

# Parallel Relative Debugging with Dynamic Data Structures

David Abramson †, Raphael Finkel ‡, Donny Kurniawan †, Victor Kowalenko †, Greg Watson §

† School of Computer Science  
& Software Engineering,  
Monash University, Clayton  
Australia, 3800  
[davida@csse.monash.edu.au](mailto:davida@csse.monash.edu.au)

‡ Department of Computer Science  
University of Kentucky  
Lexington, Kentucky  
U.S.A.  
[raphael@cs.uky.edu](mailto:raphael@cs.uky.edu)

§ Advanced Computing Lab  
Los Alamos National Laboratory,  
Los Alamos, NM, 87545,  
U.S.A.  
[gwatson@lanl.gov](mailto:gwatson@lanl.gov)

## Abstract

*This paper discusses the use of “relative debugging” as a technique for locating errors in a program that has been ported or developed using evolutionary software engineering techniques. It works on the premise that it is possible to find errors by comparing the contents of key data structures at run time between a “working” version and the new code. Previously, our reference implementation of relative debugging, called Guard, only supported comparison of regular data structures like scalars, simple structures and arrays. Recently, we augmented Guard enabling it to compare dynamically allocated structures like linked lists. Such comparisons are complex because the raw values of pointers cannot be compared directly. Here we describe the changes that were required to support dynamic data types. The functionality is illustrated in a small case study, in which a parallel particle code behaves differently as the number of processors is altered.*

## 1 Introduction

In 1994 Abramson and Sosic introduced a new debugging and testing strategy called “Relative Debugging”. Relative debugging allows a user to debug a faulty program against a working version, possibly running on another machine, in another language, and on a different operating system. It is particularly valuable when programs are ported from one platform to another, because it allows a user to locate the erroneous region very quickly using a divide-and-conquer strategy. Relative debugging is particularly useful when codes are ported to parallel computers, because subtle errors are often introduced at this stage. We have built a tool called “Guard” that implements the key features of relative debugging in addition to conventional debugging primitives.

Relative debugging works by allowing a user to compare the contents of various data structures between executing codes at particular times. Using this approach, it is possible to determine at which point a new version of the code diverges from an existing “reference” code. We have shown using many different case studies that relative debugging is a powerful technique [1] [2] [3] [4] [5] [12] [14] [16] [15].

To date, our focus has been on comparing the contents of static structures like scalars and arrays. This limitation has been partly because the applications of interest to us, namely scientific codes, make extensive use of static structures. In addition, it is fairly easy to see how to extract and compare array structures between two running codes, because arrays are usually regular, and their representations do not change much between different computer architectures or even languages. In order to cater to applications in which the shape may change as the code evolves, Guard implements a powerful algebraic specification language. This language makes it possible to describe how the data structure has changed, and so the debugger can use this mapping to match up the variables at run time.

Comparing dynamic data structures between programs is significantly more difficult than for static arrays, because we wish to compare the contents of the structures without using absolute values like pointers. In fact, we actually want to build a “normalized” structure independent of the machine and the way it stores dynamic pointers. The approach we have adopted is to first serialize the data structures, which might be arbitrarily large and may include cycles. The result has two parts: a type description and a data description. We then compare the type descriptions for equivalence. If they are equivalent, then we compute the difference between the two data descriptions. That difference is presented to the user.

This paper begins with a discussion of relative debugging, and in particular, its implementation in Guard. We then describe how we have implemented the comparison of dynamic structures, including the extensions to the machine-independent type system we have previously developed called AIF. Finally, we illustrate the new work by applying it to the debugging of a parallelized code based on SPH, a technique that is used widely in solving problems in computational fluid dynamics (CFD). In this case study, the SPH code produced different results when running on parallel machines with different numbers of processors. Guard was able to help us locate the source of this divergence and correct it.

## 2 Relative Debugging and Guard

A relative debugger like Guard implements a standard set of commands for controlling the execution of a program and for examining the state. In addition, it allows a user to launch and control more than one program at a time, possibly on different physically distributed platforms. Data comparison is facilitated by a Guard command that allows the user to define an *assertion* between data structures. An assertion specifies a pair of tuples, each containing a program name, a data structure identifier and a line number, as shown below:

```
assert prog1::var1@line1 = prog2::var2@line2
```

In this example, the assertion indicates that `var1` and `var2` should have the same values at `line1` in `prog1` and `line2` in `prog2`. In order to implement an assertion, Guard places breakpoints at the relevant line numbers. When the breakpoints are hit, Guard extracts the data from the named variables, and performs a comparison operation. The execution-control mechanism is actually much more complex than described here, and is based on a powerful data-flow interpreter [5]. This mechanism ensures that breakpoint events, which can occur asynchronously and in any order, are managed correctly. Accordingly, each assertion is compiled into a data-flow graph. Guard can support any number of assertions concurrently.

Data structures do not have to be exactly the same. For example, differences in floating-point numbers are compared within a tolerance. Moreover, the data structure types do not need to be identical, as long as Guard knows how to coerce one type into another. Thus, it is possible to compare a `short int` against a `long int`. Arrays can actually be different shapes, as long as there is a unique and regular mapping from every element of one array to another. For example, it is possible to compare a vector with a two-dimensional array as long as the user can describe the transformation between the two structures. This transformation is described using an algebraic specification language, which is then interpreted by Guard as it maps one structure to another [15].

