation details. While the technique applies equally well to all kinds of data structures, current research has focussed on reporting differences in scalars and arrays only. This is because it is not immediately obvious how to compare complex, dynamic data structures such as lists or trees, nor is it easy to visualise the differences in such structures. This is an area that requires significantly more research, and is discussed further in Chapter 8.

To the user, a relative debugger appears like any traditional interactive debugger, but it also provides additional functionality that allows data from different programs to be compared. A user specifies the comparisons they wish to make using two distinct mechanisms:

- *imperative comparison* – the user issues comparison commands to immediately compare data between two programs and observe the differences; and

- *declarative comparison* – the user defines a series of assertions that specify the conditions necessary for correct execution of the programs.

The user must also define what is meant by "different" and does this by declaring tolerance values for each comparison. The results of the comparisons can then be examined using data visualisation techniques.

IMPERATIVE COMPARISON

Relative debugging extends the traditional interactive debugging technique of displaying program state information by allowing the user to make immediate comparisons between two executing programs. The user does this by first ensuring that each of the programs has been halted at a breakpoint and then issuing a command instructing the debugger to perform a comparison. Provided that the two data structures are the same size and shape, the debugger will extract the data from the respective programs and then perform a numeric subtraction on each pair of corresponding elements. Differences that are present in the data will then be displayed to the user. This technique works well for simple situations or where the user requires immediate feedback on the program state, but because of the high level of user interaction required, can become unwieldy in certain situations such as within program loops.

A relative debugger provides support for imperative comparison using the `compare` command. An example of such a command might be as follows:

```
compare ref::RefVar = sus::SusVar
```

In this example, the `compare` command will instruct the debugger to extract the contents of `RefVar` from the program[1] *ref* and `SusVar` from the program *sus*, perform a comparison, and then report the results to the user.

DECLARATIVE COMPARISON

Relative debugging provides an alternative technique to address the limitations of imperative comparisons. In many circumstances the user may wish to specify *a priori* a set of conditions (known as *assertions*) that must be satisfied for the correct execution of the program. These assertions combine the imperative comparison information with specific location details so that the debugger is able to perform the comparisons automatically. Because the comparison process is now automated, this also addresses the situation where a high degree of user interaction is undesirable. As the name suggests, declarative comparisons are not performed immediately, but instead the assertion information is stored internally in the debugger. This allows the user to issue multiple assertion commands, and at some later time instruct the debugger to use the assertion information to automatically set the appropriate breakpoints, control the execution of the programs, and extract and compare data from key data structures. The comparison results can then be used to identify the likely location of errors in the program.

Choosing an appropriate breakpoint location is an important aspect of defining assertions. If the locations refer to source lines in the body of a program loop structure, then the assertion will be executed on each iteration of the loop. This allows the accumulation of differences in the data structures to be observed dynamically, and the location of the error to be readily pinpointed. Another useful approach is to choose locations before and after such loop structures. This can then be used to verify that the data remains invariant across the loop, and provides a useful tool to establish the correctness of the code region.

A relative debugger provides support for declarative comparisons using the `assert` command. This command is similar to the imperative `compare` command, but as

[1] Throughout this document the term "program" is used synonymously with "process" and refers to the instance of an executing code. In the case of parallel architectures a "program" refers collectively to many concurrently executing instances. We use "process" to refer to a specific instance if necessary.

mentioned, includes additional breakpoint information that is used by the debugger to manage the program execution automatically. An example of such a command might be as follows:

```
assert ref::RefVar@"prog1.c":4300 = sus::SusVar@"prog2.c":4400
```

This example also compares data in `RefVar` from the program `ref` with `SusVar` from `sus`, however the user is also able to specify the locations at which this comparison will take place. In this case the value of `RefVar` is obtained from the program when line 4300 in the source file `prog1.c` is reached, and `SusVar` is obtained when line 4400 in `prog2.c` is reached.

The full syntax for comparisons and assertions is provided in Appendix A.

COMPARISON TOLERANCE

When performing comparisons it is possible that errors may be incorrectly attributed to differences in the precision of the program variables or because of other minor numeric factors. In order to avoid this situation the user is able to specify a tolerance value. Variables are considered equivalent when the result of a comparison is within the tolerance. Tolerance can be specified in two ways:

- *absolute tolerance* – the magnitude of the difference between the variables is compared directly with the tolerance value. If v_1 and v_2 are the values of two variables, then for some tolerance ε, differences will be ignored if $|v_1 - v_2| < \varepsilon$;

- *relative tolerance* – in situations where the values being compared are very small but the differences still constitute a significant error, using absolute tolerance may swamp the difference value. Instead the difference can be first divided by the maximum of the two variables before being compared to the tolerance. If v_1 and v_2 are the values of two variables, then for some tolerance ε, differences will be ignored if $\frac{|v_1 - v_2|}{\max(|v_1|,|v_2|)} < \varepsilon$.

REPORTING DIFFERENCES

A key component of relative debugging is the reporting of differences that occur in the data structures being compared. The simplest method is to display the magnitude of the numerical difference between two values. This technique is generally used only for quickly

displaying differences in simple data structures, as it becomes much too unwieldy for large amounts of data such as when assertions are located within program loops or when large arrays are being compared. For more complex data structures, some form of data visualisation is necessary to view the difference information.

The errors in two-dimensional array data structures can be conveniently displayed as a pixel map[2], where each pixel represents the difference between corresponding elements in the arrays. The pixel map can then be used to visually determine the location of the differences in the data structures by setting the appropriate pixel to indicate the presence of a variation in the data. Colours can also be assigned to represent the relative magnitude of the differences as a means of providing additional information. Examples of typical pixel maps are shown in Figure 3.2.

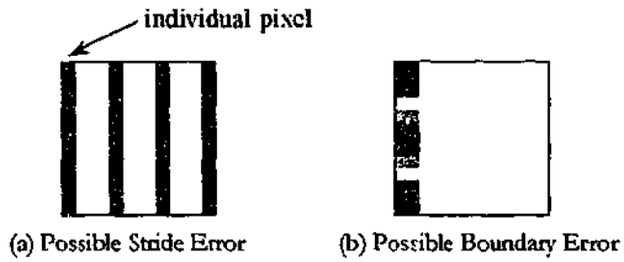


Figure 3.2: Example Pixel Maps

The pixel map in Figure 3.2(a) uses black to represent a difference and white to indicate that any difference is within the acceptable tolerance value. This example shows the typical characteristics indicating the possibility of a stride error that appears periodically in columns of the array. In Figure 3.2(b) colours have been assigned to the relative magnitude of the differences. This example shows a possible boundary error appearing in the left-most columns of the array. The colours indicate that the magnitude of the error depends on its distance from the middle of the column.

Where data structures have three or more dimensions, complex visualisation techniques are required. In programs that compute data in a time-step loop it is also useful to view differences as a series of frames to provide a graphic indication of the development of the errors. These frames can then be run consecutively as a movie. Figure 3.3 show several

[2] A two-dimensional arrangement of pixels.

frames of a three-dimensional data structure where the differences have been displayed using an iso-surface representation.

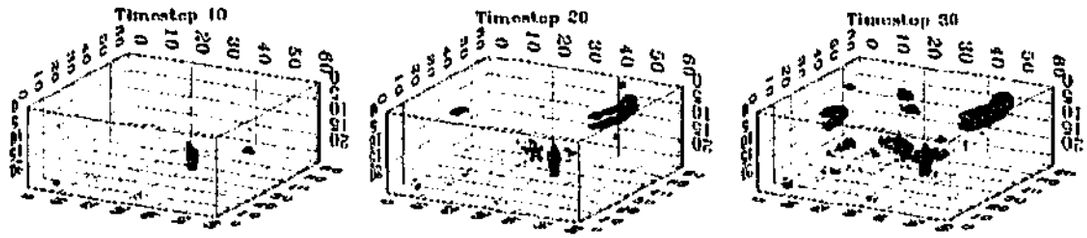


Figure 3.3: Visualisation of Differences in 3-D Data Set

There is significant evidence to suggest that visualisation of comparisons, particularly using 2-D and 3-D representations, provides the user with a means of characterising patterns of differences. In a previous study [2], a time series iso-surface representation of the error (shown in Figure 3.3) was used to identify independent errors in a meso-scale weather model. In particular, the structure of the iso-surface allowed the identification of errors in different code sections that were responsible for various physical processes. For example, an error in the physics on the planetary boundary appeared as an iso-surface that was visible at the bottom of the 3-D space used to represent a slice of the atmosphere. A second error occurring in the long-wave radiation physics code was visible in the top of the atmosphere. In another case study [69], differences showed characteristic periodic behaviour suggesting problems involving trigonometric operations. In each of these cases generalisations can be made about the nature of the patterns, but until further research is conducted in this area these are currently limited to being used for insight when making deductions about the nature of the errors.

PARALLEL RELATIVE DEBUGGING

Although initially developed to address the problems encountered when porting programs to parallel architectures, early implementations of relative debugging were limited to only supporting the comparison of sequential codes (or single process parallel codes). Conceptually, parallel relative debugging extends the relative debugging paradigm by allowing the comparison of data in a parallel program with the corresponding data in the sequential program from which it is derived. Figure 3.4 show a high level model for how this might occur.

When traditional sequential or vector programs are modified for execution on a parallel computer, it is often necessary to re-organise the key data structures and associated code. For example, if the parallel platform has physically distributed memories then the data must be partitioned and allocated to the individual processes, and in some cases the loop structures themselves must also be altered. Transformations may also be automatically introduced by parallelising compilers, or when converting codes to data parallel languages.

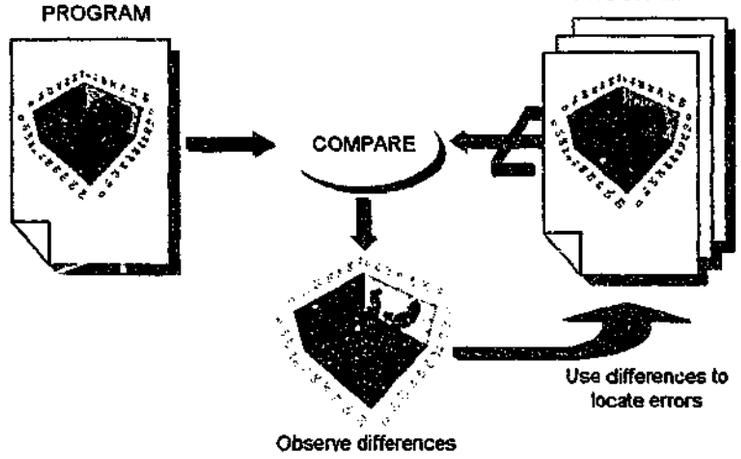


Figure 3.4: Parallel Relative Debugging

In the above diagram, a sequential data structure has been decomposed into smaller blocks that are distributed across the parallel processes. The user may wish to compare the sequential data with the parallel data, but does not need to be concerned with the details of the particular decomposition that has been used. Ideally, the debugger will be responsible for arranging for each of the distributed blocks to be recombined into a single data structure before the comparison is performed. In addition, the debugger must also manage the program synchronisation so that the data can be extracted from the correct location in each process.

The existing declarative comparison command provides the debugger with all of the information it needs to perform such a comparison, apart from a description of the data decomposition technique that has been employed. All that remains is for the user to describe

this mapping in order to provide the debugger with the details of the decomposition method. This may be achieved through a statement such as:

```
define mapping() = ... description of data decomposition ...
```

Although such a definition can be complex, once it has been established the user no longer needs to be concerned with how the decomposition takes place. Instead, the mapping can be used in combination with a normal assertion statement to perform a comparison between a sequential and a parallel data structure. For example, a typical assertion might be:

```
assert seq::SeqVar@"prog1.c":4300 = mapping(par::ParVar@"prog2.c":4400)
```

Here, `seq` refers to a single sequential program containing the array `SeqVar`. However, `par` now refers to a collection of parallel processes, each of which contains a portion of the decomposed array in the variable `ParVar`. The data decomposition description provided by `mapping` will extract data from each of the parallel processes and recombine the data in a way that allows the comparison with data from the sequential program.

Data decomposition is not the only type of transformation that can be applied to a sequential program when it is parallelised, and these additional transformations must also be supported. The next section discusses the range of transformations that must be addressed if relative debugging is to prove a useful tool to aid in the evolution of software.

Software Engineering and Relative Debugging

Many factors contribute to the continuing evolution of software. Changes may arise as a result of:

- errors that are identified and corrected as part of a regular maintenance process;
- the desire to exploit the availability of new languages and architectures; and
- the impact of other external factors such as economic and social constraints.

In these situations it is sometimes necessary to alter a particular data structure, perhaps to take advantage a new architectural feature or because of restrictions imposed by different programming languages. Because of such changes, it may no longer be a simple process to perform a comparison with previous releases of the software. However in many cases the actual data remains invariant, the changes may just involve altering the ordering of the data or

the addition of extraneous information to the data structure. In these situations, the ability to perform comparisons using relative debugging can still be an extremely valuable tool for locating errors.

The types of transformations that occur because of such software engineering changes are particularly important in parallel relative debugging. The user may wish to compare data structures at various places without concern for the different organisation of the data in each of the programs. To support this, the debugger must provide a mechanism to replicate the transformations that have been applied to the programs. It is also desirable that such a mechanism be transparent to the user so they can concentrate on identifying the location of the error, rather than on the particular implementation.

Although the number and types of changes that may be introduced to program data structures is virtually unlimited, this research has focused on those changes that commonly occur to array data structures as a result of software engineering practices, and the reorganisation of code and data as programs are parallelised. A number of researchers have examined the classification of data transformations, particularly in relation to data locality issues for cache optimisation [58,72]. We are not attempting to duplicate this prior work, but rather identify a subset of common transformations that are both of interest to the debugger user, and allow us to establish the validity of the technique. Addressing the changes to other types of data structures is left for future research.

Some of the transformations that are applied to data and code when a sequential program is parallelised consist of:

- *data parallel decomposition* which defines the data distribution method used to implement a parallel version of the program; and

- *temporal displacement* which deals with situations where a variable is promoted (such as scalar to array) or loops have been fused to optimise parallelisation.

A number of transformations are also used in a more general software engineering context, including:

- *shape transformation* where the rank of an array or the size of its dimensions have been altered to exploit a particular architecture or language features;

- *index permutation* where array indices have been reordered, such as through architectural optimisation or language differences; and

- *array slicing* where additional information has been added to an array, such as in the form of guard bands.

The following sections will examine these transformations in detail and show how they are used to support the process of relative debugging.

DATA PARALLEL DECOMPOSITION

Except for extremely coarse-grained or parameterised models, parallelisation of an algorithm on a distributed memory computer usually requires the decomposition of arrays in order to distribute data to the individual processes. In the case of data parallel languages, this decomposition is typically a *block*, *cyclic* or *block-cyclic* decomposition [24,29] and is handled automatically by the language runtime system. These types of decomposition are also typically used in automated parallelising systems [61]. Hand coded distributed memory implementations may produce much more complex algorithms, and are beyond the scope of this thesis. In many cases, particularly where process pool sizes are dynamic, the exact partitioning is not normally known until runtime.

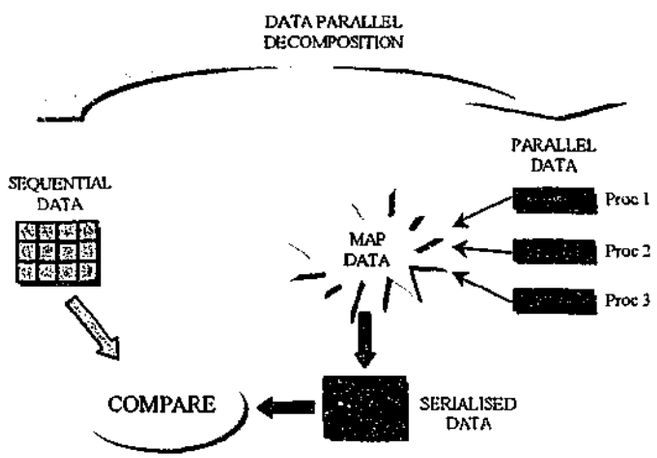


Figure 3.5: Duplication of Data Distribution

Figure 3.5 shows an example in which serial and parallel data are compared using relative debugging. In this situation, the sequential data has been distributed to multiple processes as a result of the parallelisation process. Since an assertion requires two identical data structures for comparison, the distributed data must be recombined before the comparison can take

place. This is achieved by mapping the distributed data into a serialised data structure that can be used in order to perform the comparison operation.

VARIABLE PROMOTION

Variable re-use is a common programming construct. When this occurs, a scalar value may be calculated, used and then overwritten in a new computation, usually within some form of loop construct. However, when this type of program is parallelised, it is often necessary to promote the scalars to an array, one unique value per loop iteration. Such a variation between the sequential and parallel codes makes it extremely difficult to compare the data, because there may be no point in time in which the data exists concurrently in both programs. Figures 3.6 shows code fragments demonstrating how a temporary variable in Fortran code might be promoted to an array written in C.

```
      DO 90, I = 1, 100
80      TEMP = X(I) * Y(I)
        X(I) = 2 * Y(I)
        Y(I) = PI * TEMP
90    CONTINUE
```

(a) Fortran code

```
128  for (i=0; i<100; i++)
129  {
130    temp[i] = x[i] * y[i]
131  }
132
133  for (i=0; i<100; i++)
134  {
135    x[i] = 2 * y[i]
136    y[i] = PI * temp[i]
137  }
```

(b) C code

Figure 3.6: Variable Promotion Example

Since temp is an array in Figure 3.6(b), and TEMP is a scalar in Figure 3.6(a), comparison is not possible unless some mechanism for promoting TEMP to an equivalent array is provided. At present, the only way to perform this comparison is to modify the Fortran code in order to promote TEMP to an array. This is clearly undesirable because it requires the modification of the reference program, possibly resulting in the introduction of errors or other changes in program behaviour.

LOOP FUSION

Another common optimisation for parallel programs is to fuse a number of sequential loops into one larger parallel one. Serial codes often perform computation on entire rows or in the case of 2 dimensional arrays entire columns, for vectorisation purposes. In a parallel program, multiple computations may be fused into a single calculation on each grid element, since in general each process will be applying the same computation to the partitioned data. This has the effect of reducing the overhead of parallel loops while improving data locality

and reuse [34,50]. However, where loop fusion has been employed there will never be a point in the execution of the codes when the arrays are equivalent. Figure 3.7 shows code fragments where this occurs.

```
128  for (i=0; i<100; i++)
129    x[i] = 2 * x[i]

131  for (i=0; i<100; i++)
132    y[i] = x[i] + y[i]

134  for (i=0; i<100; i++)
135    x[i] = y[i] / x[i]
```

(a) Serial code

```
100  for (i=0; i<100; i++)
     {
102    x[i] = 2 * x[i]

104    y[i] = x[i] + y[i]

106    x[i] = y[i] / x[i]
     }
108
```

(b) Parallel code

Figure 3.7: Example of Loop Fusion

In this example, comparison of the array x prior to the execution of line 131 in Figure 3.7(a) and 104 in Figure 3.7(b) is not possible. Instead, a temporary array must be constructed to hold the intermediate values from the parallel code, and the comparison delayed until all elements of this temporary array have been obtained, i.e. line 108 in Figure 3.7(b).

SHAPE TRANSFORMATION

Sometimes it is necessary to change the shape of program data structures in order to exploit language or architectural features. For example, data may be stored in a long single dimensioned vector to exploit a vector architecture, but this data may need to be re-organised into a multi-dimensioned array to take advantage of an array-based language. In other situations the shape of arrays may be altered to exploit processor performance improvements, for load balancing purposes, or for better cache utilisation [50,58].

In order to allow the comparison of transformed data structures, a relative debugger must provide a shape transformation mechanism. This allows the debugger to duplicate the common shape transformations that are applied to data in order to facilitate comparison in situations where these transformations have been utilised. Figure 3.8 shows an example of this type of transformation.

In relative debugging, shape transformation is supported using the `trans` command. This command allows the user to define a shape transformation and then apply that transformation to data that is used to perform a comparison.

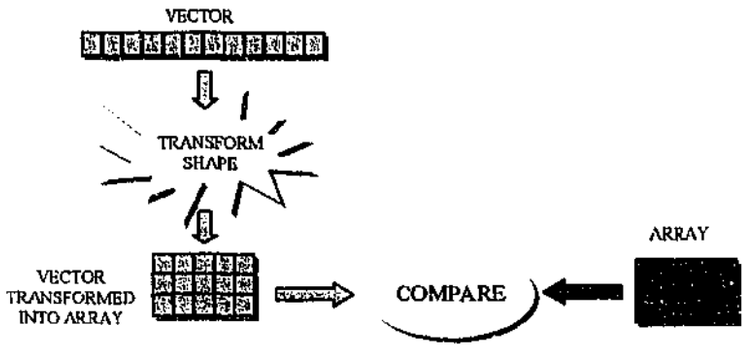


Figure 3.8: Shape Transformation

INDEX PERMUTATION

A program that is ported from a vector to a cache-based architecture may have been optimised for the vector processor and so will scan the data in a particular way. However, this may not be suitable to take advantage of the capabilities of the cache-based machine. In such a situation the order of array accesses may be need to be altered so that data locality is improved to exploit the processor cache, thereby increasing performance [58]. In other situations, permutation of array indices may be used as an alternative to loop transformations [17] or may result from language differences, such as between C and Fortran.

In general, comparison of arrays in these situations using relative debugging requires a mechanism that preserves the number of elements and the array contents, but allows arbitrary permutation of the indices. Figure 3.9 shows an example of how a typical permutation, in which a 3×5 array is transformed into the equivalent 5×3 array, can be compared using relative debugging.

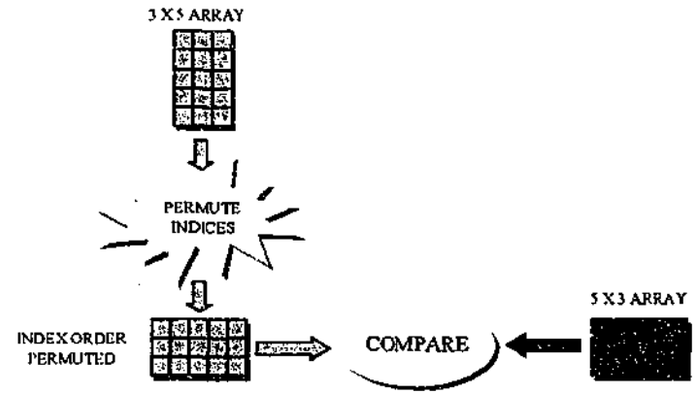


Figure 3.9: Index Permutation

ARRAY SLICING

In some circumstances, perhaps through modification to the boundary conditions of an algorithm, additional rows or columns may be added to an array. Both the imperative and declarative comparison techniques operate on all elements in the respective data structures, so direct comparison of arrays under these conditions is not possible. Instead, it is necessary to provide a mechanism to allow the extraction of a sub-array from the expanded array that is the same size and shape of the array with which it is to be compared. An example of this array slicing process is show in Figure 3.10. In this example, extra elements have been added to the ends of each row and column of 2-dimensional array, increasing the size of the 2×3 array to 4×6 elements. These extra elements must be removed before the array can be compared with a 2×3 reference array.

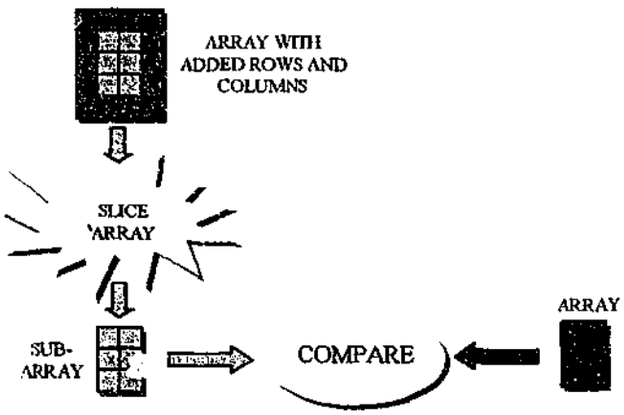


Figure 3.10: Array Slicing

DEBUGGER DESIGN ISSUES

In addition to the changes that are introduced into programs through software engineering processes, a relative debugger must also deal with programs that are executing simultaneously, possibly on physically separate machines that may have completely different architectures. A relative debugger also gives the user the unique ability to define a series of assertions that instruct the debugger on how to control and compare data extracted from many concurrently executing programs. Because of these requirements, the debugger design must address some key issues. These include:

- *process control requirements* – how the debugger will control independent processes possibly executing on remote machines;

- *heterogeneity issues* – how to deal with the architectural differences between the machines executing the programs being debugged, and the debugger host system itself; and

- *debugger control logic* – the mechanism that will be used to process the assertion commands, control the executing processes, and extract and compare the target data.

The following sections describe these design issues in detail. The next chapter will present a debugger architecture that addresses all of these issues, and incorporates the features into a powerful and innovative package.

PROCESS CONTROL

By definition, a relative debugger must be able to control two or more processes simultaneously. While many debuggers are designed to do this, the processes they control usually consist of tasks comprising a single parallel program, or threads comprising a single sequential program. The processes controlled by a relative debugger are almost always from completely independent programs, and in many cases may be executing on physically remote computer systems. In addition, the debugger must be capable of controlling a combination of sequential and parallel programs.

Because of its unique re[illegible]uirements, a relative debugger must provide a sophisticated process control mechanism. This mechanism must include the ability to:

- *control multiple independent programs* – since each assertion compares data from two different processes (which may comprise all or part of a sequential or parallel program respectively) and there are no restrictions on the number or combinations of processes used in assertions;

- *mix sequential and parallel processes* – the debugger must be able to compare data from parallel and sequential processes in order to fully support the concept of parallel relative debugging;

- *control processes on physically remote machines* – relative debugging supports comparison of data in a number of situations where the processes may be located on remote machines. This includes sequential programs running on different architectures and

the processes comprising a parallel program executing on a distributed memory or cluster architecture;

- *control both individual and arbitrary groups of processes* – comparison of data between parallel and sequential programs requires that the debugger manage the extraction of data from arbitrary processes in order to account for the data distribution method employed;

- *start and stop debugging processes arbitrarily* – so that the user is free to choose which versions of a program will be used for comparison; and

- *support as wide a range of parallel and sequential architectures as possible* – in order to ensure that the debugger provides a useful tool to aid the process of software engineering.

Clearly, a debugger that satisfies these requirements will need to employ a client/server architecture. In addition though, the control mechanism employed must be able to multiplex control across many independent processes and handle asynchronous events as they occur. The debugger must also be able to support a number of different, architecture specific mechanisms for parallel process startup and control.

HETEROGENEITY

A relative debugger has the ability to compare data from programs executing on completely different systems. However, these machines may employ completely different architectures, and consequently the data representations used may not be compatible. Further, the machines executing the programs that are being debugged might use different data representations from the debugger itself. The obvious approach to addressing this issue is to employ some mechanism for dealing with data in an architecture neutral manner. Such a mechanism must:

- be compatible with a client/server architecture;

- allow data distribution across networks and support in-core operations;

- retain data type information;

- convert to/from native formats without loss;

- support a broad range of arithmetic, logical and comparison operations;
- provide an efficient storage and execution mechanism; and
- support a wide range of architectures.

While a number of portable data format exists, none of them address all these issues. To overcome the limitation of these existing implementations, a completely new architecture independent format (AIF) has been developed. The details of this format are described in the next chapter.

CONTROL LOGIC

In addition to data and code changes, parallel computers also increase complexity since they must deal with many concurrently executing processes. This raises problems for parallel relative debugging because these multiple processes need to be controlled and synchronised if data is to be extracted and compared with a sequential counterpart. In order to support this, the declarative assertion mechanism must be extended to ensure that such synchronisation remains transparent to the user.

One of the primary tasks of a relative debugger is to evaluate a series of user-defined assertions in order to perform a comparison of data that has been extracted from two or more executing processes. Each assertion instructs the debugger on *what* data to extract from each process and *where* the data is to be extracted. In the case of parallel programs, data may be extracted from the same location in a number of processes simultaneously. Once data has been extracted, it may be manipulated or transformed in arbitrary ways before the comparison is finally performed.

In order to evaluate the assertions, a relative debugger proceeds as follows. First, it must set a breakpoint in each process and at each location specified in the assertion statements. This may involve setting multiple breakpoints in some processes. Once all the breakpoints have been set, the processes are started. Each process will reach a breakpoint at some arbitrary time, and when this occurs the debugger must extract the data that has been associated with that breakpoint by an assertion statement. This data must then be stored until data from the corresponding process is available for comparison. The stopped process is then restarted and the debugger waits for the next breakpoint to be reached.

A relative debugger must provide a control mechanism to support this process. Such a control mechanism must be capable of:

- extracting data from one process independent of another;
- handling breakpoints in any order, regardless of the static structure of assertions;
- reusing a data item in many different assertions; and
- managing many processes at once.

In the next chapter, we will describe how we employ a unique dataflow technique to address all these requirements.

CONCLUSION

The technique of relative debugging has been used in many case studies since it was first developed in 1994. It has been shown to be very successful in isolating errors that have been introduced into programs through the porting process or through software evolution. Relative debugging allows the user to focus on key data structures in the code and provides tools to allow the location of errors to be quickly located, often without requiring detailed knowledge of the program. Recent research and development undertaken by the author has extended the use of relative debugging to parallel architectures and to data parallel languages. This research has resulted in the development of a number of novel techniques that provide the fundamental basis for re-engineering the debugger to support parallel computing environments.

Porting a serial program to a parallel computer generally involves transforming the code and data structures of the program in order to take advantage of the architecture, and to ensure that optimum performance is obtained. There are a large number of these transformations available, however in many cases only a small subset of such transformations are commonly used, particularly by parallelising compilers and data parallel language run-time systems. The extension of the relative debugging paradigm to encompass serial and parallel codes necessitates some mechanism to duplicate these transformations so that comparisons between serial and parallel data structures are possible.

This chapter has examined the types of transformations that are commonly applied to code and data structures in the porting process and identified how this impacts on the relative debugging process. It has also looked at a range of issues that must be addressed to effectively implement the relative debugging paradigm.

Chapter 4

THE ARCHITECTURE OF A PARALLEL RELATIVE DEBUGGER

A parallel relative debugger combines the functions of a conventional parallel debugger with those that are specifically required to support the relative debugging paradigm. The debugger must provide the functionality of a conventional parallel debugger for two reasons. First, the user must be able to perform normal debugging operations in the process of determining programming errors. These services could be provided by a separate debugger, but the cost of switching between debuggers makes this approach undesirable. Second, when employing relative debugging (i.e. performing comparisons between programs) the debugger must ensure that multiple processes can be distributed onto different platforms and can be controlled independently of one another. Since this requirement is identical to that of a conventional parallel debugger, it is available at no additional cost or effort.

Extending conventional debugging technology to support the relative debugging paradigm requires that significant additional functionality be provided. This is because the debugger must be able to deal with:

- software engineering issues;
- issues specific to parallel environments, such as parallel architecture and language details; and
- the changes to the code and data structures that occur when programs are parallelised.

The functionality that must be provided by a parallel relative debugger can be loosely grouped into three categories:

- *relative debugging support*, including the evaluation of user-defined assertions, the storage and manipulation of data from concurrently executing processes, and the ability to deal with software engineering transformations;

- *parallel architecture support*, including the ability to interpret the parallel data structures used by data parallel language run-time systems, and functionality to deal with the code and data changes that occur as a result of parallelisation; and

- *parallel process support*, including the ability to control multiple independent processes, support for widely distributed processes, and support for heterogeneous architectures.

The debugger described here, know as GUARD[1], has been specifically designed to address these architectural considerations. As well as providing traditional debugging functions, the debugger includes four key features that address each of the functional requirements of parallel relative debugging. These include:

- a dataflow compiler and engine;

- an architecture independent data format;

- data and code transformation support; and

- a client/server architecture.

In addition, the GUARD debugger employs a modular architecture that incorporates a multi-layered interface design, switchable network protocols and a pluggable debugger backend. All these features are combined into a powerful package to address the needs of parallel software engineers.

This chapter will discuss each of the key architecture features of GUARD in detail, and describe how they meet the functional requirements of a parallel relative debugger.

DATAFLOW COMPILER & ENGINE

The evaluation of user-defined assertions is central to the operation of a relative debugger. However, the ability to define assertions over parallel programs places a significant burden on the matching and control logic of the debugger. In order to support the semantics of parallel assertions the debugger must be capable of:

[1] The latest version of GUARD is known as GUARD-2000. References to GUARD should be assumed to be this version unless otherwise stated.

- extracting data from one process independent of another;
- handling breakpoints in any order, regardless of the static structure of assertions;
- reusing a data item in many different assertions; and
- managing many processes at once.

GUARD meets all of these requirements by employing a dataflow execution mechanism to manage the evaluation of assertions [9,21]. In the dataflow model, a directed graph contains *nodes* that perform certain functions. These nodes are connected together by *edges*, and *tokens* travel from node to node via the edges. Some dataflow architectures may allow only one token to be present on an edge at any time, while others may allow many. These are known as *static* [21,21a] and *dynamic* [9,9a] dataflow architectures respectively, although some implementations combine both methods using a *hybrid* scheme [1a]. Once tokens arrive at a node, they must be stored until at least one token is available on each input edge. In the case of the dynamic architecture a mechanism is required to match the correct tokens on each input, usually with a tag field in the token. This type of mechanism is known as a *matching store*. A modified version of the static architecture, know as *static queued*, allows the queuing of tokens at the nodes [16a]. Once tokens have been matched, they can then be passed as operands to the node for execution.

In GUARD, an assertion statement is represented as a node that performs a comparison only when it has data from two processes available on its inputs, as shown in Figure 4.1. Data from each process is encapsulated in the tokens, which are generated asynchronously as the result of a process reaching a breakpoint. Since there is no guarantee that data from one process will arrive at any particular time, the architecture must ensure that tokens remain available until all the operands for the node have been generated. In addition, it is important that the time each process remains at a breakpoint is kept to a minimum. A static queued dataflow architecture allows almost immediate program restart since multiple tokens can be queued, but does not require the complexity of a matching store. This arrangement also works well if two assertions require the same data item, since a token can simply be duplicated and sent to the appropriate input for each assertion.

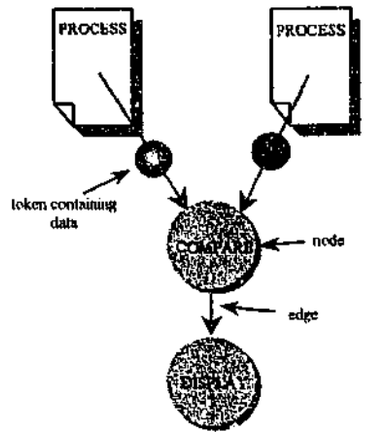


Figure 4.1: Dataflow Graph Executing an Assertion

In a normal debug session, the user will define a number of assertions about the programs being debugged. As part of its user interface, GUARD provides a compiler that translates these assertion definitions into a dataflow graph consisting of nodes that define comparison and control operations, and edges that define the flow of data through the graph. When the assertions are to be evaluated, the dataflow graph is passed to a dataflow engine for execution. To begin the execution process, the graph is "seeded" with an initial token.

Figure 4.2 shows a simplified graph that results from compiling the following series of assertions:

```
assert $S::A[0..49]@seq.c:100 = $P[0]::B@par.c:100
assert $S::A[50..99]@seq.c:100 = $P[1]::B@par.c:100
assert $S::A[0..9]@seq.c:150 = $P[0]::B[0..9]@par.c:90
assert $S::A[50..59]@seq.c:150 = $P[1]::B[0..9]@par.c:95
```

In this example, the array `A[100]` in the sequential code (`seq.c`) has been decomposed across two processes into the array `B[50]` in the parallel code (`par.c`). The syntax uses the notation `$S` to refer to the sequential process and `$P[0]` and `$P[1]` to refer to the two parallel processes. The first two assertions reflect the relationship between the array in the sequential and parallel codes. The second two assertions compare the first 10 elements of the sequential and parallel arrays at different line numbers.

The graph in Figure 4.2 contains a compare node and a display node for each assertion statement. A compare node computes the difference between its arguments. If there is a difference, then this is reported to the user via the display node. The diagram shows tokens

moving along the input edges and arriving at the compare nodes at different times. The first compare node in the diagram has received tokens on both its inputs and has fired (shown in red). The result of the comparison has been sent to the display node, which has also fired, and so has displayed the results to the user.

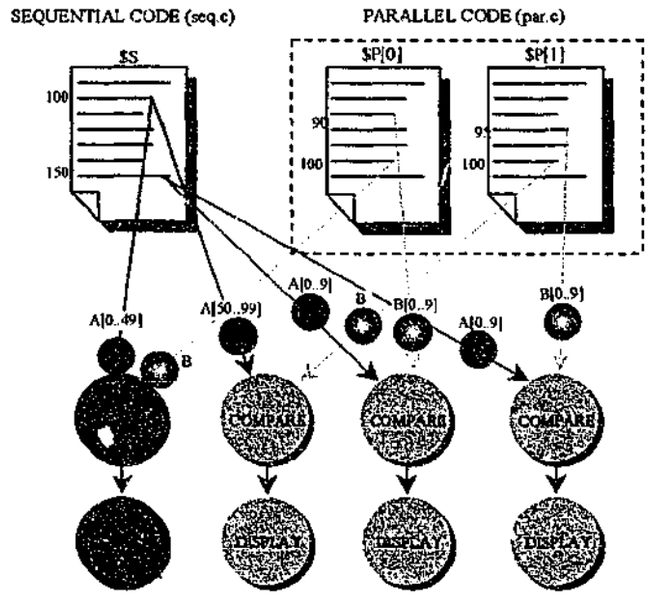


Figure 4.2: Dataflow Graph From Multiple Assertions

A few debuggers have used dataflow as a control mechanism in the past [45,56]. In one of these [56], the dataflow mechanism used a breakpoint as an event trigger for a piece of debugger code. Likewise TotalView [22] allows the user to write expressions that are executed when a breakpoint is reached, and these can be considered like dataflow expressions. GUARD extends this previous work by building complex tokens containing data from the processes being debugged, and then using this data to evaluate the assertion statements.

ARCHITECTURE INDEPENDENT DATA FORMAT

In addition to debugging programs on physically separate hosts, a relative debugger must also support programs running on heterogeneous architectures. Many debugger commands require access to data in the processes being debugged. However, the remote systems executing the target processes (and the debugger host system itself) may each use different architectural features such as word length and byte ordering. Some form of architecture

standardisation must be employed in order to be able to manipulate and compare data from these disparate systems.

The problem of sending architecture specific data over a network has been addressed by standard networking protocols such as XDR [65]. Much work has also been done on the development of architecture independent file formats, with NCSA's HDF [53] generally accepted as the defacto standard. However, none of these approaches address the issues of performing in-memory operations on data from architecturally different systems. The solution adopted by GUARD, know as the architecture independent format (AIF), is designed to achieve true architecture independence for arbitrary data types. The key components of the AIF system are:

- a format for representing data in an architecture independent manner;
- a well-defined application programming interface (API); and
- a set of library routines implementing the AIF API.

The AIF API provides a rich collection of routines that provide support for all common arithmetic, logical, comparison and file I/O operations. In addition, routines are provided to convert data between host specific and AIF formats.

Figure 4.3 shows AIF being used to add two integers from different architectures. First, data is converted into AIF using the `IntToAIF()` routine. The AIF data can then be passed to the `AIFAdd()` routine which calculates the result, also in AIF. This result can then be passed to the `AIFPrint()` routine to be displayed.

The GUARD debugger converts data into AIF as soon as it is extracted from the target process. Debugger commands that manipulate target process data (such as display, arithmetic or comparison commands, or actions generated as a result of the execution of the dataflow engine) operate only on data in this format and so are completely independent of the target architectures. This early conversion of process data to AIF ensures that the debugger remains isolated from the architectural dependencies of its own host system. A clear separation of architecture specific and architecture independent data formats has additional benefits. Defining an AIF API allows routines to be isolated to an independent library, simplifying the debugger implementation and promoting code reuse. The addition of

support for new architectures or language data types also becomes straightforward, as code need only be modified in a single location.

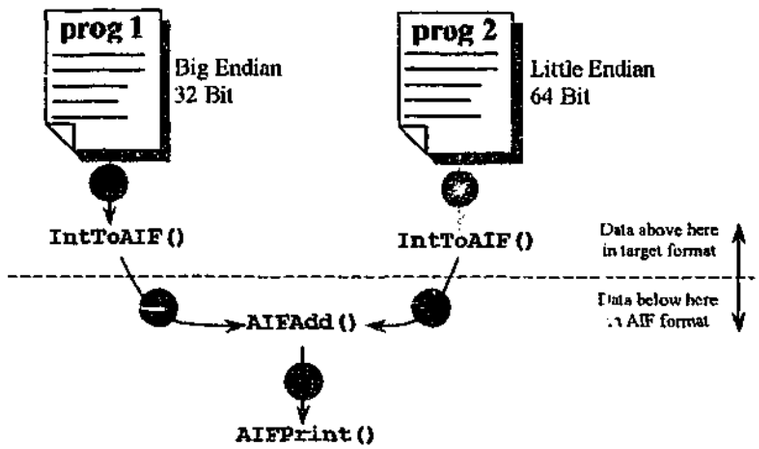


Figure 4.3: Example of AIF Usage

DATA AND CODE TRANSFORMATION SUPPORT

One important feature of relative debugging is that it minimises the detailed knowledge needed by the user when formulating assertions. This allows the user to concentrate on where an error might be occurring rather than on details of the implementation. As discussed in the previous chapter, this technique can become complicated when data or code transformations are employed during the software engineering process, or when sequential codes are parallelised, because the transformations have to be duplicated for the data to be compared. The ability to model these transformations is therefore a key component of the parallel relative debugger architecture. The GUARD debugger provides the capability that allows libraries of common data transformations to be defined (perhaps supplied as a standard transformation library) and then to be easily applied by the user when needed.

There are five main types of transformation supported by the debugger:

- data decomposition;
- shape transformation;
- index permutation;
- array slicing; and

- temporal displacement.

Data transformations are managed using the two commands "map" and "trans", combined with the array slice notation. The debugger handles code transformations resulting from temporal displacement programmatically using the "create" and "assign" commands.

DATA DECOMPOSITION

A parallel mapping function is used to specify the data decomposition technique that has been employed when a sequential code is ported to a parallel architecture. GUARD provides a mechanism to specify such a mapping using the following syntax:

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

Here, *func* is declared as a parallel map. Each map must define an `index` function and a `proc` function. The `index` function is used to specify the relationship between each element of the serial array and the corresponding element of the parallel arrays. The proc function defines the location (in terms of parallel process identifiers) of each element of the array. There are two arguments to the mapping function `func`. These are `P`, which is an *M*-element array of rank *m* of process identifiers, and `D`, which is an *N*-element data array of rank *n* that is located in the address space of each of the processes. A number of predefined functions are also provided to assist in defining mapping functions, including:

`rank(A)`	the rank *n* of array `A`
`nel(A)`	the number of elements *N* in `A`
`upper(A,i)`	the upper bound of index `i` of array `A`
`lower(A,i)`	the lower bound of index `i` of array `A`

When a map is applied to a parallel array, a new single data structure of rank *n* with $N \times M$ elements is created. This resulting array can then be used in an assertion statement as normal. Figure 4.4 shows a typical block-cyclic mapping.

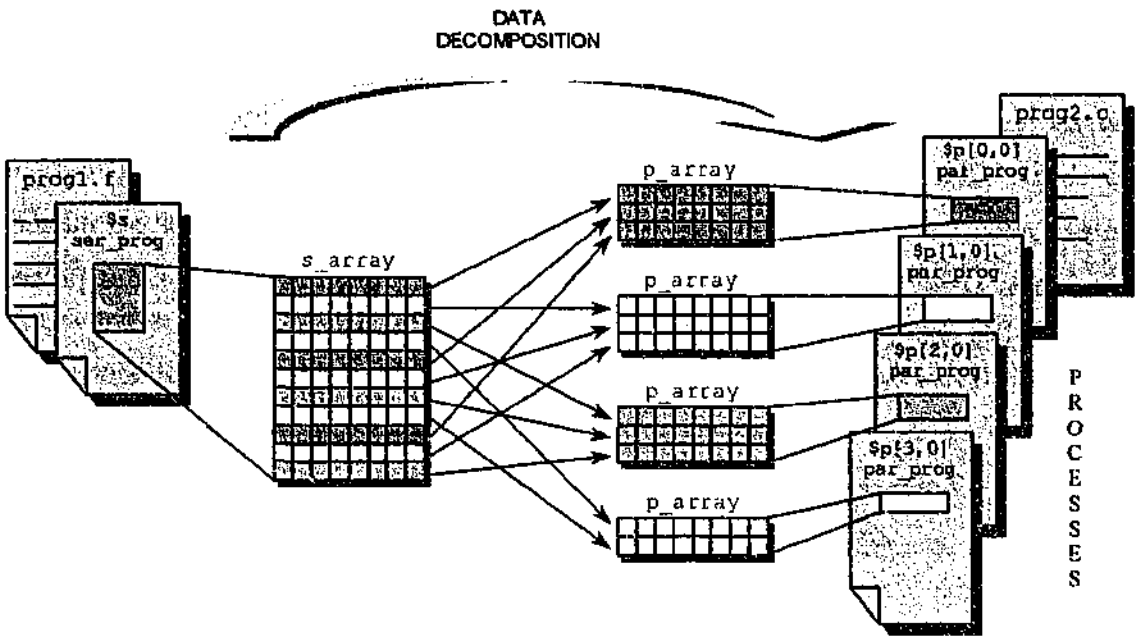


Figure 4.4: Block-Cyclic Decomposition

In this example, the sequential array `s_array` has been decomposed by assigning each row to one of the parallel arrays `p_array`. In GUARD, this would be defined as follows:

```
map bcyc(P::A)
  define index(i,x) = i < rank(A) ? x[i] : (x[i]-1)/nel(P) + 1
  define proc(i,x) = (x[i] - 1) % nel(P) + 1
end
```

Once the mapping function has been defined, it can then be applied to a parallel data structure using an assertion statement as follows:

```
invoke $s "ser_prog"
invoke $p[4,1] "par_prog" using "mpich"
assert $s::s_array@"prog1.f":55 = bcyc($p::p_array@"prog2.c":49)
```

Here, `$s` refers to the serial code `ser_prog` and `$p` refers to a 4×1 process mesh running the parallel code `par_prog`. The assertion statement applies a block-cyclic mapping to the variable `p_array` in `prog2.c` at line 49 of the parallel code. This is then compared with the equivalent variable `s_array` in `prog1.f` at line 55 of the serial code.

SHAPE TRANSFORMATION

Array shape transformation occurs when an array of rank n in a reference program is transformed into an array of rank m in a suspect program, provided the total number of elements in the array are preserved. To compare these arrays using relative debugging, a

mechanism for specifying this transformation must be provided. In order to do this, we conceptualise this process as flattening the array into a 1 dimensional vector, performing an arbitrary permutation on the vector, then blocking the permuted vector to an array of the new shape. The process of flattening and blocking an array can be defined in terms of standard transformation functions, so the user only needs to define the vector permutation operation in order to fully specify a shape transformation. Figure 4.5 shows this process transforming a rank 2 array into a rank 3 array.

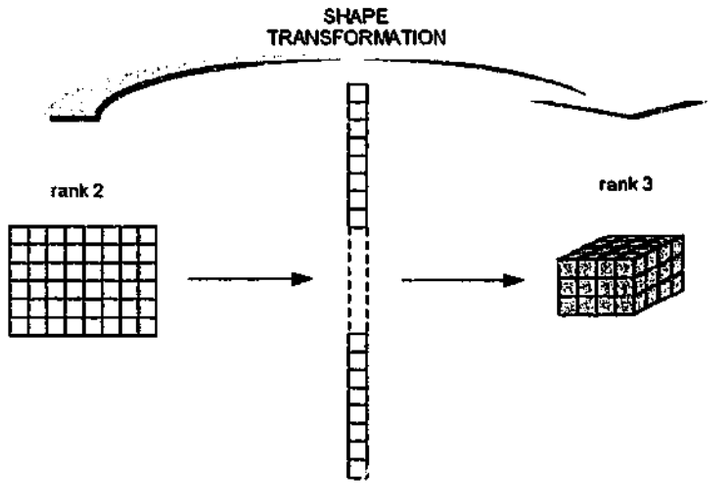


Figure 4.5: Transformation of a Rank 2 Array Into Rank 3 Array

In GUARD, transformations are defined using the following syntax:

```
trans func(A,B)
  define index(i) = expr
end
```

Here *func* is the name of the transformation, and as is the case with all transformations, it must be supplied with two arguments. The first argument A is the array to be transformed. The second argument B is used to specify the shape of the result, and can be either a program variable or an internal debugger variable. The contents of this variable are ignored; only its shape information is used. The result of the transformation is a new array of the same shape as B.

Figure 4.6 shows an example transformation in which a vector of twelve elements is transformed into a 4×3 array, and where each pair of elements is swapped. The following GUARD commands demonstrate how such a transformation can be used.

```
trans swap(A,B)
  define index(i) = (i % 2) == 0 ? i + 1 : i - 1
end


create $a[4][3]
assign $a swap($p::vector, $a)
```

In this example swap defines a transformation that maps the array supplied in the first argument into one of the same shape as the second argument, but also performs a mapping to exchange adjacent elements. The `create` command is used to create a debugger variable to store the resulting array. This command is also used to define the shape of the result, which in this case will be an array of 4×3 elements. The `assign` command is then used to apply the transformation to the program variable `vector` in process `$p` and store the result in the debugger variable `$a`.

Figure 4.6: Shape Transformation with Swap

INDEX PERMUTATION

The permutation of array indices is supported using a special case of the *map* command. In this case a mapping is defined that translates the array indices as required, but leaves the processor information unchanged. Permutations can also be combined with other types of data decompositions to perform complex translation functions.

Figure 4.7 shows a typical permutation where the order of indices is reversed. Such a permutation is commonly used when comparing arrays from Fortran and C programs, since Fortran uses column major order and C uses row major order when storing arrays. A mapping that defines this permutation is shown below, along with an assertion statement that applies the mapping.

```
map f2c(P::A)
  define index(i,x) = x[rank(A) - i - 1]
  define proc(i,x) = i
end

assert $c::Y@"c_prog.c":34 == f2c($f::X@"f_prog.f":45)
```

In this example, we assume `$c` and `$f` are serial C and Fortran codes respectively. The assert statement applies the `f2c` mapping to the Fortran array `X` at line 45 in `f_prog.f`. The result of this mapping will be the same array with indices transposed, which can then be compared directly to the C array `Y` at line 34 of `c_prog.c`.

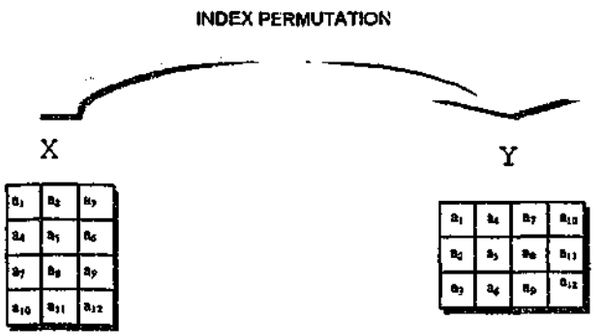


Figure 4.7: Index Permutation Example

ARRAY SLICING

Comparison of sub-arrays is achieved by providing an array slice notation. A sub-array is created from an array by specifying a range of values for each index.

Figure 4.8 shows an example where an array slice operation is required. The arrays `A` and `B` cannot be compared directly because they have different sizes and shapes. Instead, `B` must be compared with a sub-array of `A` using the syntax shown below.

```
assert $p1::A[1..3][1..4]@ "prog1.c":34 == $p2::B@"prog2.c":45
```

In this case the notation `A[1..3][1..4]` is used to specify a sub-array of `A` that contains all elements that have a first index in the range 1 to 3 and a second index in the range 1 to 4 (indices are numbered from 0).

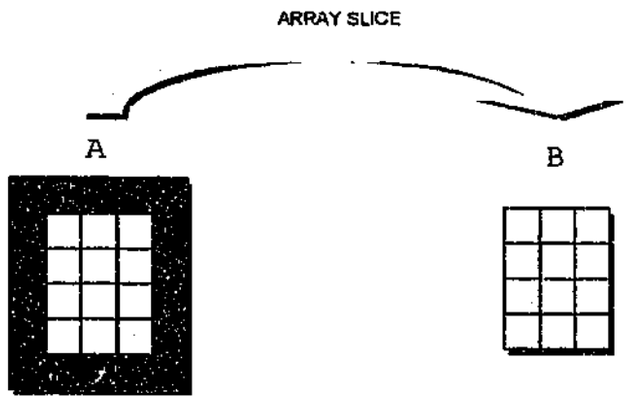


Figure 4.8 Array Slice Example

TEMPORAL DISPLACEMENT

Temporal displacement transformations, such as variable promotion and loop fusion, require the use of temporary arrays in the debugger. A temporary array can be built at run time, and then can be compared with data arrays in a target program, or with other temporary debugger arrays. In the case of variable promotion, a temporary array is used to store the scalar or array that has been promoted to a higher rank. For loop fusion, a temporary array is used to hold the intermediate values of a variable that are overwritten. In both cases, a temporary array is populated by extracting a single value during each loop iteration, as shown in Figure 4.9.

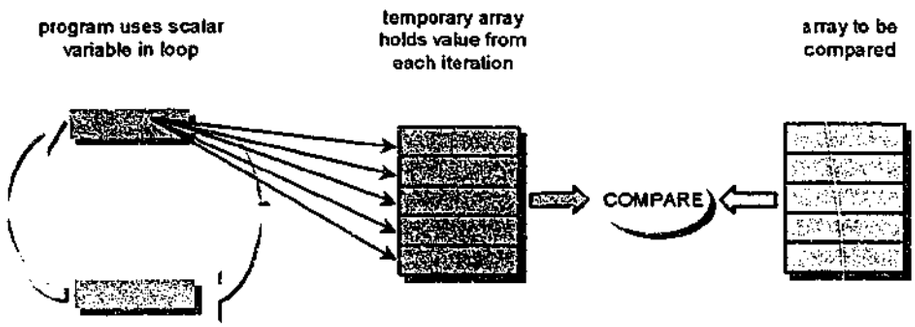


Figure 4.9: Capturing Intermediate Data Values

For the variable promotion example shown in Figure 3.6 in the previous chapter, the following statements would be used:

```
create $I
create $TEMP[100]

assign $I $f::I@"prog.f":80
assign $TEMP[$I] $f::TEMP@"prog.f":80
assert $c::temp@"prog.c":134 == $f::$TEMP@"prog.f":100
```

In this example we assume that `$c` and `$f` refer to the C and Fortran codes respectively. On each iteration of the DO loop in the Fortran code (Figure 3.6(a)), a breakpoint at line 80 will be encountered. The first assign statement will extract the value of the program variable `I` and store it in the debugger variable `$I`. The second assign statement will the extract the value of the scalar variable `TEMP` and store it in the array `$TEMP` at the element corresponding to `$I`. This process will continue until the loop is completed. Meanwhile, the array `temp` from line 134 in the C code (Figure 3.6(b)) will have been obtained, but the assertion will not proceed until the breakpoint at line 100 of the Fortran code has been reached. By this time each element of `$TEMP` will contain the corresponding value of `TEMP` at each iteration of the loop. A comparison of `$TEMP` and `temp` can then take place as usual.

Loop fusion can be handled in a similar way to variable promotion. In the example shown in Figure 3.7 in the previous chapter, a temporary variable must be created to hold the contents of array `x` at line 102 in the parallel code (Figure 3.7(b)). Once all elements of `x` have been transferred, the temporary array can then be compared with the equivalent array in the serial code (Figure 3.7(a)). If we assume that `$s` and `$p` represent the serial and parallel codes respectively, then the comparison can be performed as follows:

```
create $i
create $temp[100]

assign $i $p::i@"pprog.c":104
assign $temp[$i] $p::x[i]@"pprog.c":104
assert $s::x@"sprog.c":131 == $p::$temp@"pprog.c":108
```

Breakpoints are set at lines 104 and 108 of the parallel code (`pprog.c`) and 131 of the serial code (`sprog.c`). The breakpoint at line 104 will be encountered on each iteration of the loop, where the appropriate element of `x` will be copied to the temporary array. The parallel code will then continue until the breakpoint at line 108 is reached. At some stage, line 131 of the serial code will also be reached and the comparison specified by the assertion statement will then take place.

CLIENT/SERVER ARCHITECTURE

GUARD is a multi-process parallel debugger. Most parallel debuggers support the ability to control and manipulate processes running on remote nodes, whether on a tightly coupled shared memory system or in a distributed memory cluster. Parallel relative debugging extends this paradigm further however, since the user may be debugging a combination of serial and parallel codes, all of which are under the control of the debugger at the same time.

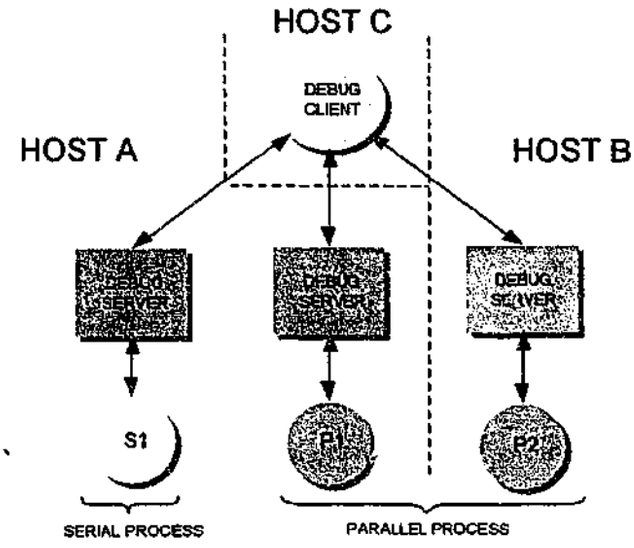


Figure 4.10: GUARD Client/Server Architecture

A number of debuggers and debugging environments have been developed to support parallel and distributed architectures, many of which employ a client/server mechanism [16,22,33,48]. Some, such as DETOP [12] have been developed to support both task and data parallel codes.

GUARD also employs a client/server model in order to ensure that the processes being debugged can be distributed onto multiple platforms and can be controlled independently. In this architecture, each of the debug servers is responsible for managing a single process that is being debugged. The client/server architecture used by GUARD is shown in Figure 4.10. This diagram shows a scenario where the user is debugging a serial process S1 on host A and a parallel program comprising two processes P1 and P2 on hosts A and B respectively. The client/server architecture places no distinction on the host running the debugger client, so in this case it is located on a third machine, host C.

Both the client and server that make up the debugger are designed to be as modular as possible. To achieve this, a three-layer model has been adopted for both the client and the server. Figure 4.11 shows the components that make up the debugger.

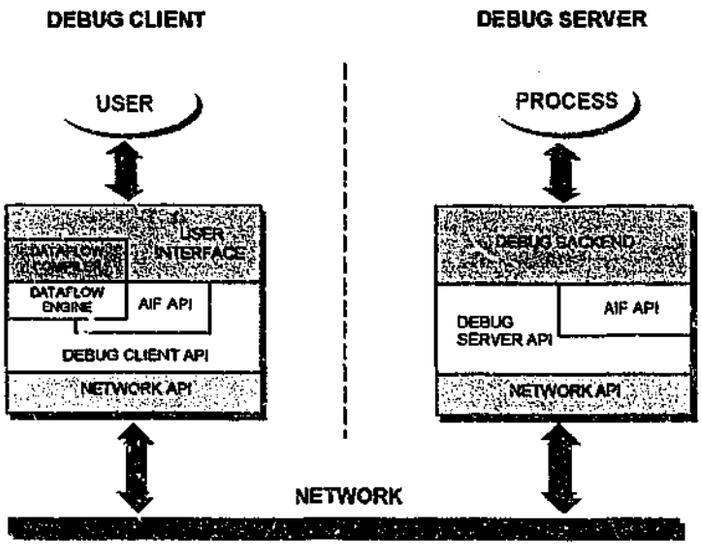


Figure 4.11: GUARD Layered Interface Model

At the top level, the debug client provides a user interface that accepts and processes user commands in either of two modes: immediate or deferred. In immediate mode, the user interface parses and decodes commands and executes them immediately, displaying any results to the user. In deferred mode, the user interface accepts commands that are in turn passed to the dataflow compiler for translation into a dataflow graph. This graph can then be executed at a later date by the dataflow engine.

At the next level, the debug client API layer provides a consistent interface to debugging actions that can be performed on target programs. The debug client API layer manages debug requests from the client regardless of whether they are a result of immediately executed commands, or are generated by the dataflow engine. This layer is responsible for translating these actions into the appropriate network requests using the network API layer, and for receiving and processing responses from the debug server. The AIF API provides an interface for managing and manipulating data in an architecture independent format. All data that is transmitted between client and server is first converted to this format using routines supplied by the API.

At the lowest level the network API is responsible for transmitting and receiving debug requests between the client and the server. A modular interface at this level allows the protocol to be tailored to suit particular requirements.

The server receives requests from the client via the network API layer. These requests are then passed to the debug server API layer, which converts the request into a form suitable for the debug backend. The backend controls the low-level architecture specific functions of the debugger. Maintaining this distinction between the server API and the backend allows for a clean separation between the architecture independent and architecture dependent parts of the server, which in turn ensures that additional architecture support can be easily added to the debugger.

A major consideration in the design of modern debuggers is one of portability. Most recent debuggers are designed to be non-vendor specific, and support as wide a range of computer architectures as possible. Portability has also been an important goal in the design of GUARD, however supporting low-level debug services on multiple architectures requires considerable overhead. Instead, GUARD provides a well-defined interface between the debug server and the low-level debug functions. This allows the services of a pre-existing multi-purpose debugger, such as GDB [66], to be used to provide the low-level functionality. If GDB is not available for a particular architecture, then the well-defined interface also enables other debuggers to be utilised in its place.

The use of a backend debugger also has other benefits. It allows language and language run-time specific details to be separated from the high-level debugger and isolated in the backend. Accordingly, the debug client and server can be designed to be language independent, relying on the backend to interpret the syntax and semantics of individual languages. Support for a new language can then be reduced to a process of modifying the backend to interpret the specific syntactic and semantic details of the language.

Most data parallel languages rely on the language run-time system to manage the distribution of parallel arrays in a manner that is normally hidden from the user. The user need not be concerned about how blocks of data will be decomposed and mapped to processes (although some languages, such as HPF [29] and FORTRAN D [24] do provide mechanisms to specify this). Similarly, it is a design philosophy of GUARD that the user should interact with the debugger in terms of the data structures themselves, without concern for their decomposition and distribution. In order to enable this functionality the debugger must

inherently understand how the parallel decomposition and distribution takes place. By isolating the parallel language support in the debugger backend, the addition of new languages is simplified, and changes required to the debugger are minimised.

CONCLUSION

This chapter has examined the architectural features that distinguish a parallel relative debugger from conventional serial and parallel debuggers. Key innovations that have been developed include:

- the use of dataflow technology to manage the evaluation of user-defined assertions;
- an architecture independent data format addresses the heterogeneous nature of the debugger;
- a framework for defining and managing data decomposition mapping, shape transformations, index permutations and array slicing; and
- a client/server architecture that provides a distributed platform on which the debugger is built.

In addition, this chapter has described some advanced features of the GUARD debugger, including a modular multi-layered architecture, a switchable network protocol subsystem, and a pluggable debugger backend. The result is a debugger architecture that provides a new and innovative technology for parallel software developers.

Chapter 5

DATA TRANSFORMATION ALGEBRA

Many of the transformations necessary for parallelising serial code discussed in Chapter 3 are already implemented by parallelising compilers such as Parafrase [57], automated by pre-processing systems [61], or documented by researchers [72], so are not new in their own right. However for the purposes of relative debugging, these transformations must be specified explicitly because the debugger needs to generate the transformation in order to access and compare data from both programs. Abstract algebra has been used to describe program data structures for some time [26,43,60]. By defining an algebra that describes the possible transformations, we provide a formal basis for extending our debugger to support these techniques.

This chapter focuses on an algebra used to describe four types of array transformations:

- data decomposition;
- shape transformation;
- index permutation; and
- array slicing.

This formalisation does not attempt to duplicate prior work, but rather presents a framework that allows data transformations to be described in terms of an algebraic abstraction. The key advantage of this algebraic abstraction is that it lends itself to interpretation by the debugger, and thus the transformations can be performed automatically. Currently the algebra is only concerned with transformations on arrays, however it is intended that the abstraction be extended to encompass other data structures and transformations in the future.

It is likely that many common transformations can be derived using the proposed algebra and then packaged into macros written in the debugger command language for later use. Thus, the user may not need to specify common transformations, but may simply choose one from a pre-specified library. However, if a new transformation is required, the algebra and

associated command language are sufficiently powerful to allow the new transformation to be specified.

DEFINITION OF NOTATION

The algebra defined in this chapter is concerned with abstractions about two kinds of objects: arrays of data in computer memory and the (physical or logical) processes that these arrays reside on. Before describing the operations that can be performed on such arrays, a notation for representing the arrays and processes will be presented.

ARRAY REPRESENTATION

An array of rank n (i.e. an n-dimensional array) of data held in a computer memory is represented as a set of n-vectors, where each n-vector represents the index values of one cell of the array. Note that the algebra is not concerned with the *contents* of the array cell, but rather with the size, shape and location of the array elements.

> **Definition.** Let $[k]=\{a \mid 1 \le a \le k\}$ be a set of indices. Then the Cartesian product $[k_1]\times\ldots\times[k_n]$ represents the index values of a rank n array (the n-vectors). We represent these n-vectors using the notation $[k_1,\ldots,k_n]=[k_1]\times\ldots\times[k_n]$. We also call this the *shape* of the array. The number of elements in the array is then given by
>
> $$N=\prod_{i=1}^{n} k_i$$

For example, the shape [5] represents a rank 1 array (a vector) containing 5 elements; the shape [4,5] characterises a 4×5 array of rank 2; and the shape [4,5,3] characterises a 4×5×3 array of rank 3. Figure 5.1 shows the three arrays represented using this notation.

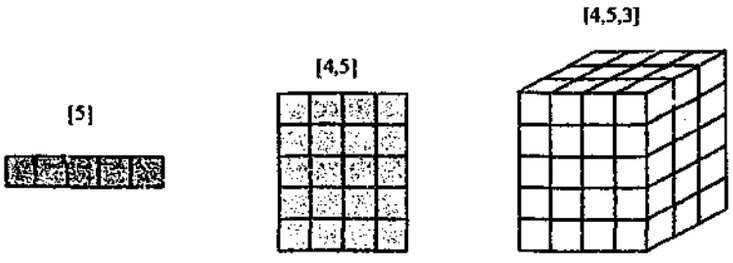


Figure 5.1: Notation for Representing Arrays

In general, array indices used in programming languages can vary between some arbitrary lower bound and upper bound rather than 1 and k. In order to simplify the algebra we need to show that such an array is equivalent to one where the index values are in $[k]$.

> **Proposition 1.** Suppose the indices of a vector (array of rank 1) are elements of the set $A=\{a\in\mathbf{Z}\mid l\le a\le u\}$ for some lower bound l and upper bound u where $l<u\,(l,u\in\mathbf{Z})$. Then A is abstractly identical, or isomorphic, to the array shape $[k]$ where $k=u-l+1$.

Proof. Let $\phi:\mathbf{Z}\to\mathbf{Z}$ define the translation $\phi(x)=x-l+1$. Since $\phi(x)\in\mathbf{Z}\ \forall\ x\in\mathbf{Z}$ we find that:

$$\begin{aligned}\phi(A) &= \{b\in\mathbf{Z}\mid\phi(l)\le b\le\phi(u)\}\\ &= \{b\in\mathbf{Z}\mid 1\le b\le u-l+1\}\\ &= [k] \text{ where } k=u-l+1\end{aligned}$$

Since ϕ is bijective, the shapes A and $\phi(A)$ are isomorphic and hence are abstractly identical.

This result extends to array shapes $[k_1,\ldots,k_n]$ of rank $n>1$ in a straightforward manner.

Example 1. Let $\phi(x_1,x_2)=(x_1-4,x_2-2)$. Then the array shape $\{(x_1,x_2)\mid 5\le x_1\le 8 \text{ and } 3\le x_2<7\}$ is abstractly identical to [3,4] as in Figure 5.2.

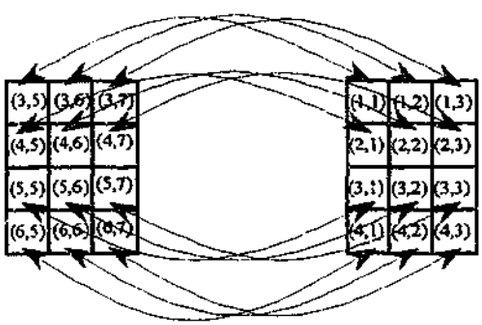


Figure 5.2: Abstractly Identical Arrays

PROCESS REPRESENTATION

In addition to arrays of data, the algebra is also concerned about how data is distributed across multiple processes on a parallel computer. If a particular process topology is described as an m-dimensional mesh then the same notation as used for arrays, can be used

to deal with these process meshes. In this case however, an index value identifies a particular process, rather than referring to a cell containing data.

> **Definition.** A process topology described as an m-dimensional mesh of processes is represented as an array shape $[p_1,\ldots,p_m]$ of rank m. The number of processes in the array is then given by $P=\prod_{i=1}^{m} p_i$

For example, 12 processes could be arranged as either a 3 × 4, 3 × 2 × 2, or 6 × 2 mesh (and other combinations). These correspond to the arrays [3,4], [3,2,2] and [6,2] respectively. Figure 5.3 shows two of these topologies.

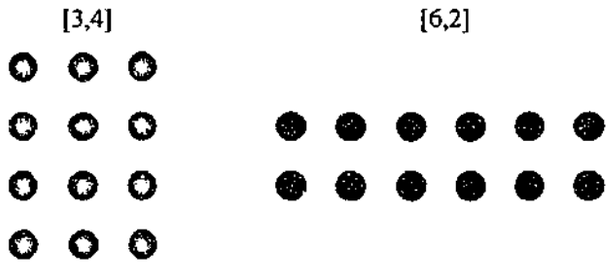


Figure 5.3: Process Topologies as Arrays

DATA DECOMPOSITION

Data parallelisation is concerned with the partitioning and distribution of data. However, in order to use relative debugging with parallel codes, it must be possible to compare equivalent serial and parallel data structures. To describe the distribution of an array onto multiple processes for relative debugging purposes, we define an algebra that specifies how each element of the array is mapped to a partitioned array on each process. Such mappings are oblivious to the array data. They are expressed entirely as a mapping of the array shapes.

> **Definition.** Let $[[k_1,\ldots,k_n],[p_1,\ldots,p_m];\alpha,\rho]$ define a heterogeneous algebra with the two carriers:
>
> $[k_1,\ldots,k_n]$ the shape of an array of rank n, and
>
> $[p_1,\ldots,p_m]$ the shape of a process mesh of rank m,

and the two operations:

$\alpha:[k_1,\ldots,k_n]\rightarrow[k_1,\ldots,k_n]$ the data mapping function, used to specify the location of each element of an array on a particular process, and

$\rho:[k_1,\ldots,k_n]\rightarrow[p_1,\ldots,p_m]$ the process mapping function, used to specify the process that a particular partitioned array is located on.

To illustrate how these functions can be used in practice, examples of block and block-cyclic decomposition are shown below. It should be noted that the equations provided are sufficient for illustrative purposes, but may not necessarily be generalise to higher order arrays. In particular, the equations are only valid when $p_i < \frac{k_i}{2}$ for $1 \le i \le n$.

Example 2. A simple example of block decomposition might be defined as follows. Let $m = n$, then for $1 \le i \le n$ define α and β by:

$$\alpha(x_1,\ldots,x_n)=(a_1,\ldots,a_n) \text{ where } a_i=\begin{cases}(x_i-1)\bmod d_i+1 & | \quad x_i \le d_i(p_i-1)\\ x_i-d_i(p_i-1) & | \quad x_i > d_i(p_i-1)\end{cases} \text{ and } d_i=\left\lfloor\frac{k_i}{p_i}\right\rfloor$$

$$\rho(x_1,\ldots,x_n)=(b_1,\ldots,b_n) \text{ where } b_i=\begin{cases}\left\lfloor\frac{x_i-1}{d_i}\right\rfloor+1 & | \quad x_i \le d_i(p_i-1)\\ p_i & | \quad x_i > d_i(p_i-1)\end{cases} \text{ and } d_i=\left\lfloor\frac{k_i}{p_i}\right\rfloor$$

For example, suppose we have the array [95,95]. For a process topology [3,2], then we have the mapping shown in Figure 5.4.

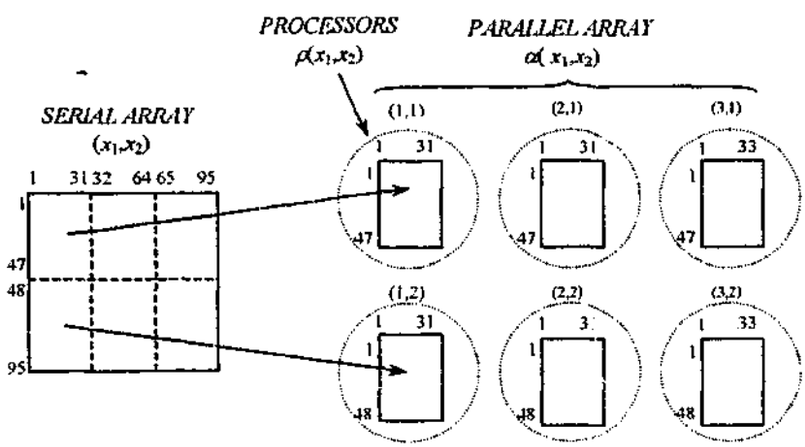


Figure 5.4: Block Decomposition Example

Example 3. As an example of simple block-cyclic decomposition, consider the case where each process is allocated successive rows of an array. For this example we consider the process mesh as a linear array with $m = 1$. We then define the following mapping:

$$\alpha(x_1,\ldots,x_n)=(a_1,\ldots,a_n) \text{ where } a_i=\begin{cases} x_i & | \quad 1\le i\le n-1 \\ \left\lfloor \frac{x_i-1}{P} \right\rfloor+1 & | \quad i=n \end{cases}$$

$$\rho(x_1,\ldots,x_n)=(x_n-1)\bmod P+1$$

Suppose we have an array [95,95] and a process topology [4], then $P=4$ and the decomposition of this array is shown in Figure 5.5.

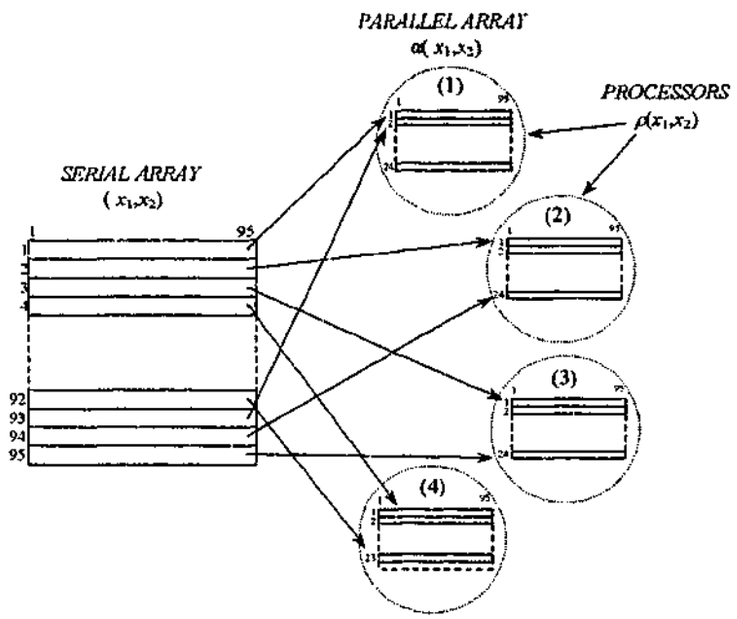


Figure 5.5: Block-Cyclic Decomposition Example

ARRAY SHAPE TRANSFORMATION

In the preceding section we introduced shapes as array index sets and considered simple reblockings of such shapes. In general, more complex shape transformations include permutations of array rows, columns or individual cells. An array shape transformation maps an array shape of rank n into a shape of rank m preserving the number of elements. Intuitively a shape transformation can be thought of as taking a rank n array, linearising it into a 1-dimensional vector, swapping elements in this vector arbitrarily, and finally rebuilding the vector into a rank m array. Each step of the process preserves the size of the array, so the final array remains the same size as the original.

Definition. Let $[\![f_1,\ldots,f_n],[k],[b_1,\ldots,b_m];\phi,\pi,\beta]$ define a heterogeneous algebra with the carriers:

$[f_1,\ldots,f_n]$ the set of indices of an array of rank n,

$[k]$ the set of indices of an array of rank 1, and

$[b_1,\ldots,b_m]$ the set of indices of an array of rank m,

and the three operations:

$\phi:[f_1,\ldots,f_n]\rightarrow[k]$ the "flatten" function and used to linearise the array,

$\pi:[k]\rightarrow[k]$ the "permute" function and used to permute the order of array elements, and

$\beta:[k]\rightarrow[b_1,\ldots,b_m]$ the "block" function and used to transform a flat array into an array of rank m.

We call the bijection $\sigma:[f_1,\ldots,f_n]\rightarrow[b_1,\ldots,b_m]$ where $\sigma=\beta\circ\pi\circ\phi$ a *general shape transformation.* Since σ is a bijection, the number of elements in the arrays, given by $f_1\times\ldots\times f_n=b_1\times\ldots\times b_m$, remains constant.

Example 4. The standard "flatten" function is always used to linearise the array in row major order. This function is defined as follows.

$$\phi(x_1,\ldots,x_n)=\sum_{i=1}^{n}q_i(x_i-1)+1 \quad \text{where} \quad q_i=\begin{cases}\prod_{j=1}^{i-1}f_j & | \quad 2\le i\le n\\ 1 & | \quad i=1\end{cases}$$

To see how the flatten function works, consider the array shape [95,95]. For this array the standard flatten function is:

$$\begin{aligned}\phi(x_1,x_2) &=(x_1-1)+95(x_2-1)+1\\ &=x_1+95x_2-95\end{aligned}$$

The transformation of the array is show in Figure 5.6.

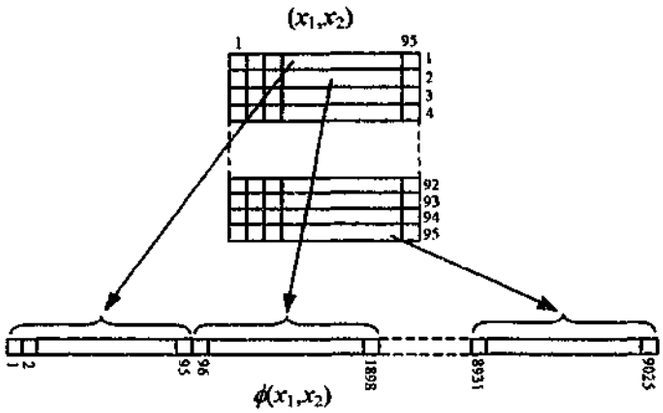


Figure 5.6: Standard Array Flatten Function

Example 5. The standard "block" function is always used to convert a flattened array into an array of rank m in row major order. This function is defined as follows.

$$\beta(x) = (x_1, \ldots, x_m) \quad \text{where} \quad x_i = \left\lfloor \frac{x-1}{r_i} \right\rfloor \bmod b_i + 1$$

$$\text{and} \qquad r_i = \begin{cases} \prod_{j=1}^{i-1} b_j & | \quad 2 \le i \le m \\ 1 & | \quad i = 1 \end{cases}$$

Suppose we now wish to block this array to the shape [361,25]. The standard block function for this array is:

$$\begin{aligned} \beta(x) &= \left(\left\lfloor \frac{x-1}{1} \right\rfloor \bmod 361 + 1, \left\lfloor \frac{x-1}{361} \right\rfloor \bmod 25 + 1 \right) \\ &= \left((x-1) \bmod 361 + 1, \left\lfloor \frac{x-1}{361} \right\rfloor \bmod 25 + 1 \right) \end{aligned}$$

Figure 5.7 shows the result of this transformation.

The composition of ϕ and β, $\beta \circ \phi$ forms a standard transformation from an array of rank n to an array of rank m. This can be considered as a special case of a more general transformation, in which a permutation $\pi : [k] \to [k]$ is applied to the index of the flattened array.

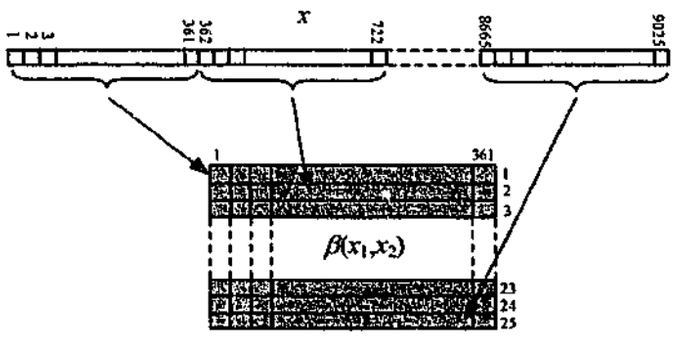


Figure 5.7: Standard Array Block Function

Example 6. To illustrate the use of the general transformation function, consider an array of shape [95,95]. Suppose we wish to transform this into an array of shape [19,25,19], but with each pair of elements swapped. First we check:

$$\begin{aligned}\prod_{i=1}^{2} f_i &= 95\times 95\\ &= 9025\\ &= 19\times 25\times 19\\ &= \prod_{i=1}^{3} b_i\end{aligned}$$

So the arrays contain the same number of elements and the transformation can be applied. Next we find ϕ and β.

$$\begin{aligned}\phi(x_1, x_2) &= (x_1 - 1) + 95(x_2 - 1) + 1\\ &= x_1 + 95x_2 - 95\end{aligned}$$

$$\begin{aligned}\beta(x) &= \left(\left\lfloor\frac{x-1}{1}\right\rfloor \bmod 19 + 1, \left\lfloor\frac{x-1}{19}\right\rfloor \bmod 25 + 1, \left\lfloor\frac{x-1}{475}\right\rfloor \bmod 19 + 1\right)\\ &= \left((x-1) \bmod 19 + 1, \left\lfloor\frac{x-1}{19}\right\rfloor \bmod 25 + 1, \left\lfloor\frac{x-1}{475}\right\rfloor \bmod 19 + 1\right)\end{aligned}$$

Finally, we define the permutation function:

$$\pi(x) = \begin{cases} x-1 & \left\lfloor\frac{x}{2}\right\rfloor > 0\\ x+1 & \left\lfloor\frac{x}{2}\right\rfloor = 0\end{cases}$$

The result of this transformation can be seen in Figure 5.8.

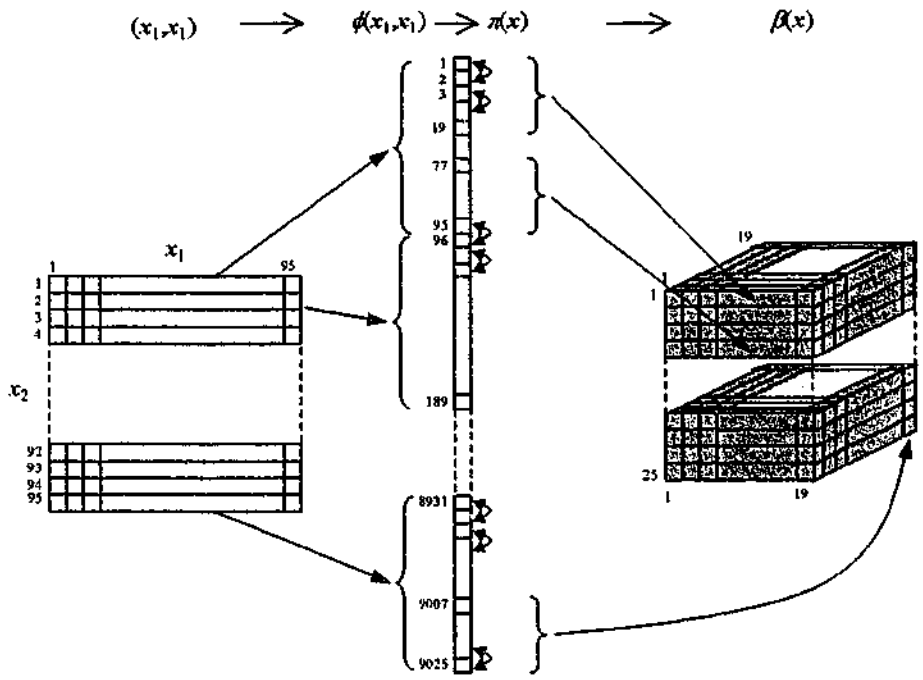


Figure 5.8: Rank 2 to Rank 3 Transformation

INDEX PERMUTATION

Index permutation introduces a very specific requirement: the ability to reverse the order of two or more of an array's indexes. After describing the concepts of data mapping and shape transformation, it should be clear that either could be used to specify index permutation. We choose to use data mapping because it is more intuitive, and more importantly because it allows the useful ability to combine data mapping and permutation in one step.

> **Definition.** Index permutation of an array is a special case of data mapping where $\alpha:[k_1,\ldots,k_n]\rightarrow[k_1,\ldots,k_n]$ is defined so that the appropriate permutation is obtained.

Example 6. An example of an index permutation that swaps each pair of indices is:

$$\alpha(x_1,\ldots,x_n)=(x_2,x_1,\ldots,x_n,x_{n-1})$$

If we are not concerned with process mapping when performing an index permutation, we can define $\rho(x_1,\ldots,x_n)=(1)$, the identity function. The index permutation shown in Figure 5.9 is obtained using the array shape [4,5] with n = 2.

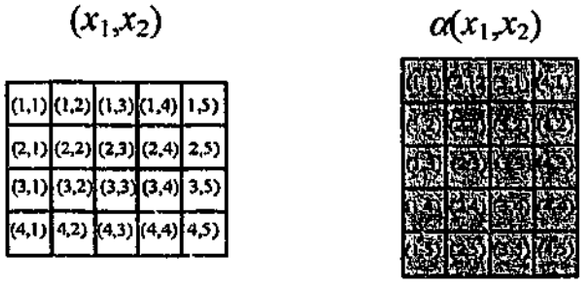


Figure 5.9 Index Permutation Operation

ARRAY SLICING

Sometimes it is necessary to consider a subset of an array. This is achieved using *array slicing*, which extracts a sub-array that preserves the rank but reduces the total number of elements in the array.

> **Definition.** Let $[[k_1,\ldots,k_n],\alpha]$ define a heterogeneous algebra where the carrier $[k_1,\ldots,k_n]$ represents the shape of an array of rank n; and the operation $\alpha:[k_1,\ldots,k_n]\rightarrow[l_1,\ldots,l_n]$ is an array slice function such that $[l_i]\subseteq[k_i]$.

Example 7. Suppose we wish to remove the first and last rows from the array [4,5] as shown in Figure 5.10. In this case we define an array slice function $\alpha(x_1,x_2)=(x_1,x_2-1)\,|\,1<x_2<5$.

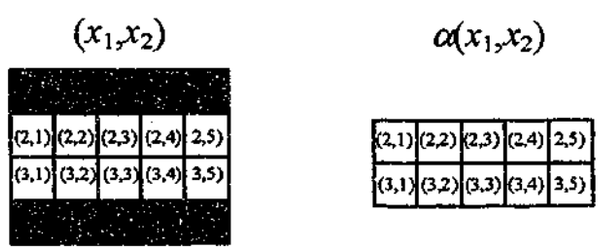


Figure 5.10 Array Slice Operation

Conclusion

This chapter has presented a series of algebraic definitions for transformations that are commonly applied to data structures during the parallelisation process: data decomposition, shape transformation, index permutation and array slicing. The algebra underpins a series of command language features that allow users to compare a data structure in a serial code with its equivalent parallel version using relative debugging. The implementation details of the command language are discussed in the following chapter.

Chapter 6

IMPLEMENTATION DETAILS

As detailed in Chapter 4, a parallel relative debugger must provide a range of functions to allow relative debugging to be employed in a parallel environment. These functions can be categorised as relative debugging support, parallel debugging support and parallel process support. Such a debugger must combine a conventional interactive parallel debugger architecture with the technology necessary to support the relative debugging of parallel and sequential programs. This chapter will describe the specific implementation details that comprise these key technologies, and form the basis of the GUARD debugger.

Most conventional portable interactive debuggers consist of three main components: a user interface (either a command line interpreter (CLI) or graphical user interface (GUI), or both); an architecture independent debugging engine that provides an abstraction of the high-level debug operations; and a set of machine specific low-level debug operations. In the case of interactive debuggers for parallel computers, these are also mostly client/server based. As detailed in the architectural description, GUARD includes all the features of a conventional debugger (apart from a GUI), but also provides a number of additional features to support parallel relative debugging. To achieve this, the GUARD implementation has been designed with a client/server parallel debug engine forming the core infrastructure of the debugger. Built on top of this core is support for the relative debugging technology.

This chapter will examine the specific implementation details of the GUARD debugger. In particular, the following components will be considered in detail:

- the debug client, including the user interface, and the dataflow compiler, engine and internal graph representation;
- the debug server, including the server startup process, interaction between the client and the server, network protocol selection and debug backend selection;
- the client/server debug API;
- the architecture independent data format API;

- support for data and code transformations;
- support for the visualisation of differences; and
- support for data parallel languages, in particular ZPL.

DEBUG CLIENT

The debug client is the primary means of user interaction with the GUARD debugger. The role of the debug client is to provide an interface between the user and the debugging operations that can be performed on one or more target programs. The client is a stand-alone application consisting of the components shown in Figure 6.1.

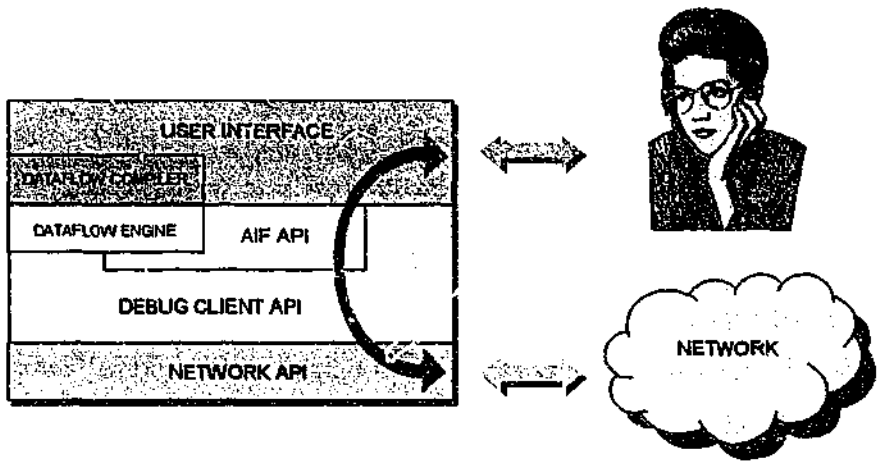


Figure 6.1: Main Components of the GUARD Client

The debug client provides three main services consisting of:

- a user interface that allows users to control programs being debugged, enter commands, and view the results of the commands;
- a dataflow complier that converts user-defined assertions into an internal dataflow graph; and
- a dataflow engine that uses the dataflow graph produced by the compiler to automatically control the execution of the programs and perform the required comparisons.

These services make use of a number of interfaces that are provided in the modular design of the debugger. Both the dataflow engine and the user interface components utilise the services provided by the AIF and debug client API's. These API's provide access to the architecture independent format services and the debug operations that are available using the client/server architecture respectively. The debug client API interfaces to the available network protocols through the network API.

USER INTERFACE

GUARD is primarily a command line driven debugger (see Chapter 8 for details of work developing a GUI). In the UNIX environment, users enter commands via the standard input, which are passed to the command line interpreter (CLI). The CLI parses the command by splitting it into a series of words, each separated by white space. The first word is used to index a lookup table that contains the address of the corresponding command routine. The command routine is called and the remaining words are collected into an array and passed as an argument to the routine. On completion, the command routine returns a status value indicating the success or failure of the operation.

GUARD provides three types of command routines:

- *process targeted,* that perform an operation on a process being debugged (such as setting a breakpoint or single stepping);
- *process optional,* that optionally perform an operation on a process (such as evaluating expressions); and
- *process independent,* that do not interact with a process (such as informational commands).

Process targeted commands are primarily responsible for translating the command and its arguments into a call to the debugger API, then displaying any results to the user. The command may also need to update internal data structures, such as a list of breakpoints, as a result of the call.

Process optional commands are similar in operation to process targeted commands, however they can also perform actions that do not require communication with a target process. An example is the *print* command, which evaluates an expression passed as an argument and

displays the result to the user. Issuing the command "`print 5+3`" does not require any communication with the target process, so the result can be immediately displayed. However, to complete the command "`print a+4`", the debugger must extract the value of the variable a from the target process, then perform the addition and display the result.

Process independent commands do not require any communication with a target process. These commands are generally informational, or are used to modify or display the internal state of the debugger.

Commands are defined by populating a *command lookup table* with the name of the command, the command type, and the address of the command routine. Various support routines are provided to facilitate the addition of new debugger commands. GUARD commands can also support sub-commands by indexing additional command lookup tables with the second and subsequent command words. The two pre-defined commands `info` and `set` support sub-commands by default.

The CLI operates in two modes: *immediate* and *deferred*. In immediate mode, commands are parsed and executed as soon as the user types them. This is the normal mode of operation for the debugger. Some immediate mode commands switch the CLI to deferred mode. When deferred mode is entered, commands are collected and stored internally until the "`end`" command is typed. At this point the debugger processes the deferred commands, then switches back to immediate mode. Commands that utilise deferred mode include "`graph`", "`map`", "`trans`", and "`func`". Appendix A provides a description of all the commands supported by GUARD.

DATAFLOW COMPILER

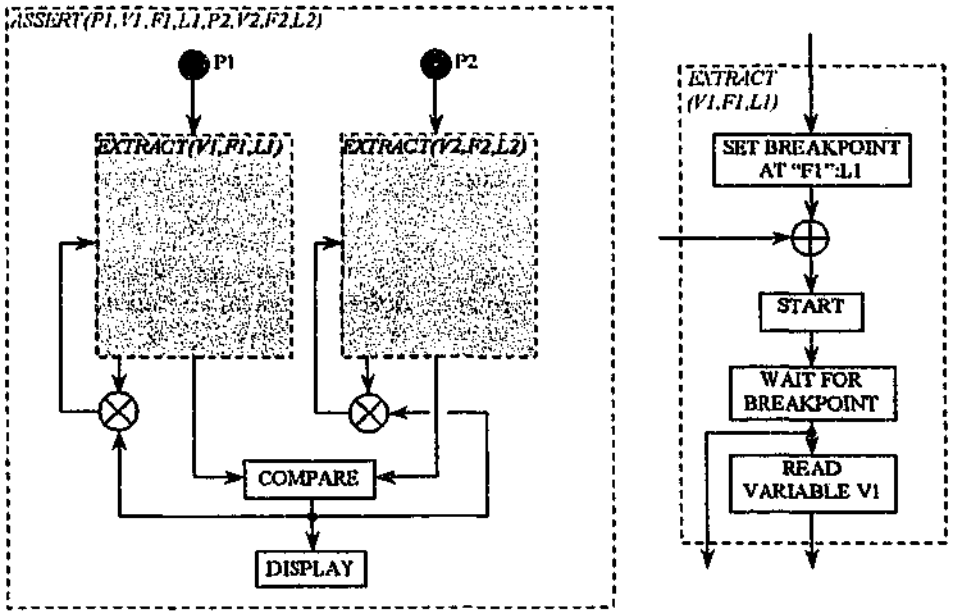
Assertions and other control statements that are collected by the deferred mode command interpreter are passed to a *dataflow compiler*. These assertion and control statements correspond to a low-level graph description. The dataflow compiler translates this graph description into a series of nodes and edges representing the dataflow graph, and stores the resulting graph internally. A simplified version of the graph description syntax is as follows (the full syntax is presented in Appendix A).

```
graph $name [-debug]
  control statement(s)
  assertion statement(s)
  ...
  control statement(s)
  assertion statement(s)
end
```

The "graph" command switches the interpreter to deferred mode, and allows a name to be associated with the resulting dataflow graph. The graph description then consists of a series of control and assertion statements. The control statements are used to modify the behaviour of the assertions that follow them. For example, the control statement "`set error 0.1 0.5 absolute`" could be used to set the lower and upper tolerances and the tolerance type for subsequent assertions. The effect of the control statements is to modify the value of *constant* nodes that are emitted by the compiler, which are discussed in more detail below. The "end" command terminates the graph description.

Compilation is completed in a single pass of the graph description. Single pass compilation places some restrictions on the semantics of the language, by precluding the use of forward references to functions and variables, and requiring that control statements precede the assertions they are intended for. Apart from the usual check of assertion syntax, the compiler also performs some limited type checking, and verifies that process references are valid.

The compiler generates a dataflow graph using a number of standard graph templates. These templates are supplied with information derived from the graph description, and are used to specify the nodes in the graph, and the edges that are used to connect the nodes together. The information passed to the templates is used to specify the values that are generated by special *constant* nodes. These nodes are used to generate information such as file and variable names and line numbers at the appropriate point the execution of the graph. The primary template is the ASSERT template which defines the structure of a dataflow graph used to process a *single* assertion. Figure 6.2(a) shows the basic ASSERT template. For simplicity, boxes in the diagram represent sub-graphs that perform complex operations. The special symbols $\otimes$ and $\oplus$ represent a synchronisation node and a merge node in the graph respectively.



(a) Basic ASSERT Template (b) Basic EXTRACT Template

Figure 6.2: Standard Graph Templates

The ASSERT template takes eight parameters, which correspond to the process name, variable name, file name and line number from each half of an assertion. So, an assertion such as:

```
assert $p1::v1@"file1.c":35 = $p2::v2@"file2.c":55
```

will result in the generation of the ASSERT template:

```
ASSERT($p1, "v1", "file1.c", 35, $p2, "v2", "$file2.c", 55)
```

The ASSERT template comprises two EXTRACT sub-graphs, a COMPARE sub-graph, a DISPLAY sub-graph and a number of nodes to perform synchronisation operations. The EXTRACT sub-graphs are generated from the EXTRACT template shown in Figure 6.2(b). One sub-graph is generated for each half of the assertion statement. The EXTRACT template takes three parameters consisting of a variable name and breakpoint location (file name and line number), specifying the name and location of a variable to be extracted from a process.

The templates in Figure 6.2 are also able to deal with multiple assertions, provided they refer to independent processes or are able to share EXTRACT sub-graphs. In the latter case the

shared sub-graphs are simply merged together and the data sent to the appropriate comparison sub-graph. For example, consider the following two assertions:

```
assert $p1::v1@"file1.c":35 = $p2::v2@"file2.c":55
assert $p3::v1@"file3.c":15 = $p2::v2@"file2.c":55
```

The right hand side of these assertions refers to the same process, variable and breakpoint location. This means that the EXTRACT sub-graphs that are generated can be merged together in the final dataflow graph. Figure 6.3 shows a dataflow graph that represents these two assertions.

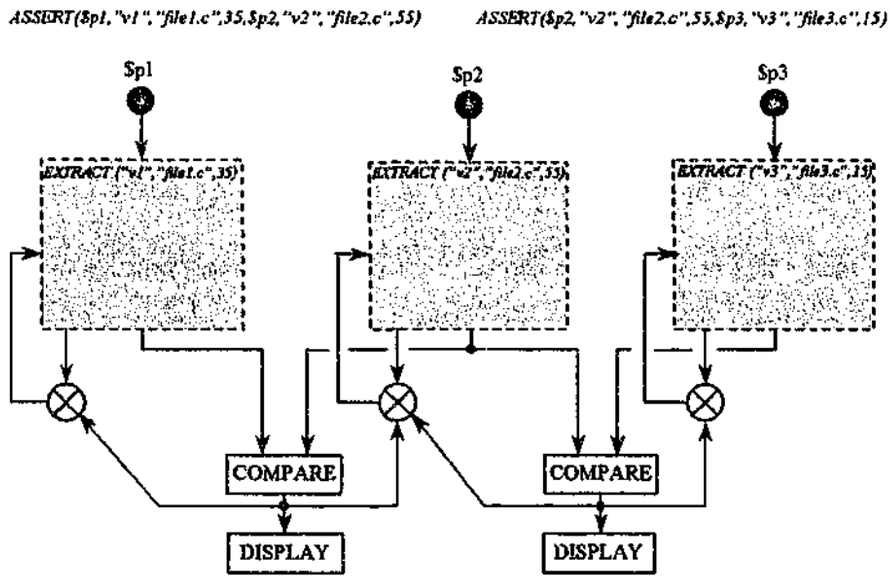


Figure 6.3: Assertions With Merged Sub-graph

While this method works well for simple assertions, problems arise when assertions extract different variables at multiple line numbers from within the same process. In order to overcome this difficulty a more general form of the EXTRACT template must be used. The new EXTRACT template is shown in Figure 6.4.

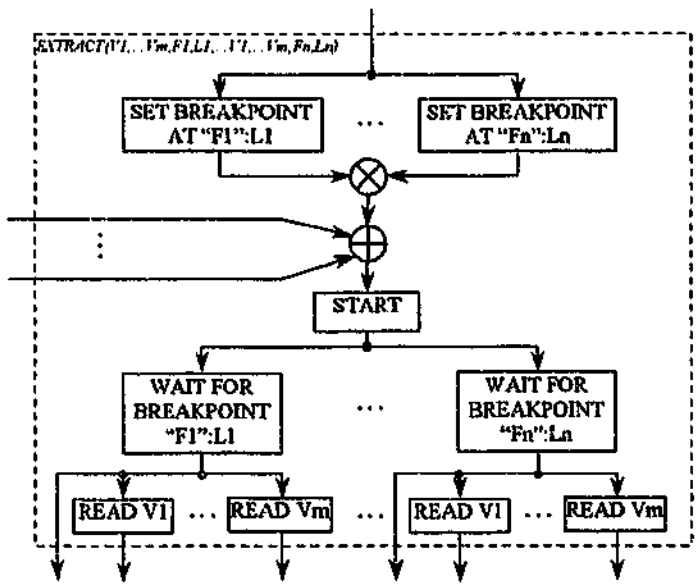


Figure 6.4: General EXTRACT Template

The generalised EXTRACT template allows *m* variables at *n* breakpoint locations to be read from a single process. When this template is used in conjunction with the ASSERT template, any arbitrary set of assertions can be defined.

As an example, consider the following assertions:

```
assert $p1::v1@"file1.c":35 = $p2::v2@"file2.c":55
assert $p1::v1@"file3.c":15 = $p2::v2@"file2.c":55
assert $p1::v2@"file3.c":15 = $p2::v2@"file2.c":40
```

These assertions will generate the following ASSERT templates:

```
ASSERT($p1,"v1","file1.c",35,$p2,"v2","file2.c",55)
ASSERT($p1,"v1","file3.c",15,$p2,"v2","file2.c",55)
ASSERT($p1,"v2","file3.c",15,$p2,"v2","file2.c",44)
```

Since all three assertions share the same processes, the EXTRACT sub-graphs for these assertions will need to be merged. The end result will be a dataflow graph containing the following two EXTRACT templates:

```
EXTRACT("v1", "file1.c", 35, "v1", "v2", "file3.c", 15)
EXTRACT("v2", "file2.c", 55, "v2", "file2.c", 44)
```

The resulting graph is shown in Figure 6.5.

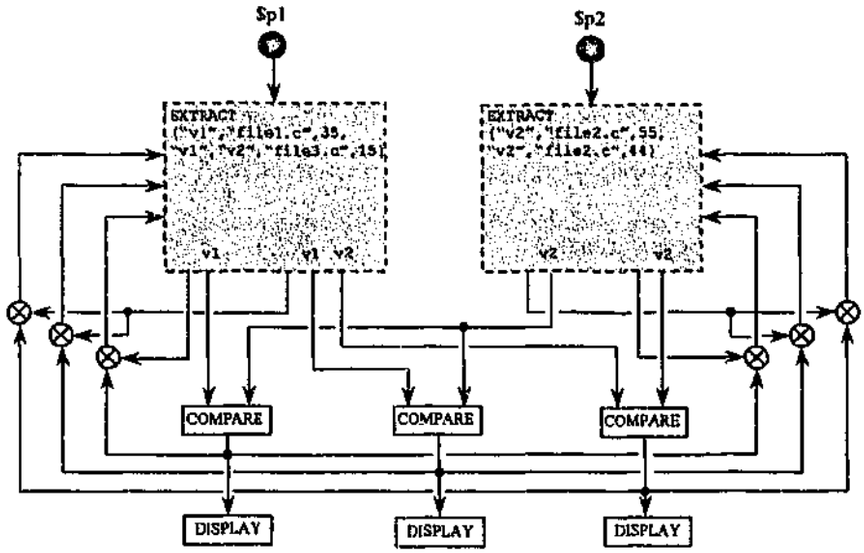


Figure 6.5: General Assertion Graph

The full graph syntax allows the use of general expressions in the specification of assertions. These expressions can contain numeric constants, internal variable references and function references, as well as the usual arithmetic and logical operators. When the compiler encounters such an expression, it is first converted into postfix form. The compiler will then generate a sub-graph that evaluates the expression and insert the sub-graph at the appropriate point in the ASSERT graph. Figure 6.6 shows and example of this translation process.

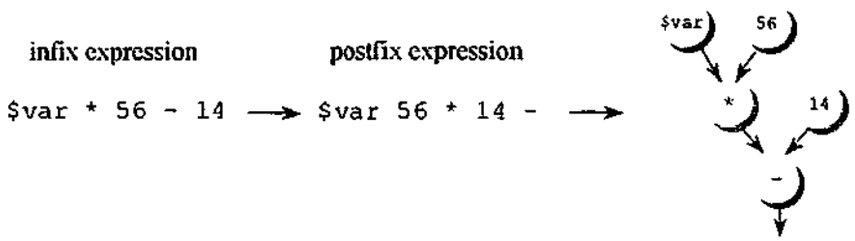


Figure 6.6: Expression Compilation

The graph syntax also supports the use of sub-array operators applied to array data types. The sub-array operator is translated into a sub-graph using a special SLICE node, with the sub-array bounds supplied as inputs to the node. Support for the transformation of array shapes is handled in a similar way, using a special TRANS node. The sub-array and shape transformation sub-graphs are inserted into the ASSERT graph at the appropriate locations by the compiler. Figure 6.7 shows examples of these translations.

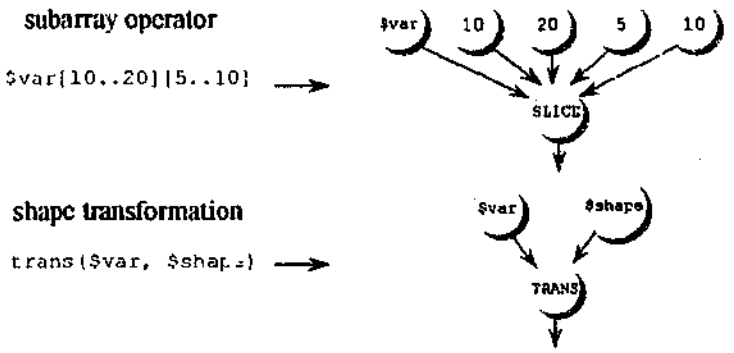


Figure 6.7: Sub-array and Shape Transformation

As described in previous chapters, support for relative debugging of parallel processes is achieved through the use of parallel data mapping. Data maps are implemented in much the same way as shape transformation functions, by using a special MAP node. However, unlike TRANS, the inputs to the MAP node are obtained from data extracted from each parallel process. In practice this means that for *n* parallel processes, there will be *n* EXTRACT templates connected to the inputs of the MAP node. The output of the MAP node is then used as input into the COMPARE sub-graph in the usual way.

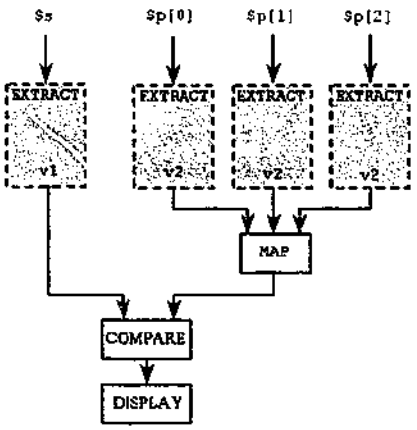


Figure 6.8: Data Mapping Graph

Figure 6.8 shows a simplified graph of the following assertion:

```
assert $s::v1@"file1.c":42 = map($p::v2@"file2.c":99)
```

In this example the data mapping function map is used to compare an array from a serial process with one distributed across number of processes. Here, `$s` represents the serial process and `$p` is used to refer to all of the parallel processes (in this example there are three, `$p[0]`, `$p[1]` and `$p[2]`). The function map, which is unspecified here, is assumed to replicate the data decomposition technique used to model the serial data in the parallel code.

INTERNAL GRAPH REPRESENTATION

The result of the compilation process is a dataflow graph that is stored using an internal graph representation. This internal format is comprised of two main components: a linked list of node objects, and a list of the tokens that are currently active in the graph. Each node object contains a node identifier, information describing the function the node is to perform, and connectivity information. Graph edges are not maintained explicitly. Instead, each node contains two arrays, one holding input objects and the other output objects. These input and output objects provide the connectivity information needed for tokens to traverse the graph. Nodes that have at least one token on an input are also kept on an active list to avoid traversing the entire graph during processing. Figure 6.9 shows this arrangement.

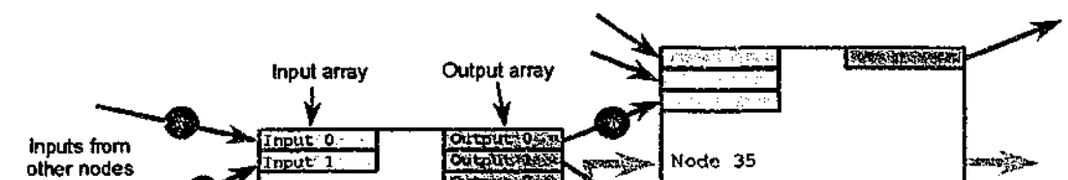


Figure 6.9: Internal Graph Format

The dataflow graph employs a modified static dataflow architecture. Each input object provides a FIFO queue that is able to store multiple tokens. Tokens must arrive at the node in the correct order as the graph matching logic only examines the head of each queue. The input and output objects also employ a form of flow control, so when an input queue is full the output object can be instructed to suspend the generation of new tokens.

The use of a static queued dataflow architecture is an important element of the debugger implementation because it allows the time that a program must wait after reaching a

breakpoint to be minimised. Data can be extracted, stored on an input queue, and then the program restarted almost immediately. The disadvantage with such an architecture is that the graph must be carefully constructed in order to avoid a deadlock situation.

DATAFLOW ENGINE

When the user issues an immediate mode "`start`" command, the graph is passed to the *dataflow engine* for execution. It is the job of the dataflow engine to manage the flow of tokens through a graph generated by the compiler. Since no tokens are present in the graph when it is first executed, a special *start node* must be primed with a single token and placed on the active list. The dataflow engine then checks the active list to see if any nodes are ready to fire. Nodes in the dataflow graph only fire when tokens are present on all their inputs, except in the special case of the merge node, $\oplus$, which fires when a token is present on *any* input. When a node fires a number of actions take place:

- the tokens are removed from the inputs and discarded;
- an output token *may* be generated;
- the node *may* perform some internal operation; and
- the node is removed from the active list.

If the node generates a token, the engine will route the token to the input of any nodes connected to its outputs, and these nodes will be placed on the active list. The active list is again checked for nodes ready to fire, and the entire process repeated. The engine will continue execution until a token is sent to an `EXIT` node, or an error occurs.

The `ASSERT` graph manages the comparison of data from separate processes. It does this by first setting breakpoints at the appropriate locations in each process, and then waiting for a breakpoint to be reached. When a breakpoint is encountered, the graph will extract the data value from the process. All these operations are managed for each process using an `EXTRACT` sub-graph. An `EXTRACT` sub-graph has two inputs: an initialisation input and a synchronisation input. A token containing a reference to a process can be sent to either input. The actions performed by the `EXTRACT` sub-graph are:

1. set one or more breakpoints at the specified lines of the source files;

2. start (or restart) execution of the process;

3. wait for a breakpoint to be reached; and

4. obtain the contents of the variables associated with this breakpoint.

A token received on the initialisation input will perform actions 1 through 4, while token received on the synchronisation input will perform actions 2 through 4. The output from the EXTRACT sub-graph consists of tokens containing the values of each of the variables at the location of the breakpoint that has been reached, and a token that can be used for synchronisation purposes.

The data from each EXTRACT sub-graph is sent to the input of a COMPARE sub-graph that performs the comparison operation. A COMPARE sub-graph has two inputs (one for each EXTRACT) and two outputs: a data output and a synchronisation output. The COMPARE sub-graph checks the value of the comparison against an upper and a lower tolerance value. The result of this check determines what output tokens are generated by the sub-graph, or if the graph execution is terminated. Figure 6.10 shows the possible output combinations for some difference value ε, and upper and lower tolerance values u and l respectively.

Tolerance	Data Output?	Sync Output?	Terminate?
$\varepsilon < l$	no	yes	no
$l \leq \varepsilon \leq u$	yes	yes	no
$\varepsilon > u$	yes	yes	yes

Figure 6.10: COMPARE Sub-Graph Actions

A token generated on the data output of the COMPARE sub-graph is sent to a DISPLAY node to be displayed to the user.

The ASSERT graph must restart execution of the processes it controls at the earliest available opportunity, so that data is continually available for comparison. However it must do so in a way that prevents one process from flooding the graph with tokens. The graph does this by delaying the process restart until the EXTRACT sub-graph has completed *and* the comparison has been performed. Since the comparison is dependent on data values from both EXTRACT sub-graphs being available, this implicitly synchronises the processes and forms a simple flow

control mechanism. The end result of this process will be a steady stream of tokens reaching the inputs of the COMPARE sub-graph.

DEBUG SERVER

User interaction and the evaluation of user-defined assertions in GUARD is the responsibility of the debug client. However the process being debugged might reside on a completely separate machine, possibly with a different architecture. The job of managing and controlling the target process is the responsibility of the debug server. The debug server provides a mechanism for accepting high-level client debug requests using an arbitrary network protocol and mapping these into low-level debug operations on a target process. The server is also responsible for managing and reporting asynchronous events that occur as a result of debugging the target, as well as handling terminal I/O streams between the client and the target process. The main components of the debug server are shown in Figure 6.11.

Client debug requests are received by the server via the network API and are passed to the debug server API layer. This layer decodes the client requests and invokes the appropriate backend routines. The debug backend provides the low-level debug routines that correspond to each of the possible client debug operations. If a backend routine produces a response, then this is passed to the debug server API, which in turn forwards it to the client via the network.

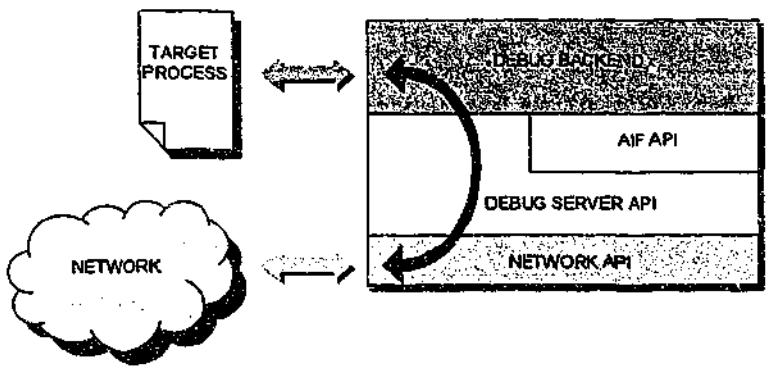


Figure 6.11: Main Components of the GUARD Debug Server

SERVER STARTUP

The mechanism used to start the debug servers is an important issue because it is fundamental to the client/server architecture. The debug server startup mechanism used by GUARD provides the following features:

- the ability to start multiple simultaneous processes for parallel codes;
- notification of server startup failure;
- user selection of network protocol; and
- user selection of debugger backend.

The debug servers are started when the client calls the `DbgInvoke()` debug client API routine. The startup method, the location and the number of debug servers that are started depends on a number of factors, including the location of the target process, whether the target process is serial or parallel and what operating system the target machine uses.

For sequential programs there are three possible methods to start the debug server. These include:

a) the server process is executed directly on the local machine using the `exec()` system call;

b) the Berkeley remote shell command is used to start the server on a remote UNIX system; and

c) the Clustor Dispatcher [7] is used to start the server on a remote NT system.

For parallel programs, the method used to start debug servers in order to debug each parallel process depends on the architecture of the parallel runtime environment. There are currently two mechanisms supported: the *explicit* method and the *wait-attach* method.

The explicit method requires that GUARD start each process via an individual debug server. To do this, GUARD executes a debug server for each parallel process (using either method (a) or (b) above) and passes the program name as an argument. Each debug server is then able to start a parallel process under its control. In order for this method to work, the number of parallel processes must be static and the run-time environment must provide a mechanism for associating a task ID with each process. The explicit startup method is shown in Figure 6.12.

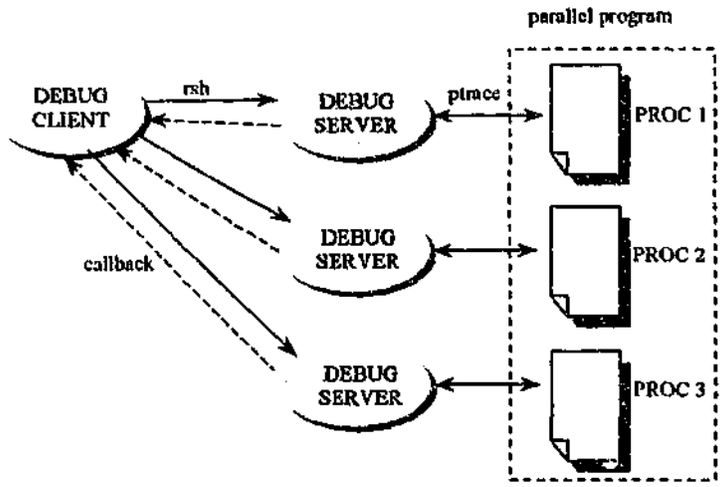


Figure 6.12: Explicit Startup Method

The wait-attach method is available for situations where the parallel run-time system does not provide support for explicit process startup. This method requires that the debugger obtain the UNIX process identifier (PID) for each parallel process, and then attach debug servers to the running processes. In order to support the wait-attach method, the parallel run-time system must know that a parallel program is to be started under debugger control, and also requires that the parallel program use a master/slave arrangement. The run-time system must ensure that each process waits at a well-known location until the debug server has had time to attach. Once the debug servers have attached to all the parallel processes, execution can continue under debugger control. The wait-attach startup method is shown in Figure 6.13.

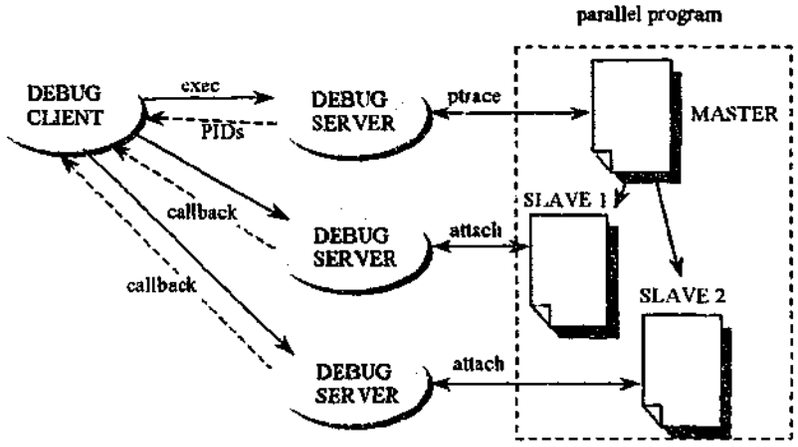


Figure 6.13: Wait-Attach Startup Method

Currently, three parallel run-time systems are supported. These are the distributed memory systems MPICH and IBM's Parallel Environment and the shared memory system P4. Figure 6.14 shows the server startup method for each of these systems, along with the startup command that is used by GUARD to initiate the parallel program.

Parallel System	Startup Method	Start Command
MPICH	explicit	`mpirun`
IBM PE	explicit	`poe`
P4	wait-attach	`rsh/exec`

Figure 6.14: Parallel System Startup Methods

A debug server that is started using any of these invocation methods is passed a number of arguments. The arguments are used to supply essential information that the server needs to know in order to communicate with the client. These arguments are as follows:

-p *proto* the server protocol, such as "rpc" or "socket", which must match the protocol used by the client;

-h *host* the host name or IP address of the client;

-c *callback* a protocol specific communication point: in the case of RPC, an RPC program number; or for sockets, a port number;

-a *arch* the architecture type, specifying a serial process, or one started by MPICH, IBM's PE or P4;

-i *pid* the PID of a process the debugger is to attach to (wait-attach method only);

-n *taskid* the task identifier of this process in a group of parallel processes (wait-attach method only);

-b *backend* the backend debugger type; and

program [*args*] the name of the target process, and any arguments it requires.

When the debug server is started using one of these invocation methods, its first action is to invoke the appropriate debug backend supplying the target process name and arguments. The server will then contact the client using the protocol specific communication point. If

the backend fails for some reason, an appropriate error condition is returned. Provided there are no other problems, the debug server supplies the client with the following information:

- the host name on which the server is executing (since this may not be known the client, particularly for parallel processes);
- a protocol specific communication point;
- for parallel processes, the process task id; and
- a port number used to handle terminal I/O.

If the client invokes a parallel program, then the startup sequence will be repeated for each process that is started, and each server will communicate this information back to the client. Once all the processes have been started, the debugger is ready to begin the debug session. The `DbgInvoke()` routine will not complete until the number of processes that start matches the number of processes requested, or an error occurs.

CLIENT/SERVER OPERATION

Once the debug server startup process is completed, the client is able to issue debug requests to the server using the selected protocol and the negotiated communication point. This process will continue until the client issues a server shutdown request using the `DbgFinish()` debug client API routine.

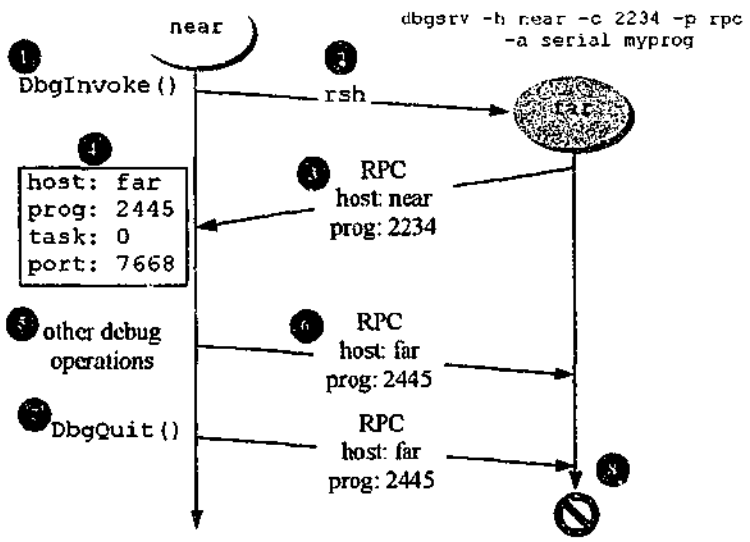


Figure 6.15: Example Client/Server Operation

An example of the client/server operation is shown in Figure 6.15. In this example, the client is running on a host called "near" and the server will be started on a host called "far". Events that occur during the life of the server are numbered consecutively. A description of each event is provided below.

❶ The debugger client issues a `DbgInvoke()` request to start a serial process on host "far" using the RPC protocol.

❷ The client API determines that the remote shell command will be used to start the server, and the server command is issued. The arguments to the command supply a host name of "near" and an RPC program number of 2234. They also indicate the process is serial and that the RPC protocol will be used.

❸ Once the server starts successfully, it issues an RPC callback request using the supplied hostname and program number.

❹ The client obtains the appropriate information from the RPC callback. At this point the startup negotiation is complete.

❺ The client can now issue debug operations to the server.

❻ Debug operations will use the hostname and RPC program number obtained during the startup negotiation.

❼ When the client has finished the debug session, it issues a `DbgQuit()` request.

❽ On receipt of the request, the server terminates.

NETWORK PROTOCOL SELECTION

Part of the startup sequence is used to determine the appropriate network protocol. The actual protocol to be used is selected by the client using the `DbgInit()` debug client API routine as described below. On the server side, the protocol is selected by supplying the protocol name as a command line argument. This is in turn used to index a protocol switch table in a similar manner to the client.

Debug Backend Selection

In order to select a debug backend, the client passes the name of the backend to the server as a command line argument. The server uses this to index a *backend switch table*. This table contains the entry points for the low-level debug routines that must be implemented by the backend. Currently only the "gdb" backend is supported, allowing the use of the GDB debugger [66] to provide the backend services. If GDB is not available on a particular architecture, then this technique allows other backends to be added with relative ease.

Client/Server Debug API

The debug API is central to the operation of GUARD as a client/server debugger. The API provides the mechanism that the client uses to communicate debug commands to the server, and the server uses to send the results of these commands back to the client. The following functionality is provided by the debug API:

- the ability to start and stop debugging processes;
- a set of high level debugging operations that can be performed on a target process;
- the ability to notify the client when debug operations are completed, or when the state of the target process changes;
- the facility to control terminal I/O channels between the target process and the client;
- a switchable low-level debugger interface; and
- a switchable network protocol interface.

The debug API is separated into two parts: a client debug interface and a server debug interface which are discussed in more detail below.

Client Debug Interface

On the client side, the debug API provides a set of routines that implement the high level debug operations. These operations are grouped into the following categories:

- debugger initialisation routines;
- process management routines, such as starting and terminating processes;
- debug operation routines, such as setting breakpoints and traversing stack frames;
- notification routines, such as waiting for a breakpoint to be reached or notification of a signal; and
- status routines, such as obtaining the state of the debugger or displaying error messages.

Appendix B provides a list of the currently implemented routines and a detailed description of each routine. Many of the API routines return pointers to pre-defined data types representing various objects of interest to the debugger. These data types allow access to debugger specific information, such as breakpoint details, call stack frame information and debugger events. A complete list of the pre-defined types is provided in Appendix C.

To begin a debug session, the client first calls the `DbgInit()` routine. This routine must always be called before any other routine so that the client interface can be initialised. This routine is also used to select the required network protocol.

Once the API has been initialised, the client must invoke the program to be debugged using the `DbgInvoke()` routine. The result of this call is an opaque pointer that must be used to identify the target process in subsequent calls to other API routines.

After invoking the target process, the client must arrange to deal with any terminal I/O that the process requires, and to service events that are generated as a result of debugging the process. To do this, callback routines are registered using the `DbgHandleInput()`, `DbgHandleOutput()` and `DbgHandleEvent()` routines.

At this point, the client is ready to issue debug commands using the debug operation and status routines available in the API. On completion of the debugging session, the client calls the `DbgQuit()` routine to terminate the target process and shut down the server. Finally `DbgFinish()` is called to release any internal data structures associated with the client API.

A typical client calling sequence for the debug API is shown below.

```
main()
{
        void *p;

        DbgInit(proto);
        ...
        p = DbgInvoke(program);
        ...
        DbgHandleInput(p, in_callback);
        DbgHandleOutput(p, out_callback);
        DbgHandleEvent(p, ev_callback);
        ...
        /* Call debug operations */
        ...
        DbgQuit(p);
        ...
        DbgFinish();
        exit(0);
}
```

Client/server network protocol selection on the client side is managed by the `DbgInit()` routine. The argument to this routine is a string identifying the protocol to be used. This string is used to look up client protocol routines in a *protocol switch table*. Currently two protocols are defined: "rpc" and "socket". The "rpc" protocol uses Sun Microsystems' Remote Procedure Call (RPC) Protocol [67] for communication between the client and server. The "socket" protocol uses standard Berkeley sockets for communication. The latter is useful for systems that do not support RPC, such as Windows NT.

While the network protocol is established at initialisation time, the client/server API allows a different debug backend to be selected for each target process. The backend is selected by the client which supplies a string argument to the `DbgInvoke()` routine. This information is passed directly to the server as a command line argument. The server then uses this information to select the appropriate backend routines in a *backend switch table*.

The client must be able to handle asynchronous activities generated as a result of the debugging operations. The client API supports this through the use of callback routines. If the process interactively reads or writes to the users terminal, this needs to be intercepted and managed via the debugger user interface. In addition, the process management routines, such as `DbgGo()` and `DbgStep()` do not return a value immediately, but rather initiate an operation that may complete at some future time. In order to receive notification of the completion of these operations, the client must register an event handling callback routine.

The use of asynchronous callbacks places some restrictions on the nature of the client. Callbacks are particularly suited to graphical environments such as X-Windows, since these environments generally operate on an asynchronous event model. The use of callbacks is less common in command line environments, since it precludes a main command loop from blocking at any stage. In order to facilitate the use of the API in command line environments, a special event handling routine is provided. The `WaitForEvents()` routine can be used to block until an event is received or input is available on the command line. By using this routine, the main loop of a command line interpreter can be simplified significantly. A typical main loop is shown below.

```
while ( !finished )
{
        DisplayPrompt();
        ...
        WaitForEvents(fileno(stdin));
        ...
        ReadAndExecuteCommand();
}
```

SERVER DEBUG INTERFACE

On the server side, the debug API is responsible for communicating client debug requests to the backend debugger and for returning results to the client. The server API is also responsible for managing asynchronous events and terminal I/O streams. There are two main components to the server interface:

- a series of server stub routines that interface to the appropriate backend services; and
- utility routines that are used by the debug backend to process debug requests.

Client debug requests are received via the network interface where they are passed to the debug server API layer. This layer decodes the request and its associated arguments and invokes the appropriate server stub routine. The server stub routines are associated with corresponding backend routines using the backend switch table. This results in the appropriate backend routine being invoked to perform the requested operation. Once the backend routine has completed its operation, an event will be generated and returned to the client. This event be delivered either synchronously or asynchronously, and can contain result or status information. Figure 6.16 shows this process in operation.

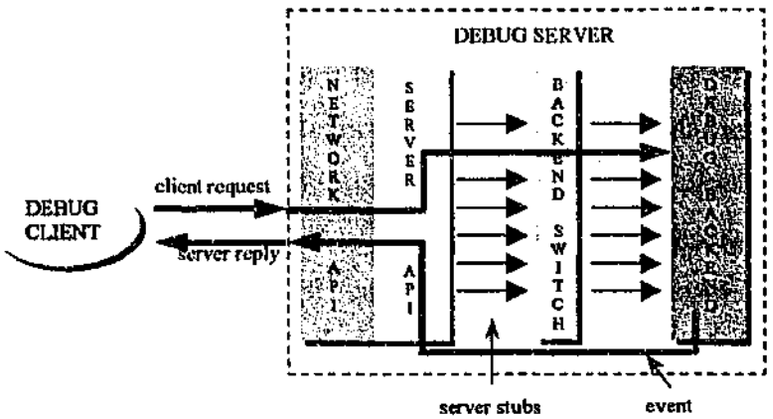


Figure 6.16: Debug Server API and Backend Switch Table

In addition to the stub routines, the debug server API provides a number of utility routines for use by the debug backend. These include routines that provide the following services:

- event creation and handling;
- debugger status management;
- error handling; and
- asynchronous task processing.

Details of these routines are provided in Appendix D.

ARCHITECTURE INDEPENDENT FORMAT

AIF is designed to achieve true architecture independence for arbitrary data types. The key components of the AIF system are:

- a format for representing data in an architecture independent manner;
- routines for converting to and from native formats;
- routines for performing arithmetic, logical and comparison operations on AIF data;
- routines for accessing components of structured data types; and
- routines for performing I/O on AIF data.

Data types are represented in AIF using a data tag and a normalised bit stream. When data is converted to AIF, a series of normalisation operations are performed. These operations include the conversion of integers to big-endian byte ordering, characters to the ASCII collating sequence and floating-point number to the IEEE 754-1985 big-endian format [31]. The format also standardises on a byte size of 8-bits and single byte characters. Since the normalisation operations can result in the loss of native format information, the bit stream is tagged with a format descriptor string (FDS). Figure 6.17 shows the current format descriptor tags.

Tag	Type	Details
`c`	character	
`i`*s*/	integer	*s* is `s` (signed) or `u` (unsigned), */* is size in bytes
`f`*l*	floating-point	*/* is size in bytes
`^`*ll*	address	type *t*, */* is size of the address in bytes
`[`*t1*`]`*t2*	array	type *t2* with index type *t1*, which must be a range
`{`*l*`=`f_1`@`o_1`#`l_1`:`t_1`,` ... `,`f_n`@`o_n`#`l_n`:`t_n`}`	structure or union	*/* is size in bytes, f_i is the name of the field, o_i is the offset in bits from structure's origin, l_i is the size in bits, and t_i is type of the field
`<`e_1`=`v_1`,` ... `,`e_n`=`v_n`>`	enumeration	e_i is the name of each value v_i.
`x`v_{min}`..`v_{max}`/`	range	based on integral type *t* with v_{min} and v_{max} as limits
`V`*l*	void	*/* is size in bytes
`R`*rt*	ZPL region	rank *r*, whose limits are base on integral type *t*

Figure 6.17: AIF Format Descriptor Tags

Tags for simple data types provide AIF library routines with a type identifier, a field indicating if the type is signed or unsigned, and the length of the data. This allows different integer sizes and single, double, and extended floating point formats to be recognised, and also ensures that the AIF library routines can perform calculations with no loss of precision. Complex data types have tags that describe the size and memory layout of the data, and can contain nested tag types.

Figure 6.18 shows an example of how a C structure is represented in AIF. In this example, the format descriptor string is used to describe the layout of a structure. The FDS starts with "`{16=`" indicating that the accompanying data represents a structure that is 16 bytes in total length. The next two fields separated by "`,`" describe the layout of the fields in the structure. The string "`a@0#32:is4`" defines the name of the first field as "`a`", along with its starting position and length (0 and 32 bits respectively), and a type of "`is4`" indicating a 4 byte signed integer. The string "`b@32#72:[r0..2is4]f4`" defines the name of the second field as "`b`" and indicates that the field starts at bit position 32 and is 72 bits in length. The

type of this field is given as "`[r0..2is4]f4`". This is a complex type that defines an array. The string "`r0..2is4`" specifies a range consisting of 3 elements of type "`is4`". When applied to an array, this specifies the number of elements in the array. The type of each element is defined as "`f4`" or a 4 byte floating point number.

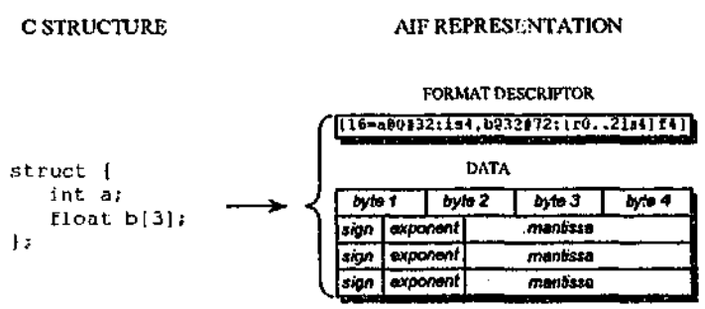


Figure 6.18: AIF Representation of a C Structure

Data is converted into AIF using one of the data conversion routines. Routines are provided to convert simple data types, such as `IntToAIF()` and `FloatToAIF()` and complex data types, such as `ArrayToAIF()` and `StructToAIF()`. All the data conversion routines return a pointer to an `AIF` structure that holds the data and tag information. This pointer can then be used as an argument to the other AIF routines. Routines are also provided convert AIF data back to native format.

Once data is converted to AIF it can then be manipulated using the arithmetic and logical operations. The code below shows an example using the AIF API to multiply two integers.

```
main()
{
        int   i = 45;
        int   j = 33;
        int   res;
        AIF *ai;
        AIF *aj;
        AIF *ar;

        ai = IntToAIF(45);
        aj = IntToAIF(33);
        ar = AIFMul(ai, aj);
        res = AIFToInt(ar);

        printf("the result is %d\n", res);

        exit(0);
}
```

In this example, the `IntToAIF()` routine is used to convert the integers into AIF. The `AIFMul()` routine then multiplies the values and returns the result, again in AIF. This can then be converted back to the native format for display.

Routines are also provided to access and manipulate complex data types, such as indexing elements of arrays, iterating over array indices or retrieving structure fields. In addition, routines are available to perform I/O on AIF objects, such as reading from and writing to files. A full list of the AIF routines is given in Appendix E.

The current implementation provides descriptors for C, Fortran, and ZPL [42]. Additional languages can be employed by defining new descriptors for each data type that is not supported by the current descriptors, and by providing routines to perform appropriate operations on these data types.

DATA AND CODE TRANSFORMATIONS

In previous chapters, we have discussed the types of transformations that are often applied to data structures and code when translating programs from serial to parallel architectures, and have presented an algebra for describing the array data transformations. The data decomposition, shape transformation, index permutation and array slicing transformations are implemented in the debugger using a new command language that allows libraries of common transformations to be developed. A mechanism for the creation and population of temporary arrays has also been added to the debugger command language in order to provide support for temporal displacement techniques such as variable promotion and loop fusion. The following section discusses these features in more detail.

DATA PARALLEL DECOMPOSITION AND INDEX PERMUTATION

As described in Chapter 4, the "map" command is used to specify how serial arrays are decomposed when a code is ported to a parallel architecture and when performing permutations of array indices. When a map is applied to a series of data structures that comprise the parallel array, a new single data structure is created. For the purposes of this discussion, we denote this resulting array by `R`, although it normally is not explicitly named.

Given a "`proc`" and "`index`" function defined by the mapping, an array of process identifiers `P` and a parallel array `A`, the following pseudo-code shows how GUARD will compute the elements of the resulting array:

for (each index $(x_1,\ldots,x_n)$ of the result array R)
  compute $(p_1,\ldots,p_m)$ **where** p_j = proc(j, $(x_1,\ldots,x_n)$) **for** j=1..m
  compute $(a_1,\ldots,a_n)$ **where** a_j = index(i, $(x_1,\ldots,x_n)$) **for** i=1..n
  set R[$x_1,\ldots,x_n$] = P[$p_1,\ldots,p_m$] : : A[$a_1,\ldots,a_n$] [1]
endfor

Note that the index and proc functions are actually supplied with two arguments. As shown above, the first argument is the index number. The second argument is an array representing the values of the indices of the current element of the result array R during the computation. This information ensures that complex mapping functions can be defined.

SHAPE TRANSFORMATION

Array shape transformation is specified using the "trans" command. This is the process of flattening the array into a 1 dimensional vector, performing an arbitrary mapping on the vector, then blocking the mapped vector to an array of the new shape. Since flattening and blocking of arrays uses the standard transformation functions ϕ and β, the user only needs to define the permutation function π (called "index" when defining the transformation) in order to specify a shape transformation.

Given a permutation function π, an array A and the shape of an array B, the result of the transformation is a new array which we denote as R. The following pseudo-code shows how the result array is computed:

for (each index $(x_1,\ldots,x_n)$ of the array A)
  compute $i = \phi(x_1,\ldots,x_n)$
  set F[i] = A[$x_1,\ldots,x_n$]
endfor

for (each index i of the array F)
  compute j = index(i)
  set F1[j] = F[i]
endfor

for (each index i of the array F1)
  compute $(y_1,\ldots,y_n) = \beta(i)$
  set R[$y_1,\ldots,y_n$] = F1[i]
endfor

The first loop computes the flattened vector F. This is done by applying the flatten function ϕ to the indices of array A to calculate a new index value i. The corresponding value of A is

[1] The notation P[$p_1,\ldots,p_m$] : : A[$a_1,\ldots,a_n$] means the $(a_1,\ldots,a_n)$ element of array A on process $(p_1,\ldots,p_m)$.

then stored in F at the location specified by *i*. The next loop applies the permute function index to each element of F to create a new vector F1. The final loop then blocks the F1 vector to the resulting array R.

VARIABLE PROMOTION AND LOOP FUSION

Support for variable promotion and loop fusion involves extracting data from within an executing program and storing the values as internal variables in the debugger. The debugger syntax then allows these internal variables to be compared with data from target programs, or with other temporary debugger variables.

The implementation of internal debugger variables uses the same implementation techniques as that used for storing data from target programs, and involves converting the data to the architecture independent data format, AIF. The AIF API provides all the support necessary for managing these internal variables. GUARD provides the user with a "`create`" command that can be used to instantiate scalars, or arrays of a specified size and shape. These data structures are initially "empty", but can be subsequently populated with data from a target program using the "`assign`" command.

VISUALISATION OF DIFFERENCES

The visualisation of differences that have been obtained by the execution of user-defined assertions underpins the relative debugging paradigm. Currently the primary means of interaction with the debugger is through a command line user interface. In order to support this visualisation, 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 or VIS-5D [27a]. In addition, a number of tools are provided to assist with this visualisation process.

When defining assertions in GUARD, the user has the ability to select a particular display type using the "`set display`" command. This command is used to specify the format that is used when the difference information is generated. The difference information can also be stored in an output file using the "`set output`" command. The available display types are shown in Figure 6.19.

Display	Description
`text`	Plain text is used to display differences directly
`hdf`	Differences will be output using the NCSA Hierarchical Data Format (HDF) [53]
`hdf5`	Differences will be output using the later HDF Version 5
`aif`	Differences will be output in the Architecture Independent Format (AIF)

Figure 6.19: GUARD Display Types

In situations where assertions generate multiple results, such as when an assertion is made within a program loop, it is useful to store difference information as consecutive data sets in a single file. Both the HDF and AIF file formats support this, and the visualisation software is able to use this information to display the differences as a series of frames in a movie.

In situations where simple 2-dimensional arrays are being compared, GUARD provides utility programs to convert the data files into GIF's for display using a web browser. If the data files include multiple data sets, then the utilities can also be used to generate animated GIF's to display simple movies. Utilities are also provided to analyse and manipulate data that has been generated from the assertions. The utility programs are shown in Figure 6.20.

Utility Program	Description
`hdf2gif`	Convert a file in HDF to an animated GIF
`aif2gif`	Convert an AIF file into an animated GIF
`hdfanalyse`	Display information about a file in HDF
`dumpset`	Convert an AIF data set file to text
`listset`	List data sets in an AIF file

Figure 6.20: Utility Programs

DATA PARALLEL LANGUAGE SUPPORT

A key component of the GUARD debugger is its ability to support data parallel languages. The current implementation provides support for the relatively new, but efficient and easy to use data parallel language ZPL [42]. As will be discussed later, this support can also be extended to other languages.

ZPL is a data parallel array-based language. The language uses the array as a fundamental data type, and provides various features that allow programmers to generalise algorithms using array semantics. The most fundamental of these is the *region*, which is used to specify a set of indices that define the bounds of an array, or over which computations on an array are performed. ZPL has been designed as a machine-independent language that is portable across multiple architectures and memory paradigms. Although it is primarily designed as a

data parallel language, ZPL executes on both sequential and parallel architectures. ZPL is also an implicitly parallel language, as the programmer does not need to explicitly specify how parallel computations are to take place, rather the compiler determines the distribution of data automatically. Parallelism is derived from the semantics of the array operations, so there are no parallel directives or mechanisms for explicit message passing. Programmers see a single address space and are able to utilise traditional sequential programming semantics for software development.

From an implementation perspective, support for a data parallel language requires the following capabilities:

- knowledge of the specific language features;
- the ability to extract locally held components of a data structure from each process; and
- the ability to recombine these components into a single data structure in the debugger.

Data parallel language run time systems generally maintain per-process data structures that contain a description of each parallel array, and provide information such as the rank of the array, the size of each dimension, and information on the distribution of the array across the processes. In order to access a block of data from an individual process, the debugger must first access this description information to determine the location and bounds of the block that is resident in the process.

In ZPL, this distribution information is stored in run-time structure called an *ensemble*, one of which is maintained for each parallel array in each process. An ensemble consists of the following (simplified) structure:

```
struct ensemble {
    int blocksize[MAXRANK];
    int offset[MAXRANK];
    int stride[MAXRANK];
    void *data;
    int numdims;
    region *regptr;
    unsigned long size;
    char *basetype;
};
```

In order to access the data specific to a particular process, the debug server for that process must first extract the upper and lower bounds for each index of the array from `regptr`, a pointer to the array's region information. The location of a particular element of the array is then computed from the `offset`, `stride` and `blocksize` information. For some element in the array whose indices are $(a_0, a_1, \ldots, a_{n-1})$, where n is the number of dimensions specified by `numdims`, the location of the data is given by:

$$\texttt{data} + \sum_{i=0}^{\texttt{numdims}-1} \left(\frac{(a_i - \texttt{offset}[i])}{\texttt{stride}[i]} \times \texttt{blocksize}[i] \right)$$

This information is interpreted by the debug server and used to obtain a copy of the data that is specific to the particular process.

To allow data parallel arrays to be used in assertions in a transparent manner, GUARD utilises the decomposition information along with the per-process location information to reassemble a complete structure. Using the appropriate mapping function GUARD is able to apply the language decomposition rules to request the components of the array from each process via the debug servers, and then reassemble these components into a complete array. Assertions can then be used to compare the data in this array with that obtained from a reference program.

Adding support for other data parallel languages to the debugger is a three-step process. First, the syntax and semantics of the language must be defined in the debugger parser. However, since the debugger only allows immediate evaluation of expressions, the full language syntax does not need to be defined. Second, the parallel data distribution information must be made accessible to the debugger backend. Finally, AIF tags may need to be added to support any new data types introduced by the language.

For example, to add support for the data parallel language C* [59] to GUARD, new language support would need to be included in the parser. C* provides additional syntax to allow scalar access of parallel arrays, adds a number of new operators such as minimum and maximum value reduction ("<?=" and ">?="), and introduces a new type syntax for shape declarations. Next, the run-time representation of parallel variables would need to be added to allow the debugger backend to access the parallel array information. In C*, as in ZPL, a single structure is used to store the parallel array distribution information. Finally, as for

ZPL, a new AIF type would need to be added so that parallel array shape information would be accessible to the debugger client.

CONCLUSION

This chapter has presented the details comprising the implementation of the GUARD debugger. In particular, the chapter has examined the aspects of the implementation that enable the debugger to support the relative debugging paradigm. These include the client/server implementation details consisting of the debug client, the debug server and a debug API that allows the client and server to communicate efficiently and effectively. In addition, the chapter has examined innovations that are uniquely specific to the GUARD debugger. These include the dataflow technology, AIF, support for data and code transformations and support for data parallel languages.

From the information provided in this chapter, it should be clear that a number of significant advances in debugger technology have been made in the process of implementing the relative debugging paradigm. It is hoped that the techniques presented here will provide other debugger designers with a powerful platform on which to build future tools for parallel computer systems.

Chapter 7

CASE STUDIES IN RELATIVE DEBUGGING

This chapter presents a series of three case studies to demonstrate the application of relative debugging in a parallel computing environment. The case studies have been chosen in order to show how relative debugging can be used on a range of codes that utilise the major parallel architectures. These include a data parallel case study, a distributed memory message passing case study, and a shared memory case study. The aim is to demonstrate that the architecture proposed in Chapter 4 is sufficiently powerful to handle the differences between two languages, between a broad range of parallel programming architectures and in the underlying platforms. The case studies also highlight the efficiency of the debugging technique for locating errors across multiple program versions.

CASE STUDY 1: DATA PARALLEL CODE

The data parallel case study illustrates the power of relative debugging when applied to a sequential C code that has been ported to the data parallel language ZPL.

The program is a hydrodynamics code, known as "simple" [18], that is used to model the hydrodynamics of a pressurised fluid inside a spherical shell. To demonstrate our debugging methodology, a problem size of 128×128 using four iterations has been chosen. The output of the "simple" code is a scalar error value that is calculated from the values used in the hydrodynamics computation. For the C and ZPL comparison, the ZPL code was run using four processes in a 2×2 mesh. The initial run of the codes produced scalar error values that differ at around the fourth decimal place. Figure 7.1 shows the values that were produced for each of the four iterations.

Iteration	C	ZPL(2x2)
1	0.984958283	0.984946715
2	0.985004506	0.984971498
3	0.985086136	0.985033153
4	0.985224992	0.985147640

Figure 7.1: Scalar Error Value

Both the C and ZPL codes employ double precision variables for all computations, so it would be expected that the scalar error value would be equivalent within the precision

available, or about 15 decimal places (although only 9 decimal places are shown in Figure 7.1). The precision of the floating-point representation also allows us to set a lower bound for the error tolerance used in defining assertions. In this case study both codes use double precision, so the lower bound will be 10^{-15}. In situations where different precision is used in each code, the lower bound will need to be adjusted to the larger of the two values.

Our initial hypothesis was that the C and the new ZPL codes were working correctly, even though on first examination it can be seen that the codes produce slightly different results. In order to account for the discrepancy, it was assumed that different numeric evaluation techniques in the language run time systems or minor numerical errors were the likely cause. Since it was not obvious at the outset which of these factors was contributing to the differences, it was necessary to adopt the three-phase approach shown in Figure 7.2 when debugging the codes.

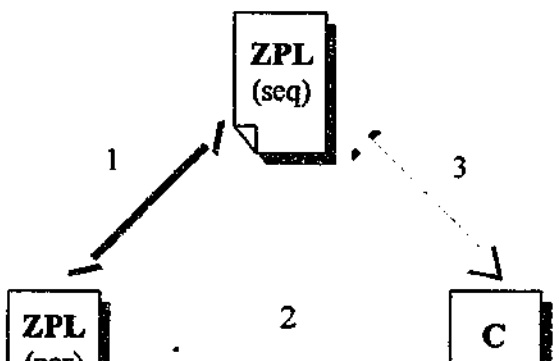


Figure 7.2: Code Comparison Steps for the "simple" Code

The first step compares a single process ZPL code with the same code on multiple processes in order to ensure that the ZPL runtime system is not introducing any differences into the results. The second step compares the parallel ZPL code to the C code so that errors in the ZPL version of "simple" can be identified and corrected. The last step compares the serial ZPL code to the C code as a final check to verify that all errors have actually been corrected.

Relative debugging relies on the ability to compare a suspect program with a reference code. As a result, a debugging methodology using an iterative refinement process can be used to narrow down the region (or regions) containing potentially incorrect code until the error is

located. This process has been applied very successfully in the past in various case studies [3,4,5].

CODE DESCRIPTION

The "simple" code is used to model the hydrodynamics of a pressurised fluid inside a spherical shell. The simulation computes values that describe the physics of the fluid at many points inside the shell over a number of time steps. Each iteration results in the computation of new values for various physical quantities such as velocity, density, energy, viscosity, pressure and temperature. Although "simple" is modelling the inside of a spherical shell, the symmetry of the problem allows the computation to be reduced to one quarter of an annular region. This region can then be transformed into Cartesian coordinates so that each physical quantity can be stored in a 2-dimensional array.

The structure of the ZPL version of the "simple" code is shown in Figure 7.3. The ZPL code combines all the hydrodynamics calculations in a single loop of N iterations, with the scalar error computation performed by lines 540 and 544. The first line computes the `En_error` array. The second line performs a reduction across all elements of the `En_error` array to produce the scalar error `Sc_error`.

ZPL introduces parallelism into the "simple" code by defining the variables for each physical quantity over a region. This region can then be automatically partitioned by the ZPL run-time system into blocks that are managed by the individual processes. The algorithm has also been designed so that computations in each phase share the same data dependencies, thus minimising the overall communications overhead required for communication between processes. More details on the ZPL implementation of "simple" can be found in [41].

The C code is structured slightly differently from the ZPL in that each phase of the hydrodynamics calculation is located in a separate module. The main body of the C code, shown in Figure 7.4, is contained in the module `simple.c`, and consists of an initialisation phase, `load()`, followed by N iterations over delta, hydro, heat and energy phases. The load phase routine `load()` is located in `load.c`, `delta()` in `delta.c`, `heat()` in `heat.c` and so on.

```
      program simple;
      ...
      region
        R = [1..DL, 1..DK];
        WEST = [1..DL, 1..DK];
      ...
      direction
        east = [0,+1];
      ...
      var
        X: [R] double;
        Heat: [R] double;
        En_error: [R] double;
        Theta: [R] double;
        Delta_t: [R] double;
        Sc_error: double;
      ...
      procedure InitPositionVelocity()

      begin
154     R := (Index2-1) * deltaR / (maxX - minX) + Rmin;
155     W := (((maxY-Index1) * PI) / (2 * (maxY - minY))) + angleOffset;
156
157     X.r := R * cos(W);
158     X.z := R * sin(W);

      end;
      ...
      procedure simple()
      ...
      [R]
      begin
        /* Initialisation Routines */
        ...
        for count := 0 to N-1 do
          ...
          /* Delta Phase */
          ...
          /* Hydro Phase */
          ...
          /* Heat Phase */
          ...
474       for i := DK-1 downto 0 do
475         [,i] Theta := Alpha * Theta@east + Beta;
476       end;
          ...
479       [WEST] Heat := (Theta - Theta@east) * R_ * Delta_t;
          ...
          /* Energy Phase */
          ...
538       Heat := Heat * Delta_t;
539
540       En_error := Int_en + Kin_en - Work + Heat;
          ...
544       Sc_error := +<< En_error;
          ...
        end;
      end;
```

Figure 7.3: ZPL Code Structure for the "simple" Code

```
main()
{
  int loop = 0;

  load();
  ...
  do
  {
    delta();
    hydro();
    heat();
    energy();
    ...
  } while (loop++ < N);
  ...
}
```

Figure 7.4: C Code Structure for the "simple" Code

In the C code, calculation of the scalar error value is performed in the routine `energy()`, shown in Figure 7.5. The scalar error value is computed as the sum of all elements in the energy error array `en_error` (line 92), which is in turn derived from the values of the energy phase arrays `int_en`, `kin_en`, and `work`, and the boundary heat flow array `heat` (line 86). Values for these arrays are computed in the corresponding phase routines.

SERIAL/PARALLEL ZPL COMPARISON

The first step in the debugging process was a comparison of the ZPL code in a single process configuration with that in a multi-process configuration, in this case four processes in a 2×2 mesh. The results from these runs showed that different process topologies produced slight variations in the scalar error value, with a magnitude of slightly greater than 10^{-15}. Since differences of this magnitude are still significant for double precision floating-point numbers, these appeared to be errors introduced by the parallel run-time system. Relative debugging was then used to determine the cause of these differences, by defining assertions over the phase variables `Int_en`, `Kin_en`, `Work` and `Heat`, and the error value `En_error`. However, no differences were visible in these variables. This meant that the source of the variations must be the final reduction operation at the end of each iteration. As the order of the floating-point operations is the only factor affected by topology changes, it is likely that the non-associative nature of these operations was the cause of the variations. This result is an important one, since it shows that non-deterministic behaviour can impact on the results of the computation, and effectively sets a lower bound to the accuracy of the final error value.

```
      extern double heat[DL][DK];
      extern double delta_t[DL][DK];
      double int_en[DL][DK];
      double kin_en[DL][DK];
      double work[DL][DK];

      energy()
      {
        int i, j;
        double local_error_sum = 0.0;
        ...
81      for (i=0; i<DL; i++) {
82       for (j=0; j<DK; j++) {
83        heat[i][j] *= delta_t[i][j];
86        en_error[i][j] = int_en[i][j] + kin_en[i][j]
                         + work[i][j] + heat[i][j];
87       }
88      }
89
90      for (i=0; i<DL; i++) {
91       for (j=0; j<DK; j++) {
92        local_error_sum += en_error[i][j];
93       }
94      }
        ...
      }
```

Figure 7.5: `energy.c` - C Scalar Error Calculation

ZPL AND C COMPARISON

Having isolated the variations introduced by the parallel non-determinism, it was now possible to identify the cause of the differences between the ZPL and C codes. The next step of the debugging process involved using an iterative refinement process to identify and correct four errors in the code.

Error 1: Extra Term In An Expression

Debugging of the ZPL and C "simple" codes starts by defining assertions for the four phase variables used in the scalar error calculation. Since the magnitude of the error was around 10^{-4} the initial error tolerance was set to $10^{-5} \leq \varepsilon \leq 10^{-1}$:

```
set error 1.0e-5 1.0e-1
assert $zpl::Int_en@"simple.z":540 = $c::int_en@"energy.c":81
assert $zpl::Kin_en@"simple.z":540 = $c::kin_en@"energy.c":81
assert $zpl::Work@"simple.z":540 = $c::work@"energy.c":81
assert $zpl::Heat@"simple.z":540 = $c::heat@"energy.c":81
```

The results from these assertions indicate differences in the `$c::heat` and `$zpl::Heat` arrays which are used to store the results of the heat phase computation. Figure 7.6 shows a visualisation of these differences. Since the variables are 2-dimensional arrays, it is convenient to visualise the differences as a 2-dimensional bitmap. These bitmaps are

generated from difference information by assigning a colour to represent the magnitude of the difference, ranging from blue for the smallest difference, through green, yellow and red, to black for the largest. White indicates no difference.

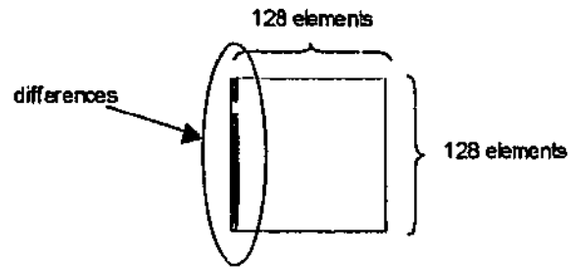


Figure 7.6: Differences between `$c::heat` and `$zpl::Heat`

Differences can be seen along the left (or western) edge of the comparison. The western edge of the array is accessed in ZPL using the syntax "`[WEST] Heat := ...`", where `WEST` has been defined as the appropriate region. A search of the ZPL code results in only one example of such syntax at line 479 in `simple.c`. The corresponding C code can be seen at lines 163-165 of `heat.c` in Figure 7.7.

```
      extern double heat[DL][DK];
      extern double theta[DL][DK];
      double temp_theta[DL][DK];
      ...
      heat()
      {
        ...
143     for (i=DK-2; i>=0; i--) {
144      for (j=0; j<DL; j++) {
145       temp_theta[j][i] = theta[j][i+1];
146      }

149      for (j=0; j<DL; j++) {
150       theta[j][i] = alpha[j][i] * temp_theta[j][i] + beta[j][i];
151      }
152     }
        ...
163     for (i=0; i<DL; i++) {
164      heat[i][0] = (theta[i][0] - temp_theta[i][0]) * r[i][0];
165     }
        ...
      }
```

Figure 7.7: `heat.c` – C Heat Phase Code

Careful examination of both codes indicates that the term `Delta_t` (shown in red in the ZPL code) has been erroneously included in the computation of `Heat` in the ZPL code in Figure 7.3. Prior to using relative debugging, this error had not been detected, even though it

is obvious post fact. In this case, relative debugging allowed us to identify a faulty statement in the ZPL code fragment, even though the syntax and implementation details of the languages are completely different.

Error 2: Incorrectly Specified Constant

After correcting the first error, there are still differences visible in the output, though the magnitude has now been reduced to around 10^{-7}. Setting the error tolerance to $10^{-10} \leq \varepsilon \leq 10^{-1}$ and re-running the original assertions now indicates differences between the `$c::int_en` and `$zpl::Int_en` arrays, which are used to store the internal energy values computed in the energy phase. Figure 7.8 shows these differences.

Figure 7.8: Differences between `$c::int_en` and `$zpl::Int_en`

By defining additional assertions, it is possible to observe that differences in many of the variables involved in the computation of the internal energy have similar characteristics to those seen in Figure 7.8. With differences occurring in so many variables, no clear path to the source of the error is evident. Instead, the initialisation code was examined for similar patterns of differences, beginning with the code to initialise the position and velocity components used throughout the program. Figure 7.9 shows the C version of this code, which corresponds to lines 154-158 of the ZPL code.

Visualising differences in variables from these codes shows the characteristic pattern in the differences between `$c::w` and `$zpl::W`, and `$c::x` and `$zpl::X`. As both `$c::w` and `$zpl::W` are computed entirely from constants, the problem must be related to these variables. Further examination indicates that the value of `PI` used in the ZPL code was only specified to 7 decimal places, while the corresponding value used in the C code, `M_PI` was specified to 20 decimal places (of which only 15 are significant).

```
101   for (i=0; i<DL+2; i++) {
102    for (j=0; j<DK+2; j++) {
103     r[i][j] = ((PEj-1)*DK + (j-1)) * deltaR / (NUM_K_PROCS *
                  DK - 1) + Rmin;
104     w[i][j] = ((M_PI * ((PEi-1) * DL + (DL-i)))
                  / (2 * (NUM_L_PROCS * DL-1))) + ANGLE_OFFSET;
        ...
107     x[i][j].r = r[i][j] * cos(w[i][j]);
108     x[i][j].z = r[i][j] * sin(w[i][j]);
109    }
110   }
```

Figure 7.9: `init.c` - C Position and Velocity Initialisation Code

Like the previous error, the difference in `PI` is obvious with hindsight. However, because a symbolic constant is used in the code, a cursory examination would not have revealed the difference. Relative debugging allowed us to identify a characteristic pattern of differences that was visible in a number of variables, suggesting that a common source was responsible. The error was eventually located by tracing this pattern back to the constant declarations, even though the two languages use different syntactic structures for defining constants.

Error 3: Invalid Boundary Conditions

On correcting the second error there are still differences in the output of the programs, though the magnitude had now been reduced further, to around 10^{-10}. Setting the error tolerance to $10^{-15} \leq \varepsilon \leq 10^{-1}$ and re-running the original assertions shows that the `$c::heat` and `$zpl::Heat` arrays are again the source of errors. A series of assertions must now be applied to narrow down the erroneous region of code in the heat phase computation. The result of these assertions is that the problem appears to be occurring between lines 474 and 479 of the ZPL code and lines 143 and 152 of the heat phase calculation in `heat.c` (Figure 7.7).

The ZPL code uses the `(i+1)`st column of `Theta` in the computation `Alpha * Theta@east + Beta` and propagates this across columns `(DK-1)` to 0 of `Theta`. The C code uses a temporary array to hold the `(i+1)`st column of theta. However, the outer loop of the C code only ranges from `(DK-2)` to 0, so the `(DK-1)`st column is not computed, and hence the C code is incorrect. An identical situation is also found in the computation of the north boundary condition.

This error is interesting because the ZPL code is actually correct, while the original C code is incorrect. When developing the ZPL code the programmer did not need to be concerned with issues such as computing loop bounds, but instead was able to concentrate on the

underlying physics of the model. In comparison, the C programmer needed to consider the loop bound issues, with the extra complexity presumably leading to the coding error. In spite of these significant implementation differences, relative debugging was able to identify the incorrect code.

Error 4: Wrong Sign

Even after correcting the third error, the magnitude of the differences in the output of the programs still remains at around 10^{-10}. Further examination of the scalar error computation indicates that while the values of `$c::en_error` and `$zpl::En_error` have a very small differences, the calculations of the energy, work and heat values are now identical. This can only point to a problem with calculating the error value itself. Close examination of the ZPL and C code shows the source of the error, which can be seen at line 540 of the ZPL code (Figure 7.3) and line 86 of the C code (Figure 7.5).

Using relative debugging it was possible to identify a pattern of differences and quickly pinpoint the location of erroneous code that was only obvious with the benefit of hindsight. Interestingly, the original paper describing the "simple" code gives the error computation as:

```
Int_en + Kin_en - Work - Heat
```

This means that both programs are actually incorrect.

SERIAL ZPL AND C COMPARISON

The final step in the debugging process is to verify that the changes made to both the ZPL and C codes have resulted in bitwise equivalence of the variables used in the physics computation. Since non-determinism was introduced by running the ZPL code in parallel, comparison of the serial ZPL and C codes must be used. For this test, the same series of assertions defined over the four phase variables were used and the error tolerance was set to 0 (although a tolerance of 10^{-16} would have been equally valid). As predicted, the results showed that each of the variables were now identical.

The debugging exercise outlined in this case study took a remarkably short time, considering that the programmer performing the case study was not the author of either version of "simple" and was not particularly fluent in ZPL. Whilst it is dangerous to generalize the results too far, this example adds to the evidence of our other case studies that support the

power of relative debugging – in the case even when the programs run on different computers and are written in different languages.

CASE STUDY 2: DISTRIBUTED MEMORY CODE

The second case study examines how relative debugging can be applied to a hand-coded distributed memory code. The program is a distributed memory version of a simple numerical model of the shallow water equations [62]. Abramson, et al. ported the shallow water equations to a number of languages and architectures [1]. This case study will use versions written in C for sequential and distributed memory architectures.

For this case study, a model size of 32×32 and a run length of 950 time steps was chosen. The distributed memory code was initially run on two processors. The codes are designed to report a number of key values every 50 time steps, so it is simple to determine if the codes are working correctly or not. Figures 7.10(a) and 7.10(b) show the output from the two versions of the code after 950 time steps in which significant errors can be clearly seen in all four values.

```
Cycle number  950    Model time in days   0.99
  Potential energy     6816.106  Kinetic Energy    41183.156
  Total Energy        47999.262  Pot. Enstrophy     2.006166e-27
```

(a) Sequential

```
Cycle number   950    Model time in days   0.99
   Potential energy     6790.644  Kinetic Energy    41219.078
   Total Energy        48009.723  Pot. Enstrophy     6.152948e-17
```

(b) Distributed Memory

Figure 7.10 Output from the "shallow" Code

CODE DESCRIPTION

According to Abramson, et al., "the shallow water equations describe the motion of an incompressible fluid with a free surface, with the constraint that the horizontal scales of motion are much larger than the vertical. The equations are a favoured choice for experiments with various model structures and numerical schemes, and thus must be ported to many different computing platforms. Although they use a very simple representation of the atmosphere, they do include the two types of horizontal wave motion important in more realistic global climate models, gravity waves and Rossby waves." [1]

The sequential C code consists of two main phases: an initialisation phase and a time-step phase. Figure 7.11 shows the overall structure of the sequential code.

```
      float   u[m][n];      /* Zonal wind */
      float   v[m][n];      /* Meridional wind */
      float   p[m][n];      /* Pressure (or free surface height) */
      float   cu[m][n];     /* Mass weighted u */
      float   cv[m][n];     /* Mass weighted v */
      float   z[m][n];      /* Potential enstrophy */
      float   h[m][n];
      float   psi[m][n];    /* Velocity stream function */
      float   dudt[m][n];   /* Time tendency of u */
      float   dvdt[m][n];   /* Time tendency of v */
      float   dpdt[m][n];   /* Time tendency of p */

      main()
      {
        int ncycle;

        ...
58      initialise(u,v,p,psi,di,dj);
        ...
61      for (ncycle = 0; ncycle < itmax; ncycle++) {
          /* Calculate cu, cv, z and h*/
67        calcuvzh(p,u,v,cu,cv,z,h,fsdx,fsdy);
          ...
          /* Calculate time tendencies of u, v and p */
71        timetend(dudt,dvdt,dpdt,z,cv,cu,h);
          ...
          /* Calculate new values for u, v and p */
78        tstep(u,v,p,dudt,dvdt,dpdt,firststep,tdt);
          ...
80        if ( firststep ) {
             /* Double tdt because all future steps are leapfrog */
81          tdt = tdt+tdt;
82          firststep = 0;
83        }
85      } /* End of time step loop */
        ...
      }
```

Figure 7.11: Sequential C version of "shallow"

Initialisation of the key data structures is performed by the `initialise()` routine. The time step loop is then executed for the number of iterations determined by `itmax`. This loop executes three routines in succession. The `calcuvzh()` routine is used to calculate the mass fluxes `cu` and `cv`, the potential enstrophy `z` and the quantity `h`. Next, the `timetend()` routine is used to calculate the time tendencies for velocity components `dudt` and `dudt`, and for the pressure `dpdt`. Finally, the `tstep()` routine is called to recalculated new values for the velocity components u and v, and the pressure p in a "leapfrog" time step process.

The distributed memory code uses a master/slave arrangement, where the master maintains primary copies of the key data structures. Each slave is sent an entire copy of the data structures, but only performs computations on a portion determined by slicing the outer loop. Before and after a calculation, each slave synchronises the edges of the data structures

with its immediate neighbours. At the end of the time step loop the slave data is copied back to the master process. This arrangement is shown in Figure 7.12.

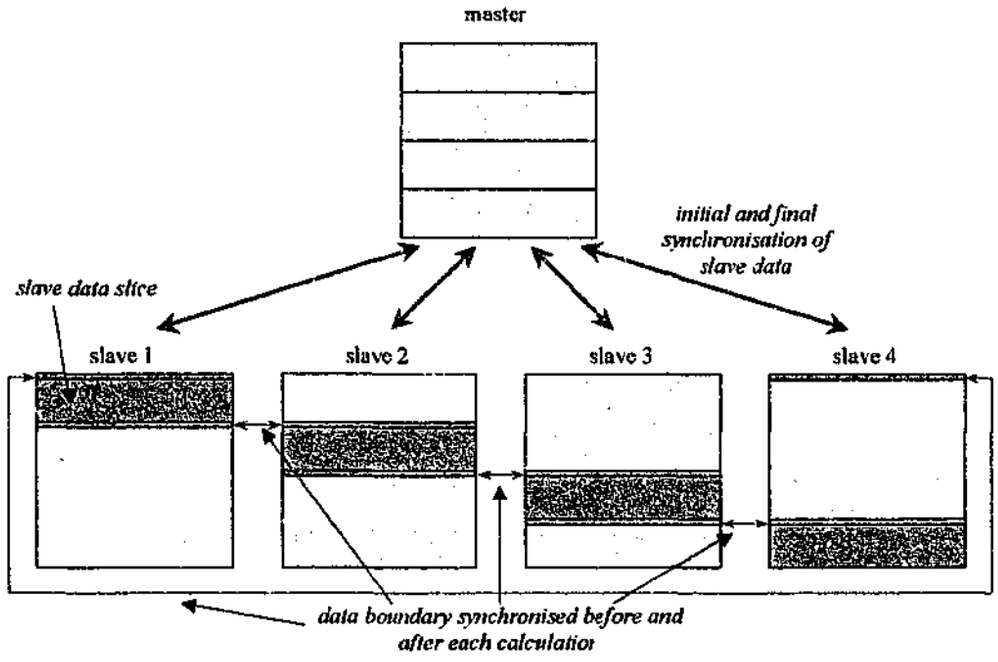


Figure 7.12: Data Decomposition and Boundary Synchronisation

The structure of the master code is shown in Figure 7.13. The master begins by initialising the distributed memory subsystem (in this case MPI) and determining the number of slave processes (lines 106 – 107). The loop at lines 137 – 162 is then used to determine the decomposition of the data structures. The values `jstart` and `jend`, which define the lower and upper bounds of the slices respectively, are computed and sent to each slave process. Next, the loop at lines 178 – 208 sends the contents of each key variable to the slaves, a row at a time. Finally, the `update_global_ds()` routine is used to recover the contents of each of the variables. This routine waits for each slave to finish its computation and send the final results, which are collected and stored in the appropriate data structure.

The slave code, shown in Figure 7.14, is similar in operation to the sequential code, however it is complicated by the addition of synchronisation code. The first action of the slave routine is to obtain the slice bounds `jstart` and `jend`. These values are then used throughout the computations to determine the slice of data the slave will be calculating.

```
      float   u[n][m];      /* Zonal wind */
      float   v[n][m];      /* Meridional wind */
      float   p[n][m];      /* Pressure (or free surface height) */
      float   cu[n][m];     /* Mass weighted u */
      float   cv[n][m];     /* Mass weighted v */
      float   z[n][m];      /* Potential enstrophy */
      float   h[n][m];
      float   psi[n][m];    /* Velocity stream function */
      float   dudt[n][m];   /* Time tendency of u */
      float   dvdt[n][m];   /* Time tendency of v */
      float   dpdt[n][m];   /* Time tendency of p */

      main(int argc, char *argv[])
      {
        int proc_cnt, jstart, jend;
        ...
106     MPI_Init(&argc, &argv);
107     MPI_Comm_size(MPI_COMM_WORLD, &proc_cnt);
        ...
137     for (i=0; i < proc_cnt; i++) {

139       jstart = (i-1) * chunk_size;

141       jend = (i * chunk_size - 1) % n;

160       MPI_Send(&jstart,MPI_INT,i,MPI_COMM_WORLD);
161       MPI_Send(&jend,MPI_INT,i,MPI_COMM_WORLD);
162     }
        ...
175     initialise(u,v,p,psi,di,dj);
        ...
178     for (i=0; i < proc_cnt; i++) {
179       for (j=0; j < n; j++) {
            /* send row of each variable */
207       }
208     }
        ...
218     update_global_ds(u);
        ...
223     update_global_ds(v);
        ...
228     update_global_ds(p);
        ...
233     update_global_ds(h);
        ...
238     update_global_ds(z);
        ...
245     MPI_Barrier(MPI_COMM_WORLD);
247     MPI_Finalize();
      }
```

Figure 7.13: Distributed Memory Master Code

Next, the loop at lines 375 – 403 is used to receive the contents of the key variables. The main time step loop at lines 405 – 449 is identical to the sequential code except that the data structure boundaries are synchronised before and after the `calcuvzh()` and `timetend()` calculations. This synchronisation is performed using the `calc_load()`, `calc_unload()`, `time_load()` and `time_unload()` routines.

```
     slave()
     {
       ...
363    MPI_Recv(&jstart,MPI_INT,MPI_ANY_SOURCE,MPI_COMM_WORLD);
364    MPI_Recv(&jend,MPI_INT,MPI_ANY_SOURCE,MPI_COMM_WORLD);
       ...
375    for (i=0; i < n; i++) {
         /* receive row of each variable */
403    }

405    for (ncycle = 0; ncycle < itmax; ncycle++) {
         ...
421      calc_load(jstart,jend,p,u,v);
422      calcuvzh(jstart,jend,p,u,v,cu,cv,z,h,fsdx,fsdy);
423      calc_unload(jstart,jend,cv,z);
         ...
430      time_load(jstart,jend,cu,cv,h,z);
431      timetend(jstart,jend,dudt,dvdt,dpdt,z,cv,cu,h);
432      time_unload(jstart,jend,dvdt);
         ...
         /* Calculate new values for u, v and p */
439      tstep(jstart,jend,u,v,p,dudt,dvdt,dpdt,firststep,tdt);
         ...
442      if ( firststep ) {
            /* Double tdt because all future steps are leapfrog */
444        tdt = tdt+tdt;
445        firststep = 0;
446      }
         ...
449    }  /* End of time step loop */

457    send_updated_ds(u);
458    send_updated_ds(v);
459    send_updated_ds(p);
460    send_updated_ds(h);
461    send_updated_ds(z);
       ...
       MPI_Barrier(MPI_COMM_WORLD);
       MPI_Finalize();
     }
```

Figure 7.14: Distributed Memory Slave Code

ERROR 1: INCORRECT INDEX VALUE

In order to begin debugging the "shallow" code, a map that describes the data decomposition technique used by the distributed memory code was defined. One major difference between the codes is the index ordering which has been swapped in the translation from sequential to parallel, presumably to take advantage of some cache effect. The map was also defined to account for this index permutation.

```
map shallow(P::A)
  define index(i,a,u,l) = a[1-i]
  define proc(i,a,u,l) = a[l] * ($procs - 1) / $n
end
```

Using this map, a series of assertions were then defined that compare the values of the key data structures u, v and p between the sequential and parallel codes just prior to entering the time step loop. This tests the initialisation code in the two programs, and ensures that the

variables are correct before any calculations are undertaken. The error limits were set to an initial estimate of the likely error range.

```
set error 1.0e-20 1.0e-1
assert $c::u@"cshallow.c":61 = shallow($p::u@"main.c":405)
assert $c::p@"cshallow.c":61 = shallow($p::p@"main.c":405)
assert $c::v@"cshallow.c":61 = shallow($p::v@"main.c":405)
```

Execution of these assertions showed that u, p and v are correct prior to entering time step loop. This confirms that the errors must be being introduced by calculations in the time step loop.

The next step was to examine the result of the computation `calcuvzh()`. This was done by defining assertions to compare the values of the variables cu, cv, h and z as follows:

```
assert $c::cu@"cshallow.c":71 = shallow($p::cu@"main.c":430)
assert $c::cv@"cshallow.c":71 = shallow($p::cv@"main.c":430)
assert $c::h@"cshallow.c":71 = shallow($p::h@"main.c":430)
assert $c::z@"cshallow.c":71 = shallow($p::z@"main.c":430)
```

After first call to `calcuvzh()`, differences were observed in the variable cv. These differences, shown in Figure 7.15, suggested that the error was related to a boundary calculation by each processor.

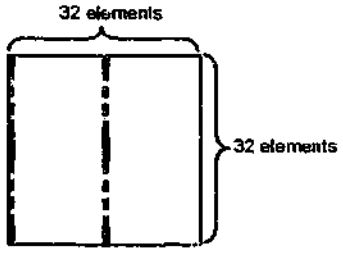


Figure 7.15: Differences in cv

To confirm this suspicion, the codes were re-run with the distributed memory code running on four processors rather than the initial two. Errors were again observed in the variable cv, however this time the differences appeared as shown in Figure 7.16.

Figure 7.16: Differences in cv Running on Four Processors

This confirmed that cv was affected by a boundary issue that depended on number of processes. Since the error was in the boundary column, the boundary code was examined for coding defects. Inspection of `calcuvzh()` code lead to the discovery of the difference shown in Figures 7.17(a) and (b).

```
cv[i][jp] = ...          cv[j][i] = ...
```

(a) Sequential *(b) Distributed Memory*

Figure 7.17: Discovered Differences in `calcuvzh()` Code

Even allowing for the index permutation, the parallel code was clearly using the wrong index. Relative debugging allowed the rapid identification of this error by observing the changes resulting from running on different numbers of processors, and irrespective of the fact that the array indexes are permuted.

ERROR 2: WRONG ARRAY ELEMENT

After correcting the code and re-running the programs using the same assertions, an error was still observed in the cv variable. The characteristics of the differences had altered significantly however, and now seemed to indicate a periodic error in the cv calculation. The new differences are shown in Figure 7.18.

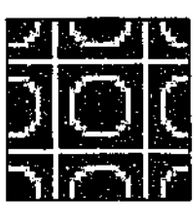


Figure 7.18: Periodic Error in cv

Further inspection of the `calcuvzh()` revealed a second error in the calculation of cv. In this case the wrong element of the array v was used in the calculation, as shown in Figure 7.19.

```
cv[i][jp] = 0.5 * (p[i][jp] + p[i][j]) * v[i][jp];
```

(a) Sequential

```
cv[jp][i] = 0.5 * (p[jp][i] + p[j][i]) * v[j][i];
```

(b) Distributed Memory

Figure 7.19: Second Error in `calcuvzh()` Code

This error was interesting because it occurred in the same statement as the previous error, but was missed in the original code inspection. Because the nature of the error in a single variable changed from a boundary problem to a periodic problem, the source of the problem was quickly identified.

ERROR 3: WRONG SIGN

After ensuring that both errors had been corrected in the parallel code, the programs were rerun to check that the cv variable was now correct. The assertions showed that all four calculated variables, cu, cv, h and z were correct after first iteration, however on subsequent iterations errors were observed accumulating in the cu variable. Figure 7.20 shows a time sequence of these errors, one frame per iteration.

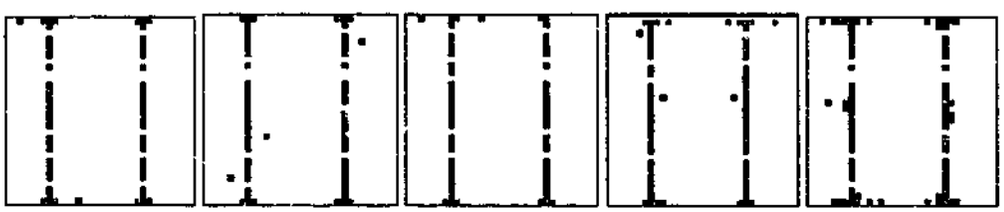


Figure 7.20: Time Sequence of Errors in cu

Inspection of the code showed that the calculation of cu is derived from the values of p and u. Since all values of cu are correct on first iteration but incorrect on second and subsequent iterations, this implies that there must be in error in the computation of p and u (which were shown to be correct immediately after initialisation).

In order to check the values of p and u at the end of the time step loop, the following assertions were defined:

```
assert $c::u@"cshallow.c":85 = shallow($p::u@"main.c":449)
assert $c::p@"cshallow.c":85 = shallow($p::p@"main.c":449)
```

When the codes were again run, errors were immediately reported in u. These are shown in Figure 7.21.

Figure 7.21: Errors in u

At this point it was clear that errors were being introduced into the variable u, but these could have resulted from either the calculation of u itself in `tstep()` or in the time tendency variable `dudt` which is calculated in `timetend()`. To verify which routine was the source of the errors, it was necessary to rerun and check the value of `dudt` immediately after the call to `timetend()`. Figure 7.22 shows the differences visible in dudt when this was done.

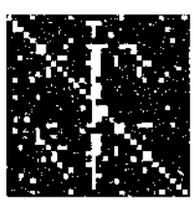


Figure 7.22: Errors Reported in `dudt`

It was now known with reasonable certainty that the errors were occurring in the `timetend()` routine. At this point a visual inspection could be used to locate the source of the error. In this case the error turned out to be a wrong sign used in the calculation of `dudt`. This can be seen in the code segments presented in Figure 7.23(a) and (b).

This error highlights the use of relative debugging to quickly reduce the possible location of an error to a small region of code. Once this has been done, simple code inspection will usually pinpoint the incorrect statement immediately, even when the actual error is very minor as in this case.

```
dudt[j][ip] =
   0.125 * (z[jp][ip] * (cv[jp][ip] + cv[jp][i]) - z[j][ip] *
   (cv[j][ip]+cv[j][i])) - (h[j][ip] - h[j][i]) * invdx;
```

(a) Sequential

```
dudt[j][ip] =
   0.125 * (z[jp][ip] * (cv[jp][ip] + cv[jp][i]) + z[j][ip] *
   (cv[j][ip]+cv[j][i])) - (h[j][ip] - h[j][i]) * invdx;
```

(b) Distributed Memory

Figure 7.23: Final Error in Distributed Memory Code

CASE STUDY 3: SHARED MEMORY

This final case study examines the used of relative debugging on a sequential C program ported to a shared memory architecture. The same shallow water model used in the previous case study is again used here, however the implementation of the shared memory code is significantly different from that of the distributed memory version. The shared memory code is written in C using the Argonne P4 system [14].

On execution of the codes, it was observed that all values are correct after the first iteration. However, after only 50 iterations these values begin to diverge significantly. The output from the sequential and parallel codes after 50 iterations is shown in Figure 7.24.

```
Cycle number     1    Model time in days   0.00
  Potential energy        0.000  Kinetic Energy    48036.828
  Total Energy        48036.828  Pot. Enstrophy     0.000000e+00

 Cycle number    50    Model time in days   0.05
  Potential energy     5681.312  Kinetic Energy    42371.008
  Total Energy        48052.320  Pot. Enstrophy     8.849624e-28
```

(a) Sequential

```
Cycle number     1    Model time in days   0.00
  Potential energy        0.000  Kinetic Energy    48036.828
  Total Energy        48036.828  Pot. Enstrophy     0.000000e+00

 Cycle number    50    Model time in days   0.05
  Potential energy     6649.025  Kinetic Energy    35192.219
  Total Energy        41841.242  Pot. Enstrophy     2.349158e-16
```

(b) Shared Memory

Figure 7.24: Output From Sequential and Parallel Codes After 50 Iterations

The shared memory version of "shallow" uses the same master/slave arrangement as the distributed memory code, but is considerably simpler since there is no longer a requirement

to synchronise data between processes. The main time step loop of the shared memory code is shown in Figure 7.25. In this version, the key data structures are maintained in global shared memory that is accessed by each process. Code synchronisation is achieved by calling the `waitfor()` routine after each calculation.

```
      struct globmem {
        float   u [m][n];     /* Zonal wind */
        float   v [m][n];     /* Meridional wind */
        float   p [m][n];     /* Pressure (or free surface height) */
        float   cu [m][n];    /* Mass weighted u */
        float   cv [m][n];    /* Mass weighted v */
        float   z [m][n];     /* Potential enstrophy */
        float   h [m][n];
        float   psi [m][n];   /* Velocity stream function */
        float   dudt [m][n]; /* Time tendency of u */
        float   dvdt [m][n]; /* Time tendency of v */
        float   dpdt [m][n]; /* Time tendency of p */
        int     ncycle;
        float   time;
        float   tdt;
        int     firststep;
        int     nproc;
      } *glob;

      slave()
      {
        ...
        jstart = (mynum - 1) * chunk_size;
        jend   = mynum * chunk_size - 1;
        ...
        waitfor();
        ...
140     while ( ncycle < itmax ) {
          ...
145       calcuvzh(jstart,jend,glob->p,glob->u,glob->v,glob->cu,
            glob->cv,glob->z,glob->h,fsdx,fsdy);

148       waitfor();
          ...
151       timetend(jstart,jend,glob->dudt,glob->dvdt,glob->dpdt,
            glob->z,glob->cv,glob->cu,glob->h);

154       waitfor();
          ...
          /* Calculate new values for u, v and p */
164       tstep(jstart,jend,glob->u,glob->v,glob->p,glob->dudt,
            glob->dvdt,glob->dpdt,glob->firststep,glob->tdt);

169       waitfor();
170       if ( mynum == 1 ) {
171         if ( glob->firststep ) {
172           /* Double tdt because all future steps are leapfrog */
173           glob->tdt = glob->tdt + glob->tdt;
174           glob->firststep = 0;
175         }
176         glop->ncycle++
177       }

179       waitfor();

180     }  /* End of time step loop */
        ...
      }
```

Figure 7.25: Shared Memory Slave Code

ERROR 1: LOOP BOUND ERROR

As in case study 2, debugging the codes begins by checking that the key variables have been correctly initialised. Rather than define a series of assertions, it was decided that it would be faster to manually set breakpoints in the sequential and parallel codes and then execute the following series of comparison statements in the debugger:

```
compare $c::u = $m::"glob->u"
compare $c::v = $m::"glob->v"
compare $c::p = $m::"glob->p"
compare $c::psi = $m::"glob->psi"
compare $c::uold = $m::"glob->uold"
compare $c::vold = $m::"glob->vold"
compare $c::pold = $m::"glob->pold"
```

These comparison statements indicated that there were no differences in the main data structures. At this point it is clear that the data was correct prior to entering the time step loop, and that errors were introduced in one or more of the calculation phases. To narrow down the location of the error, each calculation was checked in turn looking for differences in the results. First, the values of cu, cv, z and h were checked after the call to `calcuvzh()` using the following assertions:

```
assert $c::cv@"cshallow.c":71 = shallow($p::"glob->cv"@"pshallow.c":151)
assert $c::cu@"cshallow.c":71 = shallow($p::"glob->cu"@"pshallow.c":151)
assert $c::h@"cshallow.c":71 = shallow($p::"glob->h"@"pshallow.c":151)
assert $c::z@"cshallow.c":71 = shallow($p::"glob->z"@"pshallow.c":151)
```

No errors were apparent after the first iteration. However, after the second iteration errors in each of the variables was seen. These errors are shown in Figure 7.26 and were appearing in only the top two or three rows of the arrays.

Figure 7.26: Errors in cu, cv, h and z Respectively.

As the errors were introduced only on the second iteration, it was surmised that the errors must have been occurring in a subsequent calculation. Since the variables cu, cv, h and z are derived from u, v and p, the values of these variables were now checked after the call to `tstep()` using the following assertions:

```
assert $c::p@"cshallow.c":80 = shallow($p::"glob->p"@"pshallow.c":170)
assert $c::u@"cshallow.c":80 = shallow($p::"glob->u"@"pshallow.c":170)
assert $c::v@"cshallow.c":80 = shallow($p::"glob->v"@"pshallow.c":170)
```

Again, errors were observed all three variables as shown in Figure 7.27, but this time they occurred after the *first* iteration and only in the top two rows. However this still did not provide enough information to conclusively identify the cause of the error or its location.

Figure 7.27: Errors in u, v and p Respectively

Since u, v and p are derived from dudt, dvdt and dpdt, it was still necessary to check these after the call to `timetend()`. This was done using the assertions:

```
assert $c::dudt@"cshallow.c":73 = shallow($p::"glob->dudt"@"pshallow.c":155)
assert $c::dvdt@"cshallow.c":73 = shallow($p::"glob->dvdt"@"pshallow.c":155)
assert $c::dpdt@"cshallow.c":73 = shallow($p::"glob->dpdt"@"pshallow.c":155)
```

In this case, all three variables again exhibited errors in the top two rows after the first iteration. These errors are shown in Figure 7.28.

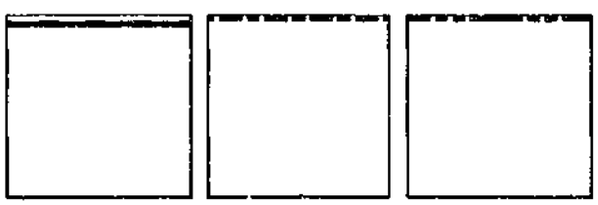


Figure 7.28: Errors in dudt, dvdt and dpdt Respectively

Using the evidence that had been gathered so far, it was now possible to conclusively identify the `timetend()` routine as the source of the errors since all other potential locations had been eliminated. Once the general location of the error had been established, the next step was to use the characteristic visualisation to identify the precise location. Since the error was manifest only in the first and second rows of the data structures (at least after the first iteration), there was a high likelihood that the problem was located in the loop bound code. Examination of the codes at this point quickly identified the error as an incorrect loop bound. This is shown in Figure 7.29.

```
for(j = 0; j <= n; j++) {
  jp = (j+1) % n;
  for (i = 0; i < m; i++){
    ip = (i+1) % m;
    dudt[ip][j] = ...;
    dvdt[i][jp] = ...;
    dpdt[i][j] = ...;
  }
}
```

(a) Sequential

```
for(j = jstart; j <= jend; j++) {
  jp = (j+1) % n;
  for (i = 1; i < m; i++){
    ip = (i+1) % m;
    dudt[ip][j] = ...;
    dvdt[i][jp] = ...;
    dpdt[i][j] = ...;
  }
}
```

(b) Shared Memory

Figure 7.29: Loop Bound Error in Shared Memory Code

While not particularly interesting in itself, this error again showed how a combination of judicious use of assertion statements and analysis of the error characteristic could quickly identify the location and nature of an error.

CONCLUSION

This chapter has presented a series of three case studies that examine the use of relative debugging to find errors in parallel programs using a range of different parallel architectures. The case studies show that relative debugging works equally effectively for data parallel programs, and for programs that have been designed to use distributed memory and shared memory architectures. A key feature of the GUARD debugger is that it allows a user to debug programs developed using three completely distinct parallel architectures within the same debugger.

In addition to support for a range of parallel architectures, a number of common themes emerge from the case studies.

- The iterative refinement technique is equally effective across different architectures, and is the primary method for locating error regions.

- Error characteristics can provide a useful tool for identifying specific errors, and can be used to trace an error to its source.

- The characteristics can change as errors are located and corrected, but this does not affect the debugging process.

- Observing changes resulting from varying the number of processes is a useful technique for isolating errors.

- Maps provide an effective mechanism for hiding data distribution and other data transformations.

- Minor coding errors can result in highly visible differences in variables, even thought the absolute magnitude of the differences is small.

- Error tolerances can be used to focus on the most significant errors first, followed by less significant errors later.

Each case study required the investment of considerable time and effort to debug all the errors that had been introduced into the codes as a result of the porting process. Clearly, many of the errors could have been located using traditional debugging practices such as instrumenting the code, or by employing existing parallel debuggers. No direct comparison was performed with these techniques. However the experience gained here suggests that relative debugging was able to provide an additional level of detail that would not ordinarily be available, and that this resulted in a faster and more efficient debugging process than would be possible using traditional debugging techniques.

Chapter 8

FUTURE DIRECTIONS & CONCLUSION

The current version of GUARD clearly demonstrates that the technique of relative debugging is a powerful tool for locating errors in parallel programs. As a research prototype however, the development of GUARD has focussed on a number of core technologies. These have included:

- a dataflow engine;
- an architecture independent data format;
- a data transformation/mapping; and
- a client/server architecture.

Although these core technologies combine to produce a powerful debugger, there are a number of areas that would benefit from further research and development. This chapter examines a number of enhancements that could be added to the current implementation in order to increase its functionality and usability. The final part of this chapter is devoted to concluding the dissertation.

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

Currently, GUARD exists as a stand-alone, command-line debugger. While a number of similar debuggers have gained wide acceptance, most notably Cygnus's GDB [66], the trend is to provide debuggers as part of a fully integrated development environment. IDEs offer a number of advantages over stand-alone tools, such as a homogeneous user interface and a simplified development cycle [46]. Research is currently underway to integrate GUARD into Microsoft's Visual Studio IDE.

Parallel debuggers also benefit greatly from the use of a graphical user interface, as is provided in many IDEs. Since parallel debuggers are generally dealing with multiple processes (and sometimes hundreds of processes), the control of these processes becomes a significant issue. A GUI provides the ideal mechanism for this purpose. In addition, the

user may wish to examine or modify the code and data from many processes simultaneously, and the GUI provides the most natural environment to facilitate this interaction.

MAPS/TRANSFORMATIONS

GUARD is unique in its ability to model data distribution and transformations that occur when serial programs are ported to parallel architectures. However, in the current implementation, both the distribution and transformation of data must be specified using a simple, low-level command language. One drawback with this technique is that it is difficult to specify complex mappings. This is because the user must have a detailed understanding of the mapping that has been used, and must produce what is in effect a formal specification of the mapping. A second drawback is that the mapping language is inherently fragile. That is, minor coding errors can lead to major changes in the final specification. The consequences of such mapping errors are that it may become difficult or impossible to locate the original program errors, or that 'spurious' errors may be introduced. A number of solutions are proposed to overcome these difficulties, however future researchers would also be advised to draw on the experiences of the formal specification community. A library of common mappings could be provided so that the user is not always required to formulate a mapping prior to beginning the debug session. Also, where mappings have been defined for unusual decompositions or translations, a mechanism for storing these mappings in the library for future use could be provided. Map creation could also be facilitated, through the use of either a semi- or fully automated analysis system. A semi-automated system could provide a graphical environment for specifying maps, allowing the user to model and test the decomposition or transformation interactively. A fully automated system could perform code analysis on the serial and parallel codes to determine the required mapping.

ASSERTIONS

Assertions, like maps, are also implemented in a very basic manner, which raises a number of problems for the user. Despite its power, the comparison of data by specifying variable and breakpoint information in assertion statements is a very "low level" approach. It relies on the programmer making sensible choices about the location from which data will be extracted.

In order that data can be compared, it is not only important that the programs are functionally equivalent at this point, but that breakpoints aren't inserted at locations where the flow of control is disrupted or data is in an indeterminate state. This issue is particularly

important for arbitrary parallel programs incorporating distributed processes and/or multithreading and requires further investigation. Moreover, relative debugging cannot currently be applied to find timing errors, which are a common cause of failure in task parallel programs. In fact, the insertion of data gathering breakpoints, as required by a relative debugger, alters the timing of the programs and may mask or highlight temporal problems (the "probe effect" [25]). It might be possible to combine the assertion constructs used here with data gathering techniques that are not as invasive as the current debug server, however, this also requires further investigation.

A user currently formulates assertions by examining the programs under consideration and by following the data-flow of the variables manually. This information is then used to create the assertions. However, it would be much simpler and less error-prone to build assertions if an interactive browser could present the data-flow of a program graphically, and in particular show the define and use points for variables. Such a tool set would need to be integrated with some form of IDE.

VISUALISATION

Visualisation of data errors also presents a challenge to the debugger user. Currently the results of comparison operations are exported into an intermediate file. This data must then be transformed into a graphical format for visualisation. Although GUARD provides tools to facilitate this process, it is still relatively cumbersome and time consuming. The obvious approach would be to provide a visualisation engine in the debugger itself, perhaps as part of a GUI. Not only would this streamline the debugging cycle, it would give the visualisation engine access to data structure and mapping information that it would not normally have. This information could be used to enhance the display format, perhaps providing a "drill down" data hiding facility to simplify the display of large or complex data structures.

As discussed in Chapter 3, there is also some evidence to suggest that visualisation of comparisons, particularly using 2-D and 3-D representations, provides the user with a means of characterising patterns of differences. This has been highlighted in a number of previous studies [3,69] where characteristic patterns have been able to be used to identify specific types of behaviour and consequently the identification of sources of errors. While it is possible to make certain generalisations about the nature of the patterns in these situations, further research is still needed if this technique is to become the basis for formulating deductions about the nature of general classes of errors.

Complex Data Types

Currently, user-defined assertions, the data mapping algebra and the visualisation facilities provided by GUARD only support scalar data types and multi-dimensional arrays. AIF provides some support for structured data types, such as the C `struct` or the Pascal `record` types, but this is limited to simple manipulation and does not support linked lists or recursive data types.

Clearly there is scope to extend the relative debugging paradigm to more complex data types such as structures, lists and trees. However a number of issues still need to be addressed. For example, how exactly is a linked list compared? Does it mean that each element contains the same values, but the link pointers can differ? What if the linked list contains a pointer to an earlier element, how should this be compared? In addition, visualisation of these complex data types can also be difficult and time consuming. The utility of GUARD could be enhanced significantly from further research into these areas.

Conclusion

Relative debugging has introduced a new paradigm into the sphere of traditional debugging techniques. Unlike other debugging methods, relative debugging is language and machine independent, and works by comparing the divergence of key data structures in simultaneously executing programs. This allows programmers to focus on finding the location of an error rather than trying to understand how the program works. Relative debugging combines the unique ability to utilise the inherent correctness of a reference code for determining errors in a suspect code, with many of the tools found in traditional debuggers.

The work that has been undertaken demonstrates that this technique can be applied equally well to parallel programs. This was not obvious from the outset. Programs ported to parallel computers undergo significant modifications in order to take advantage of architectural features, often altering key data structures in complex ways. In addition, early versions of relative debuggers were limited to single process parallel codes due to inherent limitations of the implementations, so demonstrating the technique in real situations was difficult. It has only been after considerable effort was invested to achieve a number of key advances that the applicability of the technology to parallel codes has become apparent.

A range of key achievements has been accomplished in order to develop a parallel relative debugger. These achievements, culminating in the implementation of the GUARD debugger, include:

- the introduction of dataflow techniques as a method to control the relative debugging process;
- the development of a general purpose architecture independent data format that, unlike other data formats, can be used to perform in-core arithmetic, logical and comparison operations;
- the ability to describe data decomposition and transformations in a machine-independent manner, and for the debugger to use that information in performing data comparisons;
- the ability of the debugger to work with different parallel architectures including data parallel, shared memory and distributed memory architectures; and
- the extension of a client/server architecture to support the relative debugging technology.

The implementation of GUARD described in this thesis provides a robust platform for relative debugging, and with this as a basis, has extended the relative debugging technique to parallel computers. The effectiveness of this technology has been demonstrated by presenting three case studies. Each case study shows how relative debugging can be used to efficiently locate errors in a program that has been developed using one of three traditional parallel programming paradigms: data parallel, shared memory and distributed memory. The case studies show how important information can be derived from a range of sources, including:

- the location at which differences occur;
- in which data structures differences are visible; and
- the patterns that are observed in these differences.

All this information can be utilised to quickly identify a suspect code region using iterative refinement techniques. On this basis, it is clear that that this work has made a major contribution to the pool of debugging techniques that are available to aid programmers in the evolution of parallel codes.

Appendix A

GUARD USERS MANUAL

INTRODUCTION

GUARD is a relative debugger. Unlike other debuggers, GUARD allows programmers to compare data from programs as they are executing, and then use this information as an aid to identifying the location of errors.

GUARD is also a conventional interactive parallel debugger. It can be used to debug parallel and sequential programs using normal interactive commands, such as setting breakpoints and displaying program state. Because GUARD is a parallel debugger, it will allow commands to be applied to sets of parallel processes as well as a single sequential program.

GUARD's command set is derived from GDB, since this debugger is currently used to provide the low-level debug services. The syntax that GUARD uses has also been heavily influenced by the High Performance Debugging Forum's HPD standard.

DEFINITIONS & SYNTAX

Commands in GUARD consist of a keyword followed by an optional number of arguments, as in:

command [arg1] [arg2] ...

Arguments can consist of keywords or expressions.

Expressions are comprised of the usual C-like operators, constants, internal variables and program variables and are evaluated locally by the client. Text enclosed in quotes "" is evaluated as an expression on the server, *using a syntax defined language being debugged.*

GUARD supports the use of internal variables for storing temporary values. These variables are denoted by a dollar sign followed by a name, for example `$var`. Internal variables can be referenced before they are defined, but will always return `void`.

When programs are debugged by GUARD, the executing processes are managed using process sets. The set "`all`" contains all processes that are currently being debugged by

default. Users can create new sets that contain arbitrary combinations of processes that the user wishes to control as a group. A process set can be selected as the "current set" (using the "`focus`" command), and subsequent commands will be automatically applied to all processes in this set.

Variables in a target program are referenced using a syntax consisting of an optional process specifier and an identifier or server expression separated by "`::`". If no process is specified then all processes in the current set are used. For example to refer to the variable `var` in process `$s` the program variable `$s::var` would be used. If `var` was a pointer to a structure, then this must be dereferenced by evaluating a server-side expression, such as `$s::"var->field"`.

If an internal or target program variable is an array, then elements can be referenced using standard C array syntax, such as `$s::var[3][4]`. GUARD also supports the notion of slices. An array can be sliced by specifying an array index as a range, such as `$s::var[3..10][4]`.

USAGE

GUARD is a client/server debugger. The GUARD client must be started before the debugger can be used. The following command is used when starting the debugger under UNIX:

```
guard [program [args]]
```

If arguments are supplied to the guard command, a single program will be started under debugger control, much the same way as the GDB command would.

Once the debugger is started, the user will see an introductory message, followed by a command prompt:

```
GUARD-2000 Parallel Relative Debugger
Copyright (c) 1996-2000 by Monash University

dbg all>
```

At this point, the user can issue commands to the debugger.

PROCESS INVOCATION

Process identifiers are first class debugger variables. Processes are declared using the `invoke` command which specifies the name of the executable and the number of processes to create. The syntax of the `invoke` command is:

```
invoke <decl> "command" [on "host"] [using "arch"]
```

`<decl>` is a list of one or more internal debugger variables which can be either scalars or arrays of multiple dimensions. The number of processes started is calculated from the number of scalar variables combined with the number of elements in each array in the list.

The following example starts a sequential code called "`prog1`" on machine "`host1`":

```
invoke $ser "prog1" on "host1"
```

The process identifier `$ser` can then be used to refer to the sequential process. The next example starts a 5 process parallel code called "`par`" using the MPICH parallel architecture:

```
invoke $master,$slave[2][2] "par" using "mpich"
```

Parallel processes are assigned to the process identifiers in order, so `$master` refers to the first process, and the remaining processes are assigned to elements of the two-dimensional array `$slave[2][2]`.

COMPARISONS AND ASSERTIONS

Comparisons and assertions are commands specifically designed for relative debugging. The comparison command is used to compare data between two programs interactively. For example, if the processes `$proc1` and `$proc2` are stopped at a breakpoint, the user might issue the following command to compare variables `var1` and `var2`:

```
compare $proc1::var1 = $proc2::var2
```

If the display mode is set appropriately, then differences in these variables will be displayed immediately to the user.

Assertion commands are employed when the user wishes to define a set of conditions about the programs before they are executed. Assertions are encapsulated using the "`graph`" command, and take a special form of variable reference that includes a breakpoint reference consisting of a file name and line number. An assertion comparing the variable `var1` from

file1.c at line 34 of the process $p1 with the variable var2 from file2.c at line 55 of the process $p2 would take the form:

```
graph $g
  assert $p1::var1@"file1.c":34 = $p2::var2@"file2.c":55
end
```

Entering these commands in the debugger will create a graph called $g. Issuing the command "start $g" will then instruct the debugger to execute the graph. This results in breakpoints being inserted at the appropriate locations and then the processes started. When the breakpoints are reached, the debugger will extract the contents of the variables and perform the comparison.

DEBUGGER COMMANDS

The following list describes the commands available in GUARD. Letters shown in bold correspond to the abbreviation for the command.

Commands are classified into three groups. These are:

- *process independent* – the command does not interact with a process;
- *process targeted* – the command interacts with a process, and at least one process must have been invoked; and
- *process optional* – the command may interact with a process, depending on its arguments.

Immediate mode commands are executed as soon as they are issued, and generally result in communication with the server. The "set" and "info" sub-commands are also immediate mode commands. Deferred mode commands may only be issued when the debugger is in deferred mode, and are collected by the debugger and are only processed when the "end" command is encountered. The three commands: "graph", "map" and "trans" switch the debugger from immediate to deferred mode. The "end" command switches from deferred to immediate mode.

`assign target expr [-force]`

Process Optional. Assign the value of an expression defined by expr to target. The target must evaluate to an l-value that is either an internal variable or a program variable. If target is the name of an internal variable that does not exist, a new variable is created automatically.

`break [line|func|addr]`

Process Targeted. Set a breakpoint in all processes in the current process set. The location of the breakpoint can be defined as a line number line, a function name func or an address addr.

`compare expr1 = expr2`

Process Optional. Compare (subtract) the value of expression *expr1* with the value of expression *expr2*. The values must be the same size and shape. For complex data types such as arrays and structures, the comparison is performed on each element. The output of this command depends on the settings of the *display*, *output* and *error* variables.

`cont [n]`

Process Targeted. Continue execution of all processes in the current process set from a breakpoint. If *n* is specified, then the debugger will stop on the *n*th breakpoint.

`create $var[[dim1][dim2]...]`

Process Independent. Create an internal variable. If *dim1*, *dim2*, etc. are supplied then the internal variable will be an array of the specified rank and shape.

`defset pset p1 [p2 ...]`

Process Independent. Define a process set called *pset* containing the processes *p1*, *p2*, etc.

`delete [bp1 bp2 ...]`

Process Targeted. Delete the list of breakpoints specified by *bp1*, *bp2*, etc. in the current process set. If no arguments are specified then all breakpoints will be deleted.

`down [n]`

Process Targeted. Move down the call stack by *n* frames in all processes in the current process set. If no argument is supplied then move down one frame.

`focus [pset]`

Process Independent. Change focus to process set *pset*. Subsequent process targeted commands will refer to the processes in this set.

`function func([arg1,...]) = expr`

Process Independent. Define a function (macro) called *func*. The expression *expr* can contain format arguments that are substituted by actual arguments supplied to the function. Recursion is not allowed.

`graph name`

Process Independent. Define a dataflow graph called *name*. Switches to deferred mode until the end command is encountered.

`halt`

Process Targeted. Halt all executing processes in the current process set. This is the equivalent of sending a SIGINT to the process.

`help` *[command]*

Process Independent. Display help information on the command *command.* If no argument is supplied then a list of available commands is displayed.

`info` *sub-command*

Process Independent. Display information specified by *sub-command.*

`invoke` *var-list program [args] [on host [user passwd]] [using arch]*

Process Independent. Invoke a program called *program* with arguments *args.*

If `on` is supplied, then the program is executed on the remote system *host* optionally suppling *user* and *passwd* for authentication.

If `using` is supplied, the program is started under one of the parallel run-time systems specified. These can be `mpich`, `poe` or `p4`.

For sequential programs, *var-list* is just a scalar variable. For parallel programs, *var-list* must be a list of one or more scalar variables or arrays. Parallel process tasks are assigned to each variable or array in turn, starting from the left. The total of the number of scalars and the ranks of each array determine the number of processes started.

`kill`

Process Targeted. Kill all executing processes in the current process set.

`list` *[-|[file:]line|[file:]func|addr][,[file:]line|[file:]func|addr]*

Process Targeted. List lines of all processes in the current process set.

With no arguments, list ten more lines after the previous listing. If "-" is specified, list ten lines before the previous listing.

If one argument is supplied, this specifies a line around which ten lines are listed. If two arguments are supplied, these specify the starting and ending lines to list.

`map` *name* `(`*proc*`::`*var*`)`

Process Independent. Define a data transformation map called *name.* The mapping will be supplied with an array of processes *proc* and a distributed variable *var.* Switches to deferred mode until the end command is encountered.

`next` *[n]*

Process Targeted. Skip to next statement of each process in the current process set, ignoring subroutine calls. If *n* is supplied, skip next *n* statements.

`print expr`

Process Optional. Print the value of *expr* for each process in the current process set. *Expr* can contain any combination of internal variables and program variables.

`quit`

Process Independent. Quit the debugger.

`release prog`

Process Independent. Release an invoked program from the debugger. Shuts down the debug servers for all processes associated with the program and releases the internal variable *prog*.

`restart graph`

Process Independent. Restart a graph. All processes referenced by the graph are restarted first.

`run [args]`

Process Targeted. Start executing all processes in the current process set. *args* are passed to the program as command line arguments.

`set sub-command`

Process Independent. Set information specified by *sub-command*.

`source filename`

Process Independent. Read commands from the file specified by *filename*.

`start graph`

Process Independent. Start execution of the graph specified by *graph*.

`status`

Process Targeted. Print the status of all processes in the current process set.

`step [n]`

Process Targeted. Skip to next statement of each process in the current process set. Will step into subroutine calls. If *n* is supplied, skip next *n* statements.

`trans name(var)`

Process Independent. Define a data transformation function called *name*. The data structure to be transformed will passed as the argument *var*. Switches to deferred mode until the end command is encountered.

`up [n]`

Process Targeted. Move up the call stack by *n* frames in all processes in the current process set. If no argument is supplied then move up one frame.

`version`

Process Independent. Display the debugger version.

`viewset` *[pset]*

Process Independent. Display the processes associated with the process set *pset.* If no argument is supplied, display all process sets.

`whatis` *expr*

Process Targeted. Display the data type of the expression *expr* for each process in the current process set.

`where` *[n]*

Process Targeted. Display the current stack frames for all processes in the current process set. If *n* is supplied, only the innermost *n* frames are displayed. If *n* is negative the outermost *n* frames are displayed.

SET SUB-COMMANDS

`set async = on|off`

Default `on`. Turn asynchronous mode on or off. This only affects interactive mode, and determines if the debugger waits for each command to complete before accepting additional commands.

`set dfmode = default|sync|async`

Set the dataflow mode to synchronous or asynchronous. This changes the dataflow graph that is generated using the graph command so that it operates in one of these modes. The synchronous mode is default for assertions that refer to independent processes, while the asynchronous mode is the default for assertions that refer to the same process.

`set display = text|aif|hdf|hdf5|ms`

Default `text`. Set the display format for assertions. In `text` mode, any differences are displayed as text at the users terminal. In `aif` mode, differences are output in architecture independent format (AIF). In `hdf` and `hdf5` modes, differences are output in the hierarchical data format (HDF) version 4 or version 5 respectively. In `ms` mode, the differences are displayed in a GUI environment running under Microsoft Windows (experimental).

`set error` *lower upper* `relative|absolute`

Default "`0.0 0.0 absolute`". Set the error limits for assertions. Differences below *lower* are ignored. Differences between *lower* and *upper* are displayed using the current `display` mode. Any differences above *upper* will halt the dataflow interpreter. The `relative` and `absolute` modes are used to specify if *lower* and *upper* are relative or absolute limits respectively. In `relative` mode, the difference is first divided by the maximum of the two values being compared.

`set ERROR_CHECKS = max|normal|min`

Default `normal`. Sets error checking mode to `max`, `normal` or `min`. This determines if the debugger prompts before a potentially dangerous action is taken, prompts only before a potentially fatal action or warns otherwise, or ignores potential danger respectively.

`set EVENT_INTERRUPT = on|off`

Default `on`. Determines whether program events are detected and reported to the user as soon as possible, or deferred until the user has completed typing a command.

`set EXECUTABLE_PATH = path`

Default ". /:$PATH". In order to find the location of an executable specified using the `invoke` command, the debugger searches the current directory, then all the directories included in the executable search path.

`set force = on|off`

Default off. Determines whether subsequent assertions will copy the value from the LHS variable into the RHS variable after the assertion is executed.

`set halt = on|off`

Default on. Determines if processes are halted when the dataflow interpreter terminates.

`set logfile = filename`

Default empty. Specifies the name of a file to send logging entries.

`set logging = on|off`

Default off. Turns dataflow event logging on or off.

`set MAX_LIST = lines`

Default 20. Specifies the number of lines that are displayed using the `list` command.

`set MAX_LEVELS = levels`

Default 20. Specifies the maximum number of call stack frams that are displayed using the `where` command.

`set MAX_HISTORY = length`

Default 20. Specifies the length of the command history that will be maintained.

`set MAX_PROMPT = length`

Default 20. Specifies the maximum length of the command prompt.

`set MODE = processes|threads|multilevel`

Default `processes`. Defines if the debugger is capable of debugging multiple-process single thread, single process multiple-thread or multiple-process multiple-thread programs respectively. Currently only the `processes` mode is supported.

`set output = filename`

Default empty. If defined, specifies a file to which output from assertions is sent.

`set PROMPT = prompt`

Default "`guard $ptset> `". Sets the current prompt.

`set SOURCE_PATH = path`

Default empty. Defines where the debugger will look for source files associated with the executable.

`set VERBOSE = err|warn|all`

Default `warn`. Defines the level of debugger diagnostics. These levels are `err`, `warn` and `all`, which correspond to show error messages only, show warning messages and normal command output and show maximum information respectively.

INFO SUB-COMMANDS

Each `set` sub-command has a corresponding info sub-command to display the current setting. In addition, the following `info` sub-commands are available.

`info globals`

> *Process Targeted.* Display a list of the global symbols for each process in the current process set.

`info map` *[name]*

> *Process Independent.* Display information about the definition of the map *name*. If no argument is supplied, information on all maps is displayed.

`info process` *[pset]*

> *Process Independent.* Display information about the process set *pset*. If no argument is supplied, information on all process sets is displayed.

`info trans` *[name]*

> *Process Independent.* Display information about the definition of the translation *name*. If no argument is supplied, information on all translations is displayed.

DEFERRED MODE COMMANDS

`assert var1 =/== var2`

> *Assert command only.* Assert that variable *var1* is equivalent to variable *var2*. Variables are specified in the format *proc*`::`*var*`@`*file*`:`*line* to denote a variable called *var* at line number *line* in file *file* from executing process *proc*.

`define index(a,i,u,l) = expr`

> *Map command only.* Define a function that describes the mapping of indices from a serial array to a parallel arrays. The argument `i` is the current index value, `a` is an array of index values, `u` is an array of index upper bounds and `l` is an array of index lower bounds.

`define proc(a,i,u,l) = expr`

> *Map command only.* Define a function that describes the mapping of indices from a serial array to set of processes. The argument `i` is the current index value, `a` is an array of index values, `u` is an array of index upper bounds and `l` is an array of index lower bounds.

`define func(i) = expr`

> *Trans command only.* Define a function that describes translation of elements of a vector. The argument `i` is the current index value.

`end`

> *All commands.* Finish deferred mode command.

Appendix B

DEBUG CLIENT API ROUTINES

The following routines make up the debug client API library. Each routine returns a value that depends on its function, but is generally an integer or a pointer. For routines returning an integer, an error is indicated by a value of –1. For routines returning a pointer, a NULL pointer indicates an error. The API provides the `DbgError()` and `DbgErrorStr()` routines to find out more detail about the exact error that occurred.

Most routines operate synchronously, with the exception of `DbgGo()` and `DbgStep()` which operate asynchronously. Prior to calling the asynchronous routines, the appropriate handlers must be installed using the `DbgHandle*()` routines.

DEBUGGER INITIALISATION

These routines are used to perform the necessary initialisation and shutdown operations for the client/server protocol.

```
int DbgInit(char *proto)
```

> Initialise the debugger. Must be called once prior to any other API routine. The protocol that is used between debug client and server is specified using the `proto` argument. Allowable values "`socket`" and "`rpc`". Returns 0 on success or -1 on failure.

```
int DbgFinish(void)
```

> Terminate the debugger. Must be called once after all calls to other API routines. Returns 0 on success or -1 on failure.

PROCESS MANAGEMENT

These routines are available to manage processes in the debugger. This includes routines for creating and removing, controlling the execution of, and finding the status of processes under the control of the debugger. When a process is created, a process handle is returned. This handle is then used to refer to the process in any subsequent operations.

```
void *DbgInvoke( char *prog, char **args, int nproc, char *user,
                 char *passwd, char *host_or_arch, char **env,
                 char *backend)
```

Invoke a program to be debugged. The program name and command line arguments are supplied as `prog` and `args` respectively. For parallel programs, `nproc` specifies the number of processes to start, and `host_or_arch` is used to determine the parallel architecture to use. For sequential programs `nproc` is set to 1 and `host_or_arch` can be used to specify the name of a remote host on which the program will be started, or `NULL` for the local machine. The `user` and `passwd` arguments can be used to supply authentication information if it is required. The `env` argument is used to pass local environment information to the remote processes. The `backend` argument is used to select a backend debugger. Currently only "gdb" is supported. On success a process handle is returned which can then be used by other routines to refer to this process. On failure, a null pointer is returned.

```
void *DbgProc(void *proc, int pnum)
```

If the process handle `proc` refers to a parallel process, this routine will return a handle to the process identified by `pnum`. Returns a process handle on success, or a null pointer on failure.

```
dyad_t DbgIsActive(void *proc)
```

Returns true if the process referred to by process handle `proc` has a server and is able to receive commands.

```
int DbgKill(void *proc)
```

Kill the process referred to by the process handle `proc`. After this command, the process is no longer being debugged, though the server is still running. Returns 1 on success or 0 on failure.

```
int DbgGo(void *proc)
```

Start the process referred to by the process handle `proc` running. This routine can be used to initiate execution (as in the "run" command) or continue execution after a breakpoint (as in the "cont" command). Returns 0 on success or -1 on failure.

```
int DbgStep(void *proc, int count, int step_in)
```

Single step the process referred to by the process handle `proc` one source line. If `count` is greater than one, then `count-1` lines will be skipped. If `step_in` is non-zero then the command will step into subroutine calls. Otherwise subroutine calls will be treated like a single statement. Returns 0 on success or -1 on failure.

```
int DbgInterrupt(void *proc)
```

Interrupt the running process referred to by the process handle `proc`. Returns the process to a state ready to accept a new command. If the process is not running then it will have no effect. Returns 0 on success or -1 on failure.

```
int DbgQuit(void *proc)
```

Shut down the server debugging the process referred to by `proc`. Returns 0 on success or -1 on failure.

```
int DbgQuitAll(void)
```

Shut down all servers in one operation. This is equivalent to issuing `DbgQuit()` commands for each process being debugged. Returns 0 if all server shut down successfully, or -1 otherwise.

DEBUG OPERATION

These routines are used to perform a range of operations on processes being debugged. The first argument of all these commands is a process handle that refers to the process on which the operation is to be performed.

```
char *DbgGetType(void *proc, char *expr)
```

The expression `expr` is evaluated by the process referred to in `proc`. If successful, an AIF type descriptor for the result is returned. If an error occurs, NULL is returned.

```
AIF *DbgEvalExpr(void *proc, char *expr)
```

The expression `expr` is evaluated by the process referred to in `proc`. If successful, the resulting value is converted to AIF and returned. A null pointer is returned on error.

```
int DbgSetVar(void *proc, char *var, AIF *val)
```

Sets the program variable `var` to the value represented by `val`. The routine attempts to convert `val` if its type is different from `var` or it contains a value that is not representable on the remote host. Returns 0 if successful or -1 on failure.

```
dbgbp_t *DbgSetLineBreak(void *proc, char *file, int line)
```

Set a breakpoint at line number `line` in the source file `file` of the process `proc`. If successful, returns a structure containing information about the breakpoint. On failure, returns a null pointer.

```
dbgbp_t *DbgSetFuncBreak(void *proc, char *file, char *func)
```

Set a breakpoint at the first line of the function (or subroutine) `func` in the source file `file` of the process `proc`. If successful, returns a structure containing information about the breakpoint. On failure, returns a null pointer.

```
int DbgDeleteBreak(void *proc, int bpid)
```

Delete the breakpoint identified by `bpid`. Breakpoints are managed on a per-process basis. The breakpoint identifier is available in the `bp_id` field of the `dbgbp_t` structure. Returns 0 on success or -1 on failure.

```
dbgbp_t *DbgShowBreak(void *proc)
```

Lists all the breakpoints set in the process `proc`. If successful, returns a linked list of breakpoint information. Returns a null pointer on failure.

```
dbglist_t *DbgList(void *proc, char *loc)
```

List the source lines of the current file for process `proc`. If successful, returns a linked list of source lines. If loc is empty or refers to a single location 10 lines are returned. If loc is a range, returns all lines in the range. Returns a null pointer on error.

```
int DbgSetArgs(void *proc, char *args)
```

Sets the command line arguments that are passed to the executable when it is first started using the `DbgGo()` routine. Returns 0 if successful or –1 on error.

```
dbgframe_t *DbgMoveFrame(void *proc, int count, int dir)
```

Move the current call stack frame. If `count` is greater than 1, specifies the number of stack frames to move. The parameter `dir` sets the direction of movement. If `dir` is set to 1 the routine will move up the call stack, otherwise the routine will move down the call stack. On success, returns a structure describing the current stack frame. Returns a null pointer on failure.

```
dbgframe_t *DbgShowFrame(void *proc)
```

Show each frame of the current call stack for the process `proc`. On success, returns a linked list of call stack frames. Returns a null pointer on error.

```
dbgloc_t *DbgGetLocation(void *proc)
```

Find the current breakpoint location for the process `proc`. On success, the `dbgloc_t` structure provides the location information. Returns a null pointer on error.

```
dbgvars_t *DbgGetVars(void *proc, char *file)
```

Returns the global symbols that are defined in the source file `file`. On success a linked list of `dbgvars_t` structures is returned containing the symbol information. Returns a null pointer on error.

NOTIFICATION

These routines are used to manage the asynchronous activities of the debugger. The first two routines are normally called from within an event loop to manage the processing of debugger events. The remaining routines are used to specify handler functions for specific types of events that occur in the debugger.

```
int DbgWaitEvent(void *proc, int file, sigset_t *mask, int noblock)
```

Wait for an event to occur. This routine is provided for compatibility with debuggers that use a command-line interface. The `file` argument is the file descriptor that user commands will be read from, or –1 if none. The `mask` argument is used to specify a set of signals that will be masked during the call. If `noblock` is set to 1 then the routine can be used to poll for new events.

```
int DbgWaitBreak(void *proc, sigset_t *mask)
```

Wait for a breakpoint to occur. Similar to the `WaitForEvent()` routine, but specifically waits for a breakpoint to be reached. Must only be used after a `DbgStep()` or `DbgGo()` command has been issued.

```
void DbgHandleInput(void *proc, void (*)()handler, void *arg)
```

Handle input related events. This routine is used to set a handler for events that result from the program being debugged requesting input from stdin. The handler routine is set using the `handler` parameter. Any handler specific information provided using the `arg` parameter will be passed to the handler when it is called.

```
void DbgHandleOutput(void *proc, void (*)()handler, void *arg)
```

Handle output related events. This routine is used to set a handler for events that result from the program being debugged sending output to stdout. The handler routine is set using the `handler` parameter. Any handler specific information provided using the `arg` parameter will be passed to the handler when it is called.

```
void DbgHandleSignal(int signal, void (*)()handler)
```

Handle signals. This routine is used to set a handler for signals that are sent to the client. Using this routine rather than the standard signal routines ensures that the debugger will manage all signals properly.

```
void DbgHandleEvent(void *proc, dyad_t (*)()handler, void *arg)
```

Handle asynchronous events. This routine is used to set a handler for events that result from issuing commands that operate asynchronously. The handler routine is set using the `handler` parameter. Any handler specific information provided using the `arg` parameter would be passed to the handler when it is called.

STATUS

These commands are used to find status information about the debugger or the last debug operation performed.

```
dbgstat DbgStatus(void *proc)
```

Obtain the status of the debug server.

```
void DbgError(void)
```

Print a description of the last error.

```
char *DbgErrorStr(dbgevent_t *ev)
```

Return a string containing the last error associated with the event `ev`.

Appendix C

CLIENT DATA TYPES

`dbgbp_t` A structure containing information relating to a breakpoint. Includes the following fields:

`bp_id`	breakpoint identifier
`bp_type`	type of this breakpoint
`bp_location`	location of this breakpoint (dbgloc_t)
`bp_hits`	number of times this breakpoint has been reached
`bp_stmt`	source line at breakpoint
`bp_cmds`	list of commands associated with this breakpoint

`dbglist_t` A structure used to specify a linked list of line numbers and corresponding source lines. Includes the following fields:

`list_lno`	the line number of this source line
`list_line`	the source line

`dbgframe_t` A structure used to specify a linked list of stack frames. Includes the following fields:

`frame_level`	the level of this stack frame
`frame_loc`	the stack frame information

`dbgloc_t` A structure representing a location in a source file. Consists of the following fields:

`loc_file`	a source file name
`loc_func`	a function name
`loc_addr`	an address
`loc_line`	a line number

`dbgvars_t` A structure used to represent a linked list of variable names. Contains only one field:

`var_name`	name of the variable

`dbgsig_t` A structure containing information relating to a signal event. Includes the following fields:

`sig_type`	the signal type
`sig_func`	the current source line
`sig_addr`	the current stack frame

`dbgstep_t` A structure containing information returned after a single step operation. Consists of the following fields:

`step_lno`	the current line number
`step_line`	the current source line
`step_frame`	the current stack frame

`dbgerr_t` A structure containing information relating to the last error that occurred. Contains the following fields:

`err_str`	a description of the error
`err_errno`	the error number

`dbgevent_t` A structure representing and event that has just occurred. An event consists of a type and the data associated with the event.

`dbgstat` The status of the debugger. Can be one of the following values:

`DBGSTAT_INITIALISING`	the debugger is initialising
`DBGSTAT_WAITING`	the debugger is waiting for a command
`DBGSTAT_RUNNING`	the process is running
`DBGSTAT_STOPPED`	the process is stopped
`DBGSTAT_DBGERR`	an error occurred
`DBGSTAT_INTERR`	an internal debugger error occurred

`AIF` Data that has been converted to the architecture independent format.

Appendix D

DEBUG SERVER API

The debug server API library is divided into two parts: stub routines and utility routines. The server stub routines are called when the server decodes a client debug request. The debugger backend must provide a series of routines that match up with each of the stubs, and that perform the appropriate debug operations. The utility routines are used by the debugger backend to aid in the processing of debug requests.

STUB ROUTINES

Stub routine arguments are obtained by decoding the client request and extracting the argument information. Each stub routine returns a debugger event of type `dbgevent_t *`. The contents of this event must be generated by the associated backend routine. The following is a list of stub routines, their associated arguments and a description of the function the routine must perform.

```
dbgevent_t *DbgDeleteBreak(int bp)
```

Delete the breakpoint identified by `bp`. Breakpoints are numbered starting from 1, so this value must be mapped to the internal breakpoint identifier used by the backend debugger.

```
dbgevent_t *DbgEvalExpr(char *expr)
```

Evaluate the expression specified in `expr` and return the result in AIF format.

```
dbgevent_t *DbgGetLocation(void)
```

Return the current breakpoint location for the process. A location consists of a source file name, and a line number, function name or address.

```
dbgevent_t *DbgGetType(char *var)
```

Obtain type information for the named object `var`. Currently returns the AIF type description.

```
dbgevent_t *DbgGetVars(char *file)
```

Returns a list of the global symbols that are defined in the source file `file`.

```
dbgevent_t *DbgGo(char *host, int cb)
```

Start the process running. The combination of `host` and `cb` are used to provide a protocol specific communication point for event callback.

```
dbgevent_t *DbgInterrupt(char *host, int cb)
```

Send an interrupt signal to the current process. The combination of `host` and `cb` are used to provide a protocol specific communication point for event callback.

```
dbgevent_t *DbgKill(void)
```

Terminate the process being debugged. The server remains operational after this command.

```
dbgevent_t *DbgLastBreak(void)
```

Return breakpoint number of the last breakpoint reached.

```
dbgevent_t *DbgList(char *str)
```

List the source lines of the current file. The argument `str` specifies the range of lines to list.

```
dbgevent_t *DbgMoveFrame(int count, int up)
```

Move up or down the call stack by the number of frames specified in `count`. If the argument up is non-zero, then the direction is towards the calling frame.

```
dbgevent_t *DbgQuit(void)
```

Shut down the server.

```
dbgevent_t *DbgSetArgs(char *args)
```

Set the arguments that will be passed to the process the next time it is run.

```
dbgevent_t *DbgSetLineBreak(char *file, int line)
```

Set a breakpoint at line number `line` in the source file `file`.

```
dbgevent_t *DbgSetFuncBreak(char *file, char *func)
```

Set a breakpoint at the first line of the function (or subroutine) `func` in the source file `file`

```
dbgevent_t *DbgSetVar(char *var, AIF *data)
```

Set the variable identified by var to the value specified by data. Since data is in AIF format, it must first be converted to the target format before this operation can be completed.

```
dbgevent_t *DbgShowFrame(void)
```

Return a list of the current call stack frames.

```
dbgevent_t *DbgStep(int count, int in, char *host, int cb)
```

Single step the process. The argument `count` specifies how many breakpoints to skip. If the argument `in` is non-zero then the process will step into the next subroutine call. If `in` is zero then the process will step over the next subroutine call. The combination of `host` and `cb` are used to provide a protocol specific communication point for event callback.

```
dbgevent_t *DbgStatus(void)
```

Obtain the status of the debug server.

UTILITY ROUTINES

The following routines provide a number of miscellaneous services for use by the debug server backend routines.

```
int DbgReadyForCmd(void)
```

Check if the debugger is in the correct state to accept a new command. Returns non-zero if commands are ready to be accepted.

```
int DbgStopped(void)
```

Check if the process being debugged is in the stopped state. Returns non-zero if the process is stopped.

```
dbgstat GetStatus(void)
```

Get the current status of the debugger. Possible values are: `DBGSTAT_INITIALISING`, `DBGSTAT_WAITING`, `DBGSTAT_RUNNING`, `DBGSTAT_STOPPED`, `DBGSTAT_DBGERR`, and `DBGSTAT_INTERR`.

```
void SetStatus(dbgstat stat)
```

Set the debugger status to the value of `stat`.

```
void ResetWDT(void)
```

Reset the debugger watchdog timer. If this timer expires, the debug server will automatically shut down. This routine must be called regularly to prevent this from happening.

```
void Shutdown(void)
```

Shut the server down. This routine is called once the process being debugged has terminated and all cleanup actions have been taken.

```
void AsyncCheck(dbgevent_t *(*rtn)(), dbgevent_t *event,
                char *host, int prog)
```

Cause an event to be sent to the client asynchronously. At some point, the routine `rtn` will be called with `event` supplied as its argument. The routine must return an event, which will be sent back to the client. The arguments `host` and `prog` specify the client callback communication point. If `rtn` is `ASYNC_FORCE`, the argument `event` will be sent directly to the client instead.

Appendix E

ARCHITECTURE INDEPENDENT FORMAT API

The following routines make up the AIF API library. Each routine returns a value that depends on its function, but is generally an integer, a pointer to an AIF object or some other type of pointer. Routines returning an integer normally do so to indicate the success or failure of the operation. In these routines an error is indicated by a value of –1. For routines returning a pointer, a NULL pointer indicates an error. The API provides the `AIFError()` and `AIFErrorStr()` routines to find out more detail about the exact error that occurred. Routines that return a pointer to an AIF object automatically allocate memory for that object. It is the callers responsibility to release memory for the arguments *and* the result using the `AIFFree()` routine.

CONVERSION

Conversion routines are used to convert data in target format to and from the architecture independent format. These routines require that the caller knows the type of the object being converted.

```
AIF *IntToAIF(int val)
```

Convert an integer in target format into an AIF object.

```
AIF *FloatToAIF(float val)
```

Convert a single precision floating point into an AIF object.

```
AIF *DoubleToAIF(double val)
```

Convert a double precision floating point into an AIF object.

```
AIF *VoidToAIF(char *str, int len)
```

Convert a byte string of length `len` into an AIF object of type VOID.

```
AIF *ArrayToAIF(int rank, int *min, int *max,
                char *data, char *base)
```

Convert the array pointed to by `data` into an AIF object. The argument `rank` specifies the rank of the array, `min` and `max` are arrays containing the minimum and maximum value of each dimension respectively, and `base` is the base type (element type) of the array.

```
int AIFToInt(AIF *obj, int *ret)
```

Convert the AIF object `obj` into an integer in target format. The value is returned into the location pointed to by `ret`.

```
int  AIFToFloat(AIF *obj, float *ret)
```

Convert the AIF object `obj` into a single precision floating-point value in target format. The value is returned into the location pointed to by `ret`.

```
int AIFToDouble(AIF *obj, double *ret)
```

Convert the AIF object `obj` into a double precision floating-point value in target format. The value is returned into the location pointed to by `ret`.

```
int AIFToVoid(AIF *obj, char *data, int len)
```

Convert the AIF object `obj` into a byte string of length `len`. The result is stored in the location pointed to by `data`.

```
char *AIFIntToStr(AIF *obj, int base)
```

Convert the integer AIF object `obj` into its string representation using the base supplied. The base must be 16 or less.

```
char *AIFFloatToStr(AIF *obj)
```

Convert the floating point AIF object `obj` into its string representation.

```
AIF *AIFCoerce(AIF *obj, char *type)
```

Coerce the AIF object `obj` into the type specified by the argument `type`.

ARITHMETIC & LOGICAL

Arithmetic and logical routines can be applied to any AIF types that are type compatible, and will perform any type conversions necessary. Currently, the routines support arguments of complex types array and structure, however both arguments in binary operations must be of the same type (including the size, shape and number of fields). For these objects, the routines will perform element-wise or field-wise operations. The following table shows the result type after performing an arithmetic operation.

obj1	obj2	result
integer	integer	integer
floating	integer	floating
integer	floating	floating
array	array	array
structure	structure	structure

```
AIF *AIFAdd(AIF *obj1, AIF *obj2)
```

Add the AIF object `obj1` to the AIF object `obj2` and return the result.

```
AIF *AIFNeg(AIF *obj)
```

Returns the arithmetic negation of `obj`.

```
AIF *AIFSub(AIF *obj1, AIF *obj2)
```

Subtract the AIF object `obj2` from the `obj1` and return the result

```
AIF *AIFMul(AIF *obj1, AIF *obj1)
```

Multiply the AIF object `obj1` by `obj2` and return the result.

```
AIF *AIFDiv(AIF *obj1, AIF *obj2)
```

Divide the AIF object `obj1` by `obj2` and return the result.

```
AIF *AIFRem(AIF *obj1, AIF *obj2)
```

Divide the AIF object `obj1` by `obj2` and return the remainder as the result.

```
AIF *AIFNot(AIF *obj)
```

Returns the logical COMPLEMENT of the AIF object `obj`.

```
AIF *AIFAnd(AIF *obj1, AIF *obj2)
```

Returns the logical AND of the AIF objects `obj1` and `obj2`.

```
AIF *AIFOr(AIF *obj1, AIF *obj2)
```

Returns the logical OR of the AIF objects `obj1` and `obj2`.

COMPARISON

These routines are used to perform comparisons between AIF objects. Currently, the comparison routines can only be applied to the numeric types integer and floating. The complex types array and structure are supported, and in these cases the comparison operations will be performed on an element-wise or field-wise basis.

```
int AIFIsZero(AIF *obj, int *res)
```

Test of the AIF object is equivalent to zero. Returns 1 if true, otherwise 0. The return value is placed in the variable pointed to by `res`.

```
int AIFCompare(AIF *obj1, AIF *obj2, int *res)
```

Compare the AIF objects `obj1` and `obj2`. Returns –1 if `obj1` < `obj2`, 0 if `obj1` = `obj2` and 1 if `obj1` > `obj2`. The return value is placed in the variable pointed to by `res`.

```
dyad_t AIFTypeCompare(AIF *obj1, AIF *obj2)
```

Compare the types of AIF objects `obj1` and `obj2`. Returns true if they are equivalent, false otherwise.

```
int AIFEPS(double lower, double upper, AIF *obj, int *res)
```

Compute the *epsilon value* for the AIF object `obj`. The return value, which is placed in the variable pointed to by `res`, is shown below.

Arguments	Result
obj < lower	-1
lower <= obj <= upper	0
obj > upper	1

ARRAY

The following routines are provided for performing operations specifically on AIF objects that represent arrays. AIF arrays are characterised by their *rank* or number of dimensions. An array has a *base type* that specifies the type of each element of the array. Arrays also have *bounds*, which specify the upper and lower limits of each index of the array. Routines are also provided to iterate over the indices of an array. This is done with an *index counter*, which is an array of integers representing the current value of each index of the array.

```
int AIFArrayRank(AIF *obj)
```

Returns the rank of the AIF array object `obj`.

```
int AIFArraySize(AIF *obj)
```

Returns the number of elements in the AIF array object `obj`.

```
int AIFArrayBounds(AIF *obj, int rank, int **min,
                   int **max, int **size)
```

Give the AIF array object `obj` of rank `rank`, return the minimum index value, maximum index value, and if size is not NULL, the size of each dimension of the array.

```
int AIFArrayInfo(AIF *obj, int *rank, char **type, int *itype)
```

Find out information about the AIF array object `obj`. Returns the number of dimensions in `rank` and the element type in `type`. If `itype` is not NULL, returns an array containing the type of each index.

```
AIF *AIFArraySlice(AIF *obj, int rank, int *min, int *max)
```

Perform an array slice operation on the AIF array object `obj`. The rank of the array must be supplied in `rank`, and the minimum and maximum index values of each slice in `min` and `max` respectively.

```
int AIFArrayMinIndex(AIF *obj, int n)
```

Find the minimum index value of the n^{th} array index of the AIF array object `obj`.

```
int AIFArrayMaxIndex(AIF *obj, int)
```

Find the maximum index value of the n^{th} array index of the AIF array object `obj`.

```
char *AIFArrayIndexType(AIF *obj)
```

Find the type of the index of the AIF array object `obj`.

```
AIFIndex *AIFArrayIndexInit(AIF *obj)
```

Create an index counter object and initialise it with the minimum index values from the AIF array object `obj`.

```
int AIFArrayIndexInRange( int rank, int *index, int *min,
                          int *max)
```

Check that the value of each index in the array `index` is within the range specified by `min` and `max`. Returns 1 if they are, 0 otherwise.

```
int AIFArrayIndexInc(AIFIndex *cnt)
```

Increment the index counter object `cnt`.

```
void AIFArrayIndexFree(AIFIndex *cnt)
```

Free the memory allocated for the index counter object `cnt`.

```
AIF *AIFArrayElement(AIF *obj, AIFIndex *cnt)
```

Return the element of the AIF array object `obj` referenced by the index counter object `cnt`.

```
int AIFArrayElementToDouble(AIF *obj, AIFIndex *cnt,
                            double *res)
```

Convert the element of the AIF array object `obj` referenced by the index counter object `cnt` to a double precision floating point in target format.

```
int AIFArrayElementToInt(AIF *obj, AIFIndex *cnt, int *res)
```

Convert the element of the AIF array object `obj` referenced by the index counter object `cnt` to an integer in target format.

```
int AIFSetArrayData(AIF *obj, AIFIndex *cnt, AIF *val)
```

Store the AIF object val in the element of the AIF array object `obj` referenced by the index counter object `cnt`.

STRUCTURE

The following routines are provided for performing operations specifically on AIF objects that represent structures. AIF structures are characterised by a series of *fields*, each of which is used to store an AIF object.

```
int AIFNumFields(AIF *obj)
```

Return the number of fields in the AIF structure object `obj`.

```
int AIFFieldType(AIF *obj, char *name)
```

Return the type of a field of the AIF structure object `obj`. The field is specified by the argument `name`.

```
int AIFFieldToInt(AIF *obj, char *name, int *res)
```

Convert the field name of the AIF structure object `obj` to an integer in target format. The return value is placed in the variable pointed to by `res`.

```
int AIFFieldToDouble(AIF *obj, char *name, double *res)
```

Convert the field name of the AIF structure object `obj` to a double precision floating point in target format. The return value is placed in the variable pointed to by `res`.

INPUT/OUTPUT

AIF objects can be stored in a persistent file, called a *data set file* using the `AIFOpenSet()`, `AIFWriteSet()` and `AIFReadSet()` routines. These routines allow multiple AIF

objects to be store in one file, each object being tagged with a character string for identification. A routine is also provided to enable AIF objects to be displayed.

```
int AIFCloseSet(AIFFILE *fp)
```

Close the data set file referred to by `fp`.

```
AIFFILE *AIFOpenSet(char *file, int mode)
```

Open a data set file called `file` and return a pointer to the open file. The argument `mode` is a bitwise-inclusive-OR of the values `AIFMODE_READ`, `AIFMODE_CREATE` and `AIFMODE_APPEND`.

```
AIF *AIFReadSet(AIFFILE *fp, char **tag)
```

Read an AIF object from the data set file referenced by `fp`. The object tag is returned in the string `tag`.

```
int AIFWriteSet(AIFFILE *fp, AIF *obj, char *tag)
```

Write the AIF object `obj` to the data set file referenced by `fp`. The object is tagged with the string `tag`.

```
void AIFPrint(FILE *fp, AIF *obj)
```

Print a string representation of the AIF object `obj` to the standart I/O stream `fp`.

UTILITY

The following are utility routines for performing various functions on AIF objects.

```
aiferr AIFError(void)
```

Return the last AIF error that occurred.

```
char *AIFErrorStr(void)
```

Return the last AIF error that occurred as a printable string.

```
void AIFFree(AIF *obj)
```

Free the memory allocated for the AIF object `obj`.

```
int AIFType(AIF *obj)
```

Returns a integer indicating the type of the AIF object `obj`. Possible types are: `AIF_INTEGER`, `AIF_FLOATING`, `AIF_POINTER`, `AIF_ARRAY`, `AIF_STRUCT`, `AIF_FUNCTION`, `AIF_COMPOUND`, `AIF_VOID`, and `AIF_REGION`.

```
int  AIFBaseType(AIF *obj)
```

Returns the base type of the AIF object `obj`, if the object is a complex type. Otherwise returns the type of the object.

```
long  AIFTypeSize(AIF *obj)
```

Returns the size in bytes of the AIF object `obj`.

```
AIF *CopyAIF(AIF *obj)
```

Create a copy of the AIF object `obj`.

Appendix F

SURVEY OF DEBUGGERS: 1969 – 1999

PARALLEL/DISTRIBUTED DEBUGGERS

Name	Technology	Lang. / Arch.	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
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
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	CrOS NCUBE	Flower J., Williams R.	1988
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

Name	Technology	Lang. / Arch.	Authors	Date
Voycur	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
Pdeb	shared memory parallel		Zifrony D., Averbuch A.	1989
	distributed	Modula-2, C/TUMULT	Scholten J., Jansen PG., Posthuma J.	1990
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 B., Charrier P.	1992
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

Name	Technology	Lang. / Arch.	Authors	Date
Panorama	retargetable, extensible		May J., Berman F.	1993
PDG	process-level debugger for concurrent programs, animating, hierarchical graphical representations	GRAPE	Caerts C., Lauwereins R., Peperstraete JA.	1993
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
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., Meiming S.	1996
dbxR	replay	PVM	Miei T., Takahashi N.	1996
DDB	distributed, replay		Sienkiewicz J., Radhakrishnan T.	1996

Name	Technology	Lang. / Arch.	Authors	Date
	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
dect	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, break-point setting, process monitoring, error localization, and error fixing	CHILL	Paik EH., Byun YJ., Chung YS., Lee BS.	1998
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 performance evaluation tool		Lumpp JE Jr., Sivakumar K., Diaz C., Griffioen JN.	1998
	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

SERIAL DEBUGGERS

Name	Technology	Lang. / Arch.	Authors	Date
DYDE	on-line, symbolic	assembly	Josephs WH.	1969
PEBUG	interactive and non-interactive	CDC-6500	Blair JC.	1971
DDS	monitor, breakpoints	assembly	North S.	1977
MODS	clean user interface		Hawrylak JJ.	1977
TOADC	interactve/batch, high-level	TRIDENT	Gaines JA Jr.	1978
AIDS	symbolic, interactive		Hart JJ.	1979
ALADDIN	breakpoint assertions	assembly	Fairley RE.	1979
DEB-2	batch	assembly	Lanzarone GA.	1979
ISSP	interactive, top-down	ROMTRAN	Andreussi G., Salza S.	1979
ADVISOR	AI	Fortran	Isomoto Y., Yamagata K., Ishiketa T.	1980
SYMbug	symbolic, multi-language		Duyck R.	1980
ZSID	debugger	CPM/Zilog	Miller AR.	1980
	high-level, multi-language	PL/1, Fortran, BASIC	Elliott B.	1982
	step-wise debugging		Hamlet D.	1983
	hardware support		Abramson D., Rosenberg J.	1983
DELTA	symbolic	CP-6	Walter CK.	1983
DICE	integrated programming environment, incremental compiler		Fritzson P.	1983
joff	source-level, GUI	C	Cargill TA.	1983
SWAT	high-level symbolic	C, Pascal/AOS	Cardell JR.	1983
VAX DEBUG	interactive, symbolic, multilingual debugger		Beander B.	1983
MAP	static analysis	COBOL	Tischler R., Schaufler R. Payne C.	1983
Ctrace	preprocessor	C	Steffen JL.	1984
Lilith	high-level, GUI	Modula-2	Geissmann L.	1984
SHD	screen oriented	Pascal	Gars VK.	1984
ASDB	symbolic, source-level	C	Kodama K., Fukushima S., Horii K.	1985
SDE	symbolic, macro-oriented data abstractions	EC	Katzenelson J., Strominger A.	1985
VIPS	linked-list visualisation		Shimomura T., Isoda S., Ono Y.	1985
Chillscope	event-action breakpoint	CHILL	Hallsteinsen SO.	1986
Periscope	symbolic	C, Fortran, Pascal, BASIC	Christensen W.	1986
2X	static and dynamic analysis		Clemente G., Congiu S., Moro M.	1987
DDB	interactive, source-level	C	Livshin D.	1987
gdbxtool	graphical display and editing of data structures		Potrebic P., Goldman P.	1987
Ups	GUI		Bovey JD.	1987
	execution backtracking		Agrawal H., Spafford EH.	1988
AdaProbe	friendly user interface	Ada	Altarac H., Plisson P.	1988
DOC	optimised code debugger	C	Coutant DS., Meloy S., Ruscetta M.	1988
ILM	on-line debugger	COBOL, PL/1	Varga V.	1988
Pbug	abstractions	Pascal, PL/M	Benmerl T., Huber F., Stampfl R.	1988
DDS	declarative debugging		Takahashi N., Ono S.	1989
PISLD	interactive source-level	Pascal	Chi Zn., Liu C.	1989

Name	Technology	Lang. / Arch.	Authors	Date
dbx	interactive	C, Fortran	Linton MA.	1990
Moped	tracing, backtracing	lisp	Pourheidari M., Kessler RR., Carr H.	1990
O_2	DBMS	O_2	Meersman RA, Kent W, Khosla S.	1990
PDB	GUI, object-oriented, distributed		Maybee P.	1990
Thisdb	GUI, symbolic	C	Hagen T.	1990
	algorithmic, semi-automatic	Pascal	Fritzson P., Gyimothy T., Kamkar M., Shahmehri N.	1991
	visual, program generating technique		Ming Z.	1991
CDBX	X-Windows	Cray	Rigsbee PA.	1991
CXdb	optimised code debugger	Convex	Streepy LV Jr., Brooks G., Buyse R., Chiarelli M., Garzione M., Hansen G., Lingle D., Simmons S., Woods J.	1991
Dalek	events, control and query language, dataflow		Olsson RA., Crawford RH., Ho WW., Wee CE.	1991
DARTS	dynamic, real-time		Tinmerman M., Gielen FJA.	1991
DBL	interactive, functional language		Krishnamoorthy MS., Anastasiou AD.	1991
DUDU	functional models, automatic		Allemang D.	1991
MultiScope	multiple debuggers	DOS/Windows	Kearns S.	1991
SIPDES	expert system	Pascal	Doukidis GI., Paul RJ.	1991
Spyder	checkpoint, backtrack		Agrawal H. De Millo RA. Spafford EH.	1991
VBD-II	object-oriented interface	NEC	Hiramatsu T., Ichinose N., Kojo T.	1991
Watson	GUI environment for debugger development	Cray	Murrish R.	1991
GHC	process-oriented, reflection, program transformation		Maeda M.	1992
ldb	retargetable	C	Ramsey N., Hanson DR.	1992
Opium	programmable	Prolog	Ducasse M.	1992
UDI	universal interface	C	Mann D.	1992
	distributed execution replay	CHORUS	Ruget F.	1994
ACID	language interpreter		Winterbottom P.	1994
ADAPT	automated	Prolog	Gegg-Harrison TS.	1994
HotWire	visualization	C++, Smalltalk		1994
gdb	GNU debugger	C, C++, Fortran	Butt F.	1995
cdb	machine-independent	C	Hanson DR.	1996
DDD	graphical front-end	GDB, DBX	Zeller A., Lutkehaus D.	1996
FIND	automated debugging assistant		Shimomura T.	1996
	traversal based visualization		Korn JL., Appel AW.	1998
SNiFF+	customisable		Parker T.	1998
wshdbg	debugger for CGI	TCL	Vckovski A.	1998
	program instrumentation, load-time code generation, query optimization, and incremental reevaluation		Lencevicius R., Holzle U., Singh AK.	1999
Coca	breakpoint mechanism is	C	Ducasse M.	1999

Name	Technology	Lang. / Arch.	Authors	Date
	based on events related to language constructs			
RAID	probabilistic reasoning, heuristic debugging knowledge, structural analyses	C	Burnell L., Meadows A., Bass P., Biggers K., Priest J.	1999

Appendix G

CD-ROM CONTENTS

Folder Name	Description
About	About the author.
Case Studies	The three case studies examining the use of GUARD.
Case Study 1	Data parallel case study.
Case Study 2	Distributed memory case study.
Case Study 3	Shared memory case study.
guard-0.9.17	Source code of the GUARD-2000 debugger.
doc	Documentation associated with the debugger.
src	Debugger source tree.
aif	Architecture independent format library.
compat	Compatibility library.
dbgsrv	Debug server.
gc	Dataflow compiler.
guard	Debug client.
tools	Visualisation tools.
util	Utility library.
zgdb-4.16	Modifications to GDB version 4.16.
zgdb-4.17	Modifications to GDB version 4.17.
Thesis	Electronic version of debugger thesis.
doc	Debugger thesis in Microsoft Word 2000 format.
pdf	Debugger thesis in Adobe PDF format.

REFERENCES

[1] D. Abramson, M. Dix, and P. Whiting, "A Study of the Shallow Water Equations on Various Parallel Architectures", *14th Australian Computer Science Conference*, pp. 06-1 - 06-12, Sydney, 1991.

[1a] D. Abramson and G.K. Egan, "The RMIT Data Flow Computer: A Hybrid Architecture", *The Computer Journal*, June 1990.

[2] D. Abramson, I. Foster, J. Michalakes, and R. Sosic, "Relative Debugging and its Application to the Development of Large Numerical Models", *Proceedings of IEEE Supercomputing 1995*, San Diego, December 95.

[3] D. Abramson, I. Foster, J. Michalakes, and R. Sosic, "Relative Debugging: A New Methodology for Debugging Scientific Applications", *Communications of the ACM*, Vol. 39, No. 11, pp. 69 – 77, November 1996.

[4] D. Abramson and R. Sosic, "A Debugging Tool for Software Evolution", *CASE-95, 7th International Workshop on Computer-Aided Software Engineering*, pp. 206 – 214, Toronto, Canada, July 1995.

[5] D. Abramson and R. Sosic, "A Debugging and Testing Tool for Supporting Software Evolution", *Automated Software Engineering 3*, pp. 369 – 390, 1996.

[6] D. Abramson, R. Sosic, and G. Watson, "Implementation Techniques for a Parallel Relative Debugger", *Proceedings of PACT '96*, Boston, October 1996.

[7] Active Tools Inc., *The Clustor 1.5 User Manual*, San Francisco, CA, February 1999.

[8] E. Adams and S. Muchnick, "Dbxtool, A Window-Based Symbolic Debugger for Sun Workstations", *USENIX Association Summer Conference Proceedings 1985*, USENIX Assoc., pp. 213 – 227, El Cerrito, CA, USA, 1985.

[9] Arvind, L. Bic, and T. Ungerer, "Evolution of Data-flow Computers", Chapter 1, *Advanced Topics in Dataflow* Computing, Prentice Hall, 1991.

[9a] Arvind and R.S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture", *Lecture Notes in Computer Science 259*, pp. 1-29, 1987.

[10] P. Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior", *Proceedings ACM SIGPLAN and SIGOPTS Workshop on Parallel and Distributed Debugging*, WI, USA, May 5-6, 1988.

[11] P. Bates and J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", *Journal of System Software*, 3, pp. 255 – 264, 1983.

[12] T. Bemmerl and R. Wismüller, "On-line Distributed Debugging on Scaleable Multicomputer Architectures", *High Performance Computing and Networking, Volume II: Networking Tools*, Volume 797 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 394 – 400, April 1994.

[13] L. Burnell, et. al., "RAID: A System to Aid in the Removal of Program Bugs", *Proceedings of the Twelfth International Florida AI Research Society Conference*, AAAI Press, pp. 32 – 36, Menlo Park, CA, USA, 1999.

[14] R. Butler and E. Lusk, "Monitors, Messages, and Clusters: the p4 Parallel Programming System", *Parallel Computing*, 20 April 1994.

[15] D. Callahan and J. Subholk, "Static Analysis of Low-level Synchronization", *SIGPLAN Notices*, Vol. 24, No. 1, January 1989.

[16] D. Cheng and R. Hood, "A Portable Debugger for Parallel and Distributed Programs", *Proceedings of Supercomputing '94*, pp. 723 – 732, November 1994.

[16a] Y.M. Chong, "Data Flow Chip Optimizes Image Processing", *Computing Design*, pp. 97-103, October 1984.

[17] M. Cierniak and W. Li, "Unifying Dat . and Control Transformations for Distributed Shared-Memory Machines", *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995.

[18] W. Crowley, C. Hendrickson, and T. Rudy, *The SIMPLE Code*, Lawrence Livermore Laboratory, UCID-17715, February 1, 1978.

[19] J. Cuny, et. al., "The Ariadne Debugger: Scalable Application of Event-Based Abstraction", *ACM SIGPLAN Notices*, Vol. 28, No. 26, pp. 85 – 95, December 1993.

[20] Convex Computer Corporation, *Convex CXdb User's Guide*, Second Edition, October 1993, DSW-473.

[21] J. Dennis, "The Evolution of "Static" Data-flow Architectures", Chapter 2, *Advanced Topics in Dataflow Computing*, Prentice Hall, 1991.

[21a] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor, *Proc. 2nd ISCA*, pp. 126-132, January 1975.

[22] Dolphin Interconnect Solutions, Inc. (now Etnus, Inc.), *TotalView Multiprocess Debugger User's Guide*, Version 3.7.7, Revision 8, October 1997.

[23] M. Folk, L. Kalman and W. Whitehouse, *HDF User's Guide*, Version 4.1r1, National Centre for Supercomputing Applications, May 1997.

[24] G. Fox, et al., *Fortran D Language Specification*, Center for Research on Parallel Computation, Rice University, CRPC-TR90079, December 1990.

[25] J. Gait, "A Probe Effect in Concurrent Programs", *Software Practice and Experience*, Vol. 16, No. 3, pp. 225 – 233, 1986.

[26] J. Goguen, J. Thatcher, and E. Wagner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types", *Current Trends In Programming Methodology, Vol. IV: Data Structuring*, Yeh, R., (ed.), Prentice-Hall, pp. 80-149, Englewood Cliffs, New Jersey, 1978.

[27] Hewlett Packard Company, *HP/DDE Debugger User's Guide*, First Edition, B3476-90015, July 1996.

[27a] W. Hibbard and D. Santek, "The VIS-5D System for Easy Interactive Visualization", *Proceedings of IEEE Visualization '90*, pp. 129-134, 1990.

[28] High Performance Debugging Forum, "HPD Version 1 Standard: Command Interface for Parallel Debuggers", ed. C. Pancake and J. Francioni, *Technical Report CSTR-97*, Dept. of Computer Science, Oregon State University, 1997.

[29] High Performance Fortran Forum, *High Performance Fortran Language Specification*, Version 2.0, Center for Research on Parallel Computation, Rice University, CRPC-TR92225, January 1997.

[30] IBM Corporation, *IBM AIX Parallel Parallel Environment: Programming Primer*, Release 2.0, SH26-7223, June 1994.

[31] Institute of Electrical and Electronic Engineers, "Binary Floating-Point Arithmetic", *IEEE Std.754-1985*, Piscataway, N.J., 1985.

[32] Institute of Electrical and Electronic Engineers, "IEEE Standard Glossary of Software Engineering Terminology", *IEEE Std 610.12-1990*, New York, USA, 1990.

[33] P. Kacsuck, J. Cunha, G. Dózsa, and J. Lourenço, "A Graphical Development and Debugging Environment for Parallel Programs", *Parallel Computing*, Vol. 22, No. 13, pp. 1747-1770, 1997.

[34] K. Kennedy and K. McKinley, "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution", *Proceedings of the Sixth Workshop on Languages and Compilers for* Parallel *Computing*, Portland, Oregon, August 1993.

[35] B. Lazzerini and L. Lopriore, "Abstraction Mechanisms for Event Control in Program Debugging", *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, pp. 890-901, USA, July 1989.

[36] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs With Instant Replay", *IEEE Transactions on Computers C-36*, Vol. 4, pp. 471 – 482, April 1987.

[37] C. H. LeDoux and D. S. Parker, "Saving Traces For Ada Debugging", In *Ada In Use*, Proceedings of the Ada International Conference, ACM, Cambridge University Press, pp. 97 – 108, 1985.

[38] M. M. Lehman, "Programs, Programming and the Software Life Cycle", *Proceedings IEEE Special Issue on Software Engineering*, pp. 1060 – 1076, September 1980.

[39] M. M. Lehman, "The Programming Process" in *Program Evolution: Processes of Software Change*, M. M. Lehman and L. A. Belady eds., Academic Press Inc., USA, 1985.

[40] C. Lin and R. J. LeBlanc, "Event-based Debugging of Object/Action Programs", *Proceedings ACM SIGPLAN and SIGOPTS Workshop on Parallel and Distributed Debugging*, WI, USA, May 5-6, 1988.

[41] C. Lin and L. Snyder, "A Portable Implementation of SIMPLE", *International Journal of Parallel Programming*, Vol. 20, No. 5, 1991.

[42] C. Lin and L. Snyder, "ZPL: An Array Sublanguage", *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds, pp. 96 – 114, 1993.

[43] J. Lipson, *Elements of Algebra and Algebraic Computing*, Addison-Wesley, Reading, Massachusetts, 1981.

[44] R. Loeser and E. M. Gaposchkin, "The Second Law of Debugging", *Software-Practice & Experience*, Vol.6, No.4, pp. 577-578, UK, October-December 1976.

[45] J. Lumpp, T. Casavant, H. Siegel, and D. Marinescu, "Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems", *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, pp. 476 – 483, 1990.

[46] B. Magnusson and S. Minör, "III -- an Integrated Interactive Incremental Programming Environment Based on Compilation", *Proceedings of the ACM SIGSMALL Symposium on Small Systems*, 1985.

[47] R. F. Mathis, "Pre-execution, Batch, Iinteractive, and Postmortem Debugging", *Computer Science Conference '75*, ACM, pp. 9, New York, NY, USA, 1975.

[48] J. May and F. Berman, "Panorama: A Portable Extensible Parallel Debugger", *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 96-106, San Diego, May 1993.

[49] C. E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs", *ACM Computing Surveys*, Vol. 21, No. 4, pp. 593 – 622, December 1989.

[50] K. McKinley, S. Carr, and C.-W. Tseng, "Improving Data Locality with Loop Transformations", *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 4, pp. 424–453, July 1996.

[51] M. A. F. Müllerburg, "The Role of Debugging Within Software Engineering Environments", *ACM SIGPLAN Notices*, Vol. 18, No. 8, pp. 81 – 90, USA, August 1983.

[52] G. J. Myers, *Software Reliability: Principles and Practices*, John Wiley & Sons, USA, 1976.

[53] The National Center for Supercomputing Applications, *NCSA HDF Specification and Developer's Guide*, University of Illinois at Urbana-Champaign, November 1993.

[54] P. G. Neumann, *Computer Related Risks*, ACM Press, Addison-Wesley, New York, USA, 1995.

[55] N. Ramsey and D. R. Hanson, "A Retargetable Debugger", *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, pp. 22 – 31, ACM, 1992.

[56] R. Olsson, R Crawford, and W. Ho, "A Dataflow Approach to Event-Based Debugging", *Software-Practice and Experience*, Vol. 21, No. 2, pp. 209 – 229, February 1991.

[57] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors", *Proceedings of the International Conference on Parallel Processing*, St. Charles IL, pp. II39 – 48, August 1989.

[58] G. Rivera and C.-W. Tseng,, "Locality Optimization for Multi-Level Caches", *Proceedings of the ACM/IEEE SC99 Conference*, Portland, Oregon, November 1999.

[59] J. Rose and G. Steele Jr., "C*: An Extended C Language for Data Parallel Programming", *Technical Report PL 87-5*, Thinking Machines Corporation, Cambridge, MA, 1987.

[60] A. Rosenberg, "Storage Mappings for Extendible Arrays", *Current Trends In Programming Methodology*, Vol. IV: *Data Structuring*, Yeh, R., (ed.), pp. 263-311, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[61] M. Rosing and S. Yabusaki, "A Programmable Preprocessor for Parallelizing Fortran-90", *Proceedings of the ACM/IEEE SC99 Conference*, Portland, Oregon, November 1999.

[62] R. Sadourny, "The Dynamics of Finite-Difference Models of the Shallow Water Equations", *Journal of Atmospheric Science*, Vol. 32, pp. 680 – 689, 1975.

[63] R. Sosič, "Design and Implementation of Dynascope, a Directing Platform for Compiled Programs", *Computing Systems*, Vol. 8, No. 2, pp. 107 – 135, 1995.

[64] R. Sosic and D. Abramson, "Guard: A Relative Debugger", *Software – Practice and Experience*, Vol. 27, No. 2, pp. 185 – 206, February 1997.

[65] R. Srinivasan, "XDR: External Data Representation Standard", *RFC 1832*, Sun Microsystems, Inc., August 1995.

[66] R. Stallman, *Debugging with GDB – The GNU Source Level Debugger*, Edition 4.12, Free Software Foundation, January 1994.

[67] Sun Microsystems Inc., "RPC: Remote Procedure Call Protocol Specification Version 2", *RFC-1057*, June 1988.

[68] R. Tischler, R. Schaufler, and C. Payne, "Static Analysis of Programs as an Aid To Debugging", *ACM SIGPLAN Notices*, Vol. 18, No. 8, pp. 155 – 158, USA, August 1983.

[69] G. Watson and D. Abramson, "Relative Debugging For Data-Parallel Programs: A ZPL Case Study", *IEEE Concurrency*, Vol. 8, No. 4, pp. 42 – 52, USA, October 2000.

[69a] G. Watson and D. Abramson, "The Architecture of a Parallel Relative Debugger", *13th International Conference on Parallel and Distributed Computer Systems – PDCS 2000*, Las Vegas, Nevada, August 2000.

[70] J. C. Weber, "Interactive Debugging of Concurrent Programs", *SIGPLAN Notices* Vol. 18, No. 8, pp. 112 – 113, 1983.

[71] R. Wismuller, M. Oberhuber, and J. Krammer, "Interactive Debugging and Performance Analysis of Massively Parallel Applications", *Parallel Computing*, Vol. 22, No. 3, pp. 415 – 442, March 1996.

[72] M. Wolf and M. Lam, "A Data Locality Optimizing Algorithm", Proceedings of the *SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

GLOSSARY

abstract algebra

The mathematics of generalized abstract arithmetical operations.

application programming interface (API)

A software interface that enables applications to communicate with each other.

assertion

A condition specified *a priori* that must be satisfied for the correct execution of a program.

asynchronous

Processes or actions whose execution can proceed independently.

big-endian

A computer architecture in which, within a given multi-byte numeric representation, the most significant byte has the lowest address.

bijective

A bijective function maps each element of a set A onto one and only one element in a set B and maps each element of B onto one and only one element in A.

block-cyclic distribution

A technique for distributing two dimensional data arrays to different processors in a parallel application that gives a one or more columns or rows of data values to each processor.

breakpoint

A point in a program that, when reached, triggers some special behaviour useful to the process of debugging.

cache

A high-speed memory, local to a single processor, whose data transfers are carried out automatically in hardware.

callback

The mechanism by which a server program can invoke a service in a client program.

cartesian product

A set of all pairs of elements (x, y) that can be constructed from given sets, X and Y, such that x belongs to X and y to Y.

client/server architecture

An arrangement whereby a computer program (the client) sends requests for services to another computer program (the server) across a communications network.

command line interpreter (CLI)

A program which reads textual commands from the user or from a file and executes them.

data decomposition

A technique where the data on which a sequential computation operates is partitioned into smaller pieces in a way that is suitable for parallel computation.

dataflow

A model of parallel computing in which programs are represented as dependence graphs and each operation is automatically blocked until the values on which it depends are available.

data parallelism

A model of parallel computing in which a single operation can be applied to all elements of a data structure simultaneously.

debug

To detect, locate, and correct faults in a computer program.

distributed memory

Memory that is physically distributed amongst several modules.

graph

A collection of nodes and edges symbolising a system of interrelations.

graphical user interface (GUI)

A user interface based on graphics (icons, pictures and menus) instead of text.

heterogeneous

Containing components of more than one kind.

isomorphic

Two mathematical objects that have the same structure, i.e. for every component of one there is a corresponding component of the other.

iso-surface

An implicit surface that exists wherever a continuous scalar field in a volume is at a particular value.

little-endian

A computer architecture in which, within a given multi-byte numeric representation, bytes at lower addresses have lower significance.

message passing

A style of interprocess communication in which processes send discrete messages to one another.

nondeterminisim

A property of a computation that may have more than one result.

parallel computer

A computer system made up of many identifiable processing units working together in parallel.

parallelisation

The process of turning a serial computation into a parallel one.

pixel map

A two-dimensional arrangement of picture elements (pixels).

probe effect

The interaction between the debugger and the program being debugged, generally timing related, that affects the appearance of a program error.

process

The fundamental entity of the software implementation on a computer system.

processor

A hardware device that executes the commands in a stored program in a computer system.

relative debugging

The process of locating and identifying errors by comparing a suspect program against a reference code.

sequential computer

A computer comprising a single central processing unit (CPU) that executes a program to perform a sequence of read and write operations on an attached memory. Also known as a Von Neumann architecture.

shared memory

Memory that appears to the user to be contained in a single address space and that can be accessed by any process.

software development

A problem-solving process that involves the translation of a complex problem into detailed instructions that direct a computer to solve the problem.

software evolution

Continuous growth through the initial development and ongoing maintenance of software.

software life cycle

A two-phase process consisting of a design phase and a testing phase.

vector computer

A computer designed to apply arithmetic operations to long vectors or arrays.