 *Relative debugging is a powerful paradigm that lets us locate errors in programs that result from porting or rewriting code. The authors describe their experience using relative debugging to compare a program written in a sequential language with one that was ported to the data-parallel language ZPL.*

Relative Debugging for Data-Parallel Programs: A ZPL Case Study

In this article, we describe Guard99, a new implementation of relative debugging for parallel platforms that lets a programmer locate errors by observing the divergence in key data structures as two programs are simultaneously executed. The implementation uses a client-server, machine-independent architecture, so it can debug

programs that execute on different platforms. By employing an architecture-independent implementation layer, the debugger is able to hide variations in data representation on these different platforms from the user.

The case study we report represents the first time a relative debugger has been used to locate errors by comparing one program written in a data-parallel language to another written in a sequential language (for more information on relative debugging, see the related sidebar). The data-parallel language ZPL,¹ developed at the University of Washington, incorporates special linguistic constructs that let a programmer specify parallel data structures and how to perform the computation in parallel. ZPL is a high-level machine-independent parallel programming language that has been ported to a wide variety of platforms.² Our decision to use it in the case study was arbitrary but logical, because we had access to the language designers and an actual implementation. The techniques described here apply equally well to other data-parallel languages, and we designed the debugger implementation so that adding support for additional languages is a straightforward process.

Guard99 found errors previously not located in a sample ZPL program—the programmers assumed the program was operating correctly even though there were minor numeric differences. Guard99 showed that some of these differences could be attributed to variations in the execution of sequential and parallel versions of ZPL. The work is particularly important because a programmer not fluent in ZPL or the underlying application was able to debug the ZPL code. The errors were located quickly and efficiently.

ZPL

ZPL is a data parallel array-based language³ that supports the array as a fundamental data type. It also provides various features that let programmers generalize algorithms using array semantics. Although it is primarily designed as a data-parallel language, ZPL executes on both sequential and parallel architectures. It 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

Relative debugging

Relative debugging is a high-level technique that lets us compare data in a *reference* program to that of a *suspect* program.^{1,2} Often, programs derived from a reference program, perhaps through porting or some other form of an evolutionary software process, suffer from introduced errors. Relative debugging provides a powerful technique for locating porting errors quickly. Various case studies reporting the results of using relative debugging have been published.^{1,3-4} The concept of relative debugging is both language and machine independent. It lets users compare data structures without concern for the implementation, so they can focus on the cause of the errors rather than on the implementation details.

Several versions of relative debugging have been built. The first one, Guard95, only supported programs running on sequential platforms.² Guard95 included some limited machine independence, but this was mostly restricted to supporting machines with different byte ordering. The current implementation, Guard99, provides support for parallel computer systems using an enhanced dataflow mechanism,⁵ and it also includes a machine-independent layer. It lets us compare data when the compiled structures differ and when the machine architectures use different byte orderings or word sizes. For example, it can compare real numbers in different formats and sizes, as well as characters that are represented by different collating sequences.

In our case study, arrays are implemented quite differently in C and ZPL, as a ZPL array can be physically distributed across a number of processes, relying on complex data structures to manage the parallel implementation issues. Guard99 lets the user maintain this viewpoint by ensuring that these differences are hidden. The ZPL runtime library and Guard99 transparently manipulate parallel data structures to allow the comparisons to be performed (we discuss the debugger architecture that supports these features in the main text).

To the user, a relative debugger seems like any traditional debugger except that it also provides additional commands for comparing data from different processes. The debugger can control more than one process simultaneously, so when the processes halt at breakpoints, data comparison can be performed using the `compare` statement.

This technique is known as an imperative comparison and works well for simple cases. An alternative declarative comparison technique is also provided to allow the user to define a set of criteria for the correct operation of the suspect program a priori. These criteria are defined using a series of assertion commands.

When performing comparisons, errors might be incorrectly attributed to differences in the precision of the program variables or other minor numeric factors. To avoid this, Guard99 lets the user specify a tolerance value. Variables are considered equivalent when the result of a comparison is below this value. Two different types of tolerance are supported: absolute and relative. For absolute tolerance, the magnitude of the difference between the variables is compared directly with the tolerance value. So, for some tolerance ϵ , the error is ignored if $|v_1 - v_2| < \epsilon$. In situations where the values are very small but the differences still constitute a significant error, relative

tolerance can be used. Here, the difference is first divided by the maximum of the two variables before being compared to the tolerance. In this case, any errors will be ignored if

$$\frac{|v_1 - v_2|}{\max(|v_1|, |v_2|)} < \epsilon .$$

Guard99 provides three methods for visualizing comparison results: a simple text format, hierarchical data format (HDF), and architecture-independent format (AIF).⁵ The text format is generally used only for quickly visualizing simple data structures, as it becomes much too unwieldy for large amounts of data. HDF and AIF are intermediate formats that can be used as input into a visualization package to generate 2D and 3D representations of the data. HDF is limited to multidimensional scientific data sets but is currently the de facto standard data format. AIF allows arbitrarily complex data structures to be represented, and we discuss the format in more detail in the ZPL section of the main text because it underpins the debugger's machine and language independence.

Some evidence suggests that visualizing comparisons, particularly using 2D and 3D representations, provides the user with a means of characterizing patterns of differences. In a previous study, a time-series isosurface representation of the error was used to identify independent errors in a mesoscale weather model. In particular, the isosurface's structure lets us identify errors in different code sections that were responsible for various physical processes. For example, an error in the physics on the planetary boundary layer was identified through an isosurface that was visible at the bottom of the 3D space. Another, different error in the long-wave radiation physics code was visible in the top of the atmosphere. This and other studies are discussed in more detail elsewhere.⁴ Some differences in our case study show characteristic periodic behavior, suggesting problems involving trigonometric operations. In all these cases, we can make generalizations 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.

References

1. D. Abramson and R. Sasic, "A Debugging and Testing Tool for Supporting Software Evolution," *Automated Software Eng.*, Vol. 3, Nos. 3-4, Aug. 1996, pp. 369-390.
2. R. Sasic and D. Abramson, "Guard: A Relative Debugger," *Software—Practice and Experience*, Vol. 27, No. 2, Feb. 1997, pp. 185-206.
3. D. Abramson and R. Sasic, "A Debugging Tool for Software Evolution," *CASE-95, Seventh Int'l Workshop Computer-Aided Software Eng., Proc. 7th Int'l Workshop Computer-Aided Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1995, pp. 206-214.
4. D. Abramson et al., "Relative Debugging: A New Methodology for Debugging Scientific Applications," *Comm. ACM*, Vol. 39, No. 11, Nov. 1996, pp. 69-77.
5. D. Abramson, R. Sasic, and G. Watson, "Implementation Techniques for a Parallel Relative Debugger," *Proc. PACT '96*, IEEE Computer Soc. Press, Los Alamitos, Calif, 1996.

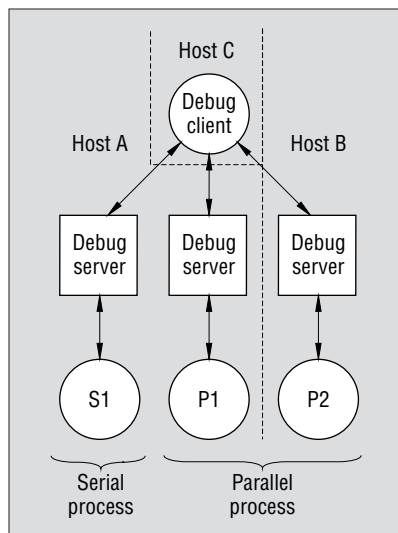


Figure 1. The client-server architecture Guard99 uses.

semantics of the array operations, so there are no parallel directives or mechanisms for explicit message passing. Programmers see a single address space and can use traditional sequential programming semantics for software development.

ZPL provides most attributes of traditional sequential programming languages. In addition, it supplies a number of new features to support the array semantics. *Regions* specify either a set of indices that define an array's bounds or a section of the array over which computations are performed. For example, the following declarations define a 10×10 region and an array containing 100 elements:

```
region R = [1..10, 1..10];
var A: [R] double;
```

Prefixing a statement with a region specifies that all operations on arrays of the region's same rank are to be performed for the indices defined by the region. The statement

```
[R] A := B + C
```

results in the sum of elements of *B* and *C* that are within the region *R* being stored in the corresponding elements of *A*. Any elements of *A* that are outside the region are unchanged by the statement.

Vector constants are denoted in ZPL using *directions*, which let a programmer specify relative positions that can perform transformations on regions. A typical declaration might be

```
direction north = [-1, 0];
           east  = [0, 1];
```

There are a number of special operators

for transforming or referencing regions. The `of` operator takes a direction and region as its operands and defines a new boundary region that is adjacent to the base region. The `@` operator, when applied to an array, translates the array's region by a specified direction, then references those elements of the array. Thus, to refer to elements of *B* obtained by adding the direction north to each index tuple in *R*, we'd use the following statement:

```
[R] A := B@north;
```

Periodic and mirrored boundary conditions are supported using the `wrap` and `reflect` operators. The `wrap` operator causes values on an array's boundary to be copied from the "opposite" side of the array, while the `reflect` operator results in values on the boundary of an array being copied from elements adjacent to the boundary. `wrap` and `reflect` are typically used as follows:

```
[north of R] wrap A;
[east of R] reflect B;
```

Traditional reduce and scan semantics that apply a function accumulatively over an array are also supported. The `reduce` operator produces a scalar value representing the accumulated result of the function applied to each element in the array. The result of a `scan` operation is an array in which the *i*th element is the accumulation of the function applied to the first *i* elements of the array, in row-major order. In the following statements

```
Sum := ++<A; Plus := +||A;
```

the sum of all the elements of *A* is stored in the scalar `Sum`, and each element in the array `Plus` contains the sum of the preceding elements in each row.

ZPL achieves a high degree of machine independence by compiling to SPMD code for an abstract parallel machine in which the processes are arranged in a 2D mesh. The ZPL compiler developed at the University of Washington generates ANSI C code that is postcompiled using a native C compiler on the target system. This C code interfaces to libraries that emulate the abstract parallel machine on the target architecture and provide a well-

defined interface to architecture-specific features. In the process, the logical process mesh is mapped onto the target processor topology.

Architecture details

A relative debugger's architecture has more functional requirements than traditional sequential and parallel debuggers. In addition to the need to control two or more processes simultaneously, the processes being debugged might also run on physically distributed systems, possibly employing different machine architectures. Managing assertions requires simultaneously controlling multiple processes and extracting data at arbitrary times in the execution life of the process. Support for data-parallel languages requires the debugger to interpret the parallel data structures that the language runtime system uses.

Guard99 addresses these requirements through a variety of different mechanisms. A client-server architecture controls multiple processes on distributed systems and architectural differences are handled by providing a machine-independent data representation that is used for all data manipulation activities. A dataflow compiler translates user-specified assertions into a graph that a dataflow engine can later execute. Using a dataflow mechanism overcomes the control and synchronization problems that are inherent in the relative debugging technique. Finally, a debug back end incorporates data-parallel language support.

CLIENT-SERVER ARCHITECTURE

Programs controlled by a relative debugger can be any combination of sequential and parallel codes. Sequential programs might be specific to a small range of machines, such as particular vector architectures. Many massively parallel processing systems use networks of independent nodes and rely on the implementations of parallel programming models—such as a Message Passing Interface and Parallel Virtual Machine—to manage process creation.

A number of debuggers and debugging environments have been developed to support parallel and distributed archi-

tectures, many of which employ a client-server mechanism.⁴⁻⁷ Some, such as Detop,⁸ support both task and data parallel codes. Guard99 also employs a client-server model to ensure that the processes being debugged can be distributed onto multiple platforms and controlled independently. Figure 1 shows the client-server architecture Guard99 uses.

Figure 2 shows details of the components that make up the debug client. The client user interface 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 graph. The dataflow engine can then execute this graph at a later date.

The debug client Application Programming Interface layer manages the debug requests from the client, regardless of whether they are a result of immediately executed commands or are generated by the dataflow engine. It also provides a consistent interface to debugging actions that can be performed on target programs. 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 architecture-independent format API provides an interface for managing and manipulating data in an AIF. All data transmitted between client and server is first converted to this format using routines that the AIF API supports.

The debug servers are responsible for managing the processes that are being debugged. Each server manages exactly one process—so, for example, if a parallel program starts 10 processes, then 10 debug servers also need to be created. Figure 2 also shows the components that constitute a debug server. The server receives requests from the client through 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 back end. The back end con-

trols the debugger's low-level architecture-specific functions. Maintaining this distinction between the server API and the back end allows for a clean separation between the architecture-independent and architecture-dependent parts of the server. This ensures that additional architecture support can be easily added to the debugger.

ARCHITECTURE-INDEPENDENT DATA FORMAT

In addition to debugging programs on physically separate hosts, a relative debugger must also support programs running on heterogeneous architectures. Data from remote systems must be transferred to the client as a result of formatting or display commands, or for arithmetic or comparison operations generated by the dataflow engine when executing assertions. The remote systems (and the client) might each employ different architectural features such as word length and byte ordering.

In the client-server model we present, the client requests data from a remote process by sending a request to the debug server controlling that process. The server first converts the data into an architecture-independent (data) format on the remote system and the formatted data is transferred to the client. The client then uses AIF library routines for all manipulation, arithmetic, and comparison operations performed on this data.

Standard networking protocols such as XDR have addressed the problem of sending architecture-specific data over a network. Much work has also been done on the development of architecture-independent file formats, with the National Center for Supercomputing Applications' hierarchical data format now accepted as the de facto standard. However, none of these approaches address the issues of performing in-memory operations on data from architecturally different systems. As a result, we have provided a well-defined API and

have developed library routines that implement all the necessary arithmetic and conversion operations.

AIF achieves architecture independence through format standardization and by employing data tags. Byte size is standardized to eight bits, and integer byte ordering is big-endian (for a given multibyte numeric representation, the most significant byte has the lowest address). Characters are single byte and are expressed in terms of the ASCII collating sequence, and floating-point numbers use the big-endian IEEE 754-1985 format. Simple data types using other architecture-specific data representations are translated into the standard formats when converted to AIF.

When data is converted to the standard format, it is tagged with a format descriptor string that describes the data's size and layout. Table 1 shows the tags that are currently available. Tags for simple data types provide AIF library routines with information such as the size of the data and whether or not the data is signed. This supports different integer sizes and single, double, and extended floating-point formats and ensures that the AIF library routines can perform calculations with no loss of precision. Complex data types have tags that describe the data's size and memory layout and contain nested tag types. Figure 3 shows an example of how a C structure is converted to AIF.

The current implementation provides descriptors for C, Fortran, and ZPL. Support for another language can be added by defining new descriptors for

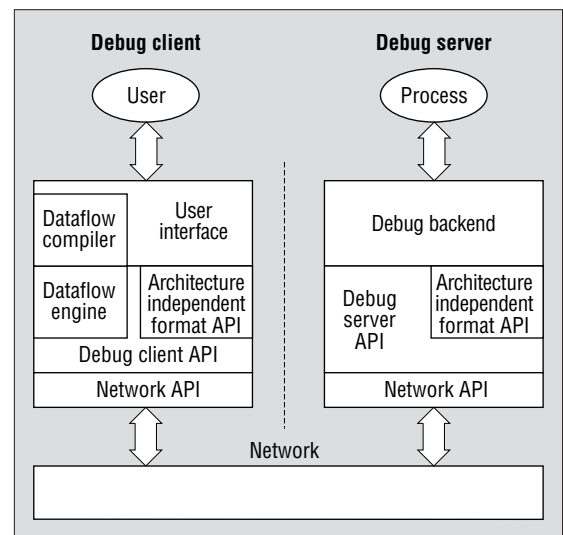


Figure 2. The debugger architecture.

Table 1. AIF format descriptor tags.

TAG	TYPE	DETAILS
cs	Character	<i>s</i> is s (signed) or u (unsigned)
is/	Integer	<i>s</i> is s (signed) or u (unsigned), <i>l</i> is size in bytes
f/	Floating point	<i>l</i> is size in bytes
^t	Address	Type <i>t</i> , <i>l</i> is size of the address in bytes
[t1]t2	Array	Type <i>t2</i> with index type <i>t1</i> , which must be range
{l = f₁@o₁#l₁: t₁, ..., f_n@o_n#l_n: t_n}	Structure or union	<i>l</i> is size in bytes, <i>f_i</i> is the name of the field, <i>o_i</i> is the offset in bits from structure's origin, <i>l_i</i> is the size in bits, and <i>t_i</i> is type of the field
<e₁=v₁, ..., e_n=v_n>	Enumeration	<i>e_i</i> is the name of each value <i>v_i</i>
xv_{min}..v_{max}t	Range	Based on integral type <i>t</i> with <i>v_{min}</i> and <i>v_{max}</i> as limits
v/	Void	<i>l</i> is size in bytes
Rrt	ZPL region	Rank <i>r</i> , whose limits are based on integral type <i>t</i>

each of the languages data types that the current descriptors don't cover and by providing routines to perform appropriate operations on these data types. For example, including support for Data Parallel C would require adding a shape descriptor and routines that implement shape semantics to satisfy the AIF API's requirements.

DATAFLOW ENGINE

In general, the processes being controlled are not synchronized. Thus, they might reach breakpoints and send data to the debugger at arbitrary times in relation to each other. Because of this, relative debugging uses a dataflow execution mechanism for processing user-defined assertions. We previously examined the reasons for choosing such a mechanism over more conventional execution mechanisms.⁹ Here, we only summarize the concept, which provides an operational semantics for relative debugging constructs in terms of dataflow graphs. In the debugger, the deferred mode command

interpreter collects assertions and other control statements and passes them to the dataflow compiler. The compiler translates the assertions into a dataflow graph, which is stored internally. Figure 4 shows a simplified dataflow graph that results from compiling an example assertion. When the user issues an immediate mode command to start the graph, it is passed to the dataflow engine for execution.

Nodes in the dataflow graph *fire* when tokens are present on all their inputs. The result of a node firing is to generate another token. The graph edges specify connections between nodes, which determine where to send the resulting token.

In this example, executing the graph begins by setting breakpoints at line 4,300 in \$proc1 and 4,400 in \$proc2 and by sending the appropriate tokens to a SETBP node. The graph starts executing the programs using the CONT nodes and then waits until the breakpoints are reached using the WAIT nodes. Once a program is stopped at the appropriate breakpoint, data is extracted

using the READ node and is sent to the COMP node for comparison. At the same time, sending a token back to the CONT node restarts program execution. The programs are free to resume execution as soon as they restart; however, if they contain a loop structure, they might hit the breakpoints again. Because the dataflow engine will continue to check for programs stopped at a breakpoint, the end result will be a steady stream of tokens reaching the COMP node's inputs. As soon as data becomes available on both inputs of the COMP node, a comparison is performed. If differences exist, the results are sent to the DISP node to be displayed to the user.

This approach's main advantage is that it implements the semantics of assertion statements naturally. Two data structures can only be compared when the data is available from both processes, which in turn depends on when arbitrary breakpoints have been reached. Because a compare node will only fire when tokens are present on both its inputs, the dataflow engine enforces the matching rules automatically.

DATA-PARALLEL LANGUAGE SUPPORT

ZPL, like other data-parallel languages, relies on the language runtime 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 and Fortran D, provide mechanisms to specify this). Similarly, the user should interact with the debugger in terms of the data structures themselves, without concern for their decomposition and distribution.

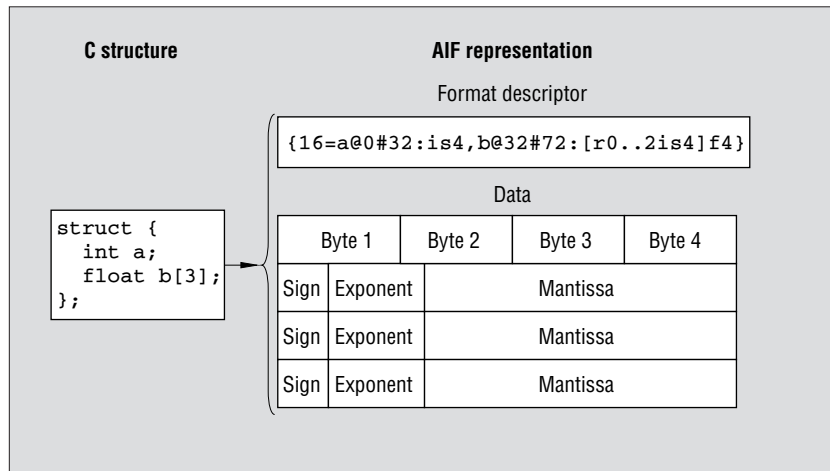


Figure 3. Architecture-independent format conversion example.

Data-parallel language runtime systems generally maintain a per-process data structure that contains a description of each parallel array and provides information such as the array's rank, each dimension's size, and information on the array's distribution across the processes. 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 resides in the process. In ZPL, this distribution information is stored in a runtime 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;
};
```

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, \dots, a_n) , where n is the number of dimensions specified by *numdims*, the location of the data is given by

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

The debug server interprets this information and uses it to obtain a copy of the data that is specific to the particular process.

To let data-parallel arrays be used in assertions in a transparent manner, Guard99 uses the decomposition infor-

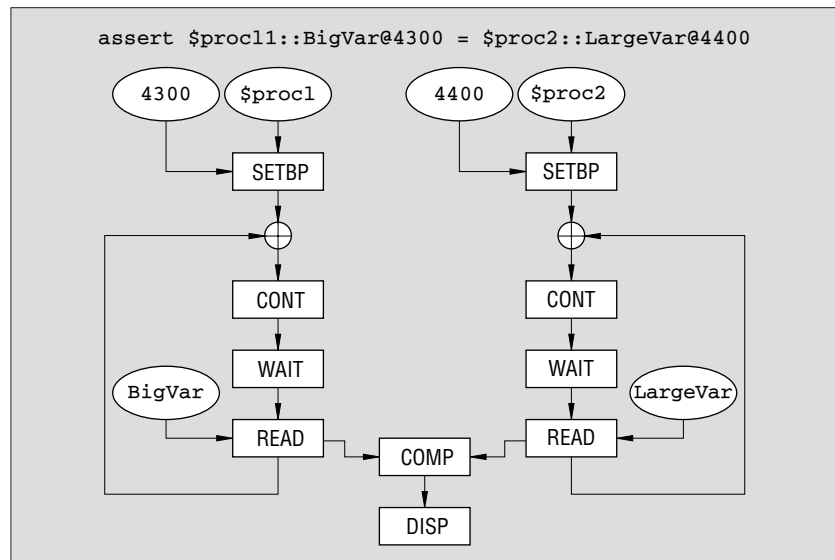


Figure 4. An assertion compiled to a dataflow graph.

mation along with the per-process location information to reassemble a complete structure. Because Guard99 knows the language type of the program being debugged, it can apply the language decomposition rules to request the components of the array from each process via the debug servers. It then reassembles these components into a complete array, and assertions can be used to compare the data in this array with that obtained from a reference program.

Adding data-parallel language support to the debugger then becomes a three-step process. First, the syntax and semantics of the language must be defined in the debugger parser. However, because 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 back end. Finally, AIF tags might need to be added to support any new data types the language introduces.

For example, to add support for the data-parallel language C* to Guard99, new language support would need to be included in the parser.¹⁰ 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 runtime representation of parallel variables would need to be added to allow the debugger back end to access the parallel array information. In C*, as in ZPL, a single

structure stores 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.

DEBUG BACK END

The development of Guard99 has been primarily concerned with the technique of relative debugging. We avoided the overheads of developing a machine-specific debug engine by providing a well-defined interface between the debug server and the low-level debug functions. This lets us use a pre-existing debugger, such as GDB, for this purpose.¹¹

In addition to providing the low-level debugging functions, using a back end debugger also lets us isolate language-specific details in one place. Accordingly, the debug client and server can be designed to be language independent, relying on the back end to interpret the syntax and semantics of individual languages. Supporting a new language, in this case ZPL, then becomes a process of modifying GDB to interpret the language's specific syntactic and semantic details. This process also has the resulting benefit of enhancing GDB as a standalone debugger.

Simple: A case study

Our case study illustrates the power of relative debugging when applied to two widely different programming languages. It examines the problem of debugging a sequential C code that has been ported to the data-parallel language ZPL. We

Table 2. Scalar error value.

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

```

                                simple.z

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 5. ZPL code structure.

demonstrate that the architecture proposed earlier is sufficiently powerful to handle the differences in the two languages and their underlying platforms.

The case study also highlights the efficiency of the debugging technique for locating errors across multiple program versions.

The program is the Simple hydrodynamics code.¹² To demonstrate our debugging methodology, we chose a problem size of 128×128 elements using four iterations. 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, we chose to run the ZPL code 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. Table 2 shows the values that were produced for each of the four iterations.

Both the C and ZPL codes employ double-precision variables for all computations, so we would expect the scalar error value to be equivalent to within the precision available, or about 15 decimal places. The precision of the floating-point representation also lets us 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 worked correctly, even though first examination showed that the codes produce slightly different results. To account for the discrepancy, we assumed that different numeric evaluation techniques in the language runtime systems or minor numerical errors were the likely cause. We adopted the three-phase approach when debugging the codes, because it was not obvious at the outset which of these factors contributed to the differences.

The first step compares a single process ZPL code with the same code on multiple processes 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 abil-

ity to compare a suspect program with a reference code. As a result, a debugging methodology using an iterative refinement process can help narrow 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.^{13–15}

CODE DESCRIPTION

The Simple code models the hydrodynamics of a pressurized fluid inside a spherical shell. The simulation computes values that describe the fluid's physics 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 modeling the inside of a spherical shell, the problem's symmetry reduces the computation 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 2D array.

Figure 5 shows the structure of the ZPL version of the Simple code. 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 computes the `En_error` array, and the second 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. The ZPL runtime system can then automatically partition this region into blocks that the individual processes can manage. The algorithm has also been designed so that computations in each phase share the same data dependencies, thus minimizing the overall communications overhead. More details on the ZPL implementation of Simple appear elsewhere.¹

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 (see Figure 6) is contained in the module `simple.c` and consists of an

initialization 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.

For the C code, calculation of the scalar error value is performed in `energy()` (see Figure 7). 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 AND PARALLEL ZPL COMPARISON

The first step in the debugging process was to compare the ZPL code in a single-process configuration with that in a multiprocess 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} . Because differences of this magnitude are still significant for double-precision floating-point numbers, these appeared to be errors introduced by the parallel runtime system.

Guard99 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

```

simple.c

double x[DL+2][DK+2];
double heat[DL][DK];
double en_error[DL][DK];
double theta[DL][DK];
double delta_t[DL][DK];

main()
{
    int loop = 0;

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

```

Figure 6. C code structure.

```

energy.c

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. C scalar error calculation.

factor affected by topology changes, it is likely that the nonassociative nature of these operations caused the variations. This result is important, because it shows

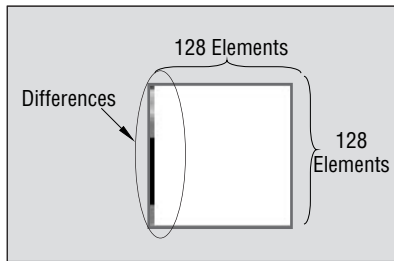


Figure 8. Differences between `$c::heat` and `$zpl::Heat`.

that nondeterministic behavior can affect the computation's results and effectively set a lower bound to the final error value's accuracy.

ZPL AND C COMPARISON

Having isolated the variations the parallel nondeterminism introduced, we could 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 the ZPL and C Simple codes starts by defining assertions for the four phase variables used in the scalar error calculation. Because the magnitude

of the error was around 10^{-4} , we also set the initial error tolerance to $10^{-5} \leq \epsilon \leq 10^{-1}$:

```
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 store the results of the heat phase computation. Figure 8 shows a visualization of these differences. Because the variables are 2D arrays, it is convenient to visualize the dif-

ferences as a 2D bitmap. These bitmaps are generated from difference information by assigning a color to represent the difference's magnitude, ranging from blue for the smallest difference, through green, yellow, and red, to black for the largest. White indicates no difference.

The differences appear 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 9.

Careful examination of both codes indicates that the term `delta_t` (shown in bold in the ZPL code) was erroneously included in the computation of `Heat` in the ZPL code. Prior to using `Guard99`, this error was not detected, even though it is obvious post fact. In this case, relative debugging let us identify a faulty statement in the ZPL code fragment, even though the syntax and implementation details of the languages completely differ.

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 $10^{-10} \leq \epsilon \leq 10^{-1}$ and rerunning the original assertions now indicates differences between the `$c::int_en` and `$zpl::Int_en` arrays, which store the internal energy values computed in the energy phase. Figure 10 shows these differences.

By defining additional assertions, we can observe that differences in many of the variables involved in the computation of the internal energy have similar characteristics to those in Figure 10. With differences occurring in so many variables, no clear path to the error's source is evident. Instead, we chose to examine initialization code for similar patterns of differences, beginning with the code to initialize the position and velocity components used throughout

```

                                heat.c

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 9. C heat phase code.

the program. Figure 11 shows the C version of this code, which corresponds to lines 154–158 of the ZPL code (Figure 5).

Visualizing 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 relate to these variables. Further examination indicates that the value of PI used in the ZPL code is only specified to seven decimal places, while the corresponding value used in the C code, M_PI , is specified to 20 decimal places (of which only 15 are significant).

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 let us 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

There are still differences in the output of the programs, although the magnitude has now been further reduced to around 10^{-10} . Setting the error tolerance to $10^{-15} \leq \epsilon \leq 10^{-1}$ and rerunning the original assertions shows that the $\$c::\text{heat}$ and $\$zpl::\text{Heat}$ 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`.

The ZPL code uses the $(i+1)$ st column of Theta in the computation $\text{Alpha} * \text{Theta@east} + \text{Beta}$ and propagates this across columns $(\text{DK}-1)$ 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 $(\text{DK}-2)$ to 0, so the $(\text{DK}-1)$ 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 identified 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 $\$c::\text{en_error}$ and $\$zpl::\text{En_error}$ have 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 and line 86 of the C code in Figure 11.

Using relative debugging, we identify a pattern of differences and quickly pin-

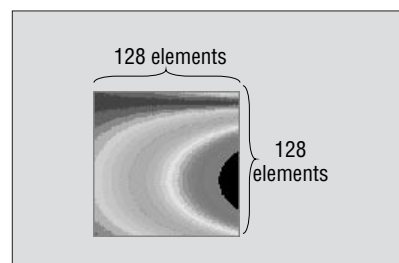


Figure 10. Differences between $\$c::\text{int_en}$ and $\$zpl::\text{Int_en}$.

point the location of erroneous code that is only obvious with the benefit of hindsight. Interestingly, the original paper¹² describing the Simple code gives the error computation as

$$\text{Int_en} + \text{Kin_en} - \text{Work} - \text{Heat}.$$

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 resulted in bitwise equivalence of the variables used in the physics computation. Because nondeterminism was introduced by running the ZPL code in parallel, we must compare the serial ZPL and C codes. For this test, we used the same series of assertions defined over the four phase variables and set the error tolerance 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 section 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. Although it is dangerous to generalize

```

load.c
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 /
104                  (NUM_K_PROCS * DK - 1) + Rmin;
105          w[i][j] = ((M_PI * ((PEi-1) * DL + (DL-i)))
106                  / (2 * (NUM_L_PROCS * DL-1))) + ANGLE_OFFSET;
107          ...
108          x[i][j].r = r[i][j] * cos(w[i][j]);
109          x[i][j].z = r[i][j] * sin(w[i][j]);
110      }

```

Figure 11. C position and velocity initialization code.

the results too far, this example adds to the evidence of our other case studies that supports the power of relative debugging—in the case even when the programs run on different computers and are written in different languages.

DESPITE ITS POWER, comparing data by specifying variable and breakpoint information in assertion statements is still a very low-level approach. It relies on the programmer making sensible choices about the location from which data will be extracted. To compare data, it is not only important that the programs are functionally equivalent at this point, but that breakpoints aren't inserted at locations where the control flow is disrupted or data is in an indeterminate state. This issue is particularly important for arbitrary parallel programs incorporating distributed processes 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 a relative debugger requires, alters the timing of the programs and might mask or highlight timing problems. It might be possible to combine the assertion constructs used here with data-gathering techniques, which are not as invasive as the current debug server, but this also requires further investigation.

Guard99 has been used extensively to examine the relationships between separately executing programs, but there is some evidence that the relative debugging methodology could also be useful for monitoring information in separate *processes* of a parallel or multithreaded program—for example, to ensure that data is initialized correctly or that message buffers are in a consistent state.

Here, we addressed the use of a relative debugger for data-parallel programs. However, we have also implemented a scheme letting the user describe the distribution of data structures for arbitrary task-parallel programs. The system uses a declarative algebraic approach and lets the program ignore the decomposition when

formulating the assertions. It also lets a user describe changes that occur in a data structure and associated code when a program is modified for parallel execution.

Finally, a relative debugger is only one tool that can help find errors. An extremely powerful interactive environment could be built by combining a relative debugger with a call tree browser and dataflow analyzer. Using such a tool would make it easier to trace error propagation through a program, because the user could trace the source of errors in one variable to those that are used to compute the state. In fact, such an approach might facilitate automatic iterative refinement by traversing a program's dataflow. Such an exciting prospect certainly requires further investigation. //

ACKNOWLEDGMENTS

The Australian Research Council, under the Large Grants Scheme, supported this work. We would also like to acknowledge our colleagues Larry Snyder, Bradford Chamberlain, and Sung-Eun Choi at the University of Washington for the assistance they provided in obtaining the results we reported in this article.

References

1. C. Lin and L. Snyder, "ZPL: An Array Sublanguage," *Languages and Compilers for Parallel Computing, 6th Int'l Workshop Proc.*, Springer-Verlag, Berlin, 1994, pp. 96–114.
2. B. Chamberlain et al., "Factor-Join: A Unique Approach to Compiling Array Languages for Parallel Machines," *Workshop on Languages and Compilers for Parallel Computing, Proc. 9th Int'l Workshop*, Springer-Verlag, Berlin, 1997, pp. 481–500.
3. L. Snyder, *A Programmer's Guide to ZPL*, MIT Press, Cambridge, Mass., 1999.
4. D. Cheng and R. Hood, "A Portable Debugger for Parallel and Distributed Programs," *Proc. Supercomputing '94*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1994, pp. 723–732.
5. P. Kacsuck et al., "A Graphical Development and Debugging Environment for Parallel Programs," *Parallel Computing*, Vol. 22, No. 13, 1997, pp 1747–1770.
6. J. May and F. Berman, "Panorama: A Portable Extensible Parallel Debugger," *SIGPLAN Notices*, Vol. 28, No. 12, San Diego, 1993, pp. 96–106.
7. *TotalView Multiprocess Debugger Users' Guide Version 3.7.7*, Dolphin Interconnect Solutions, Rev. 8, Oct. 1997.

8. T. Bemmerl and R. Wismüller, "On-Line Distributed Debugging on Scaleable Multicomputer Architectures," *High Performance Computing and Networking, Volume II: Networking Tools, Lecture Notes in Computer Science*, Vol. 797, Springer-Verlag, New York, 1994, pp. 394–400.
9. D. Abramson, R. Sasic, and G. Watson, "Implementation Techniques for a Parallel Relative Debugger," *Proc. PACT '96*, IEEE Computer Soc. Press, Los Alamitos, Calif, 1996.
10. J. Rose and G. Steele Jr., *C*: An Extended C Language for Data Parallel Programming*, Tech. Report PL 87-5, Thinking Machines Corp., Cambridge, Mass., 1987.
11. R. Stallman, *Debugging with GDB—The GNU Source Level Debugger, Edition 4.12*, Free Software Foundation, Boston, 1994.
12. W. Crowley, C. Hendrickson, and T. Rudy, *The SIMPLE Code*, Lawrence Livermore Laboratory, Livermore, Calif., UCID-17715, 1978.
13. D. Abramson and R. Sasic, "A Debugging and Testing Tool for Supporting Software Evolution," *Automated Software Eng.*, Vol. 3, Nos. 3–4, Aug. 1996, pp. 369–390.
14. D. Abramson and R. Sasic, "A Debugging Tool for Software Evolution," *CASE-95, Seventh Int'l Workshop Computer-Aided Software Engineering, Proc. 7th Int'l Workshop Computer-Aided Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1995, pp 206–214.
15. D. Abramson et al., "Relative Debugging: A New Methodology for Debugging Scientific Applications," *Comm. ACM*, Vol. 39, No. 11, Nov. 1996, pp. 69–77.

Greg Watson is a Senior Research Fellow at the Monash University School of Computer Science and Software Engineering. His research interests include programming tools for parallel and distributed computers, and parallel languages and runtime systems. He is also interested in operating system design, nano technology, and AI. He is currently Vice-President of the Internet Society of Australia and Chair of the Australian .au Domain Administration (auDA). Contact him at greg@csse.monash.edu.au

David Abramson is head of the School of Computer Science and Software Engineering at Monash University, Australia and is project leader in the Cooperative Research Centre for Distributed Systems Nimrod Project and the Australian Research Council funded Guard project. His current interests are in high performance computer systems design and software engineering tools for programming parallel and distributed supercomputers. Contact him at davida@csse.monash.edu.au.