Differences between the contents of structures can be reported using various techniques. For example, for low dimensioned data, or scalars, it is generally adequate to report the values and the differences. For higher ordered data structures like multidimensional arrays, it is possible to use scientific visualization packages like Open DX [8]. Such visualizations can be very helpful in determining why two programs produce different data, because it is possible to highlight the nature of the divergence. Previously, our experience was in visualizing multidimensional floating-point arrays [1][2]. In this

paper we use some new visualizations more suited to dynamic data.

## 3 The Architecture Independent Form (AIF)

Guard is machine and architecture independent. It can extract data from two different machines and manipulate and compare the data on a third. In order to provide this functionality, we have built a machine and language independent representation for data types called Architecture Independent Form (AIF) [15]. AIF represents types using string expressions. These expressions can then be stored and manipulated as simple strings, and the data they represent can be manipulated using a set of portable routines. As a result, we can perform operations on data types that are not supported by the base architecture, for example, adding 64-bit integers on 16-bit processors, or manipulating big-endian quantities on little-endian machines. AIF facilitates the high degree of machine heterogeneity provided by Guard.

### 3.1 Type descriptions

An AIF representation has two parts, namely the type descriptor and the data descriptor. Type descriptors are represented in ASCII and consist of a single letter followed, if necessary, by modifiers indicating precision and sign and any subordinate type descriptors. Table 1 shows the currently supported set of AIF types. The items above the double lines were supported in previous versions of AIF [5] [15], and the new types below the double line have been added specifically to support dynamic and structured data.

The atomic types are straightforward. The void type, described by the type descriptor `v`, has a 0-length associated data descriptor. A character is always a single byte of data, so its type descriptor, `c`, gives no size information.

Strings are assumed to be null-terminated, so their type descriptor, `s`, also lacks size information. The integer 3 could be represented by type descriptor `is1` (one-byte signed integer), `iu8` (8-byte unsigned integer), or a variety of alternatives. The associated data descriptor has the specified number of bytes with the integer 3 stored in big-endian format. There are no inherent restrictions on precision, although some precisions cannot be converted to the native integer representation on some architectures. Floats are similar, but they are harder to render in a canonical form. We have chosen to store the data bytes in big-endian order but otherwise do not interpret the bits of mantissa and exponent. The result is that floats are not as machine-independent as we would prefer; which needs to be addressed.

Letter	Modifiers	Description
v		Void
c		Character
s		String
i	s or u, b	signed/unsigned integer, <i>b</i> bytes
<		enumeration
f	<i>b</i>	float, <i>b</i> bytes
[	<i>l</i> .. <i>hd<sub>1</sub></i> <i>d<sub>2</sub></i>	array with bounds <i>l</i> .. <i>h</i> , index type <i>d<sub>1</sub></i> , base type <i>d<sub>2</sub></i>
{	<i>n<sub>1</sub>:d<sub>1</sub></i> , <i>n<sub>2</sub>:d<sub>2</sub></i> , ... }	record with field <i>i</i> named <i>n<sub>i</sub></i> of type <i>d<sub>i</sub></i>
^	<i>d</i>	pointer to a component with type descriptor <i>d</i>
%	<i>n/d</i>	component names <i>n</i> with type descriptor <i>d</i>
>	<i>n/</i>	reference to component named <i>n</i>

Table 1 – AIF type descriptions

Arrays are all considered one-dimensional. As in Pascal, multidimensional arrays can be represented as arrays of arrays. The type descriptor starts with [ and includes two numbers indicating the low and high bounds and two subordinate type descriptors, one for the index type and the other for the base type. The index type must be an integer or enumeration so the bounds make sense. A character array of two elements, containing “a” and “b”, could have type descriptor [0..1is4]c; the associated data descriptor would be two bytes long and contain the two characters.

Structured types are more complex. Pointers to data are described with the ^ type descriptor, which takes as a modifier a type descriptor for the base type. A pointer to a pointer to the character “a”, for instance, has the type descriptor ^^c. Pointer data types require one byte of data in addition to whatever data the base type requires. The extra byte of data has a special meaning: 0 indicates a null pointer, 1 indicates a non-null pointer, and other values are discussed shortly. In this example, the data descriptor is three bytes long, containing 1, 1, and “a”. It does not contain any machine addresses involved in the double indirection, because that information is not germane to the value.

Records have an arbitrary number of named fields. Their type descriptor, which starts with {, has an arbitrary number of comma-separated (name, type descriptor) pairs. Names are ASCII strings. The amount of data associated with a record is dependent on the amount of data required by each field.

Data structures can be recursively defined. For example, a linked list of integers has a type that depends on itself: {a:is4,b:^{a:is4,b:^{+a:is4, ...}}. The

recursion is broken by introducing a name such as n1 for the replicated part and referring to it, giving us the compact representation %n1/{a:is4,b:^>n1/}. In this type descriptor, the characters / and : are used as delimiters to end names such as a, b, and n1.

Components can also be named when substructures have the same type. For example, a binary tree might have type descriptor %n2/{left:^>n2/,right:^>n2/}.

Data itself can be circular when there are pointers. For instance, a linked list such as the one named n1 above might have one element that points to itself. The special pointer values are extended in order to cover this case. A pointer value of 2 indicates a forward reference, and the subsequent 4 bytes are reserved to hold a reference name. A pointer value of 3 indicates a backward reference in the data stream. Again, the subsequent 4 bytes holds the reference name. A pointer value of 4 indicates an invalid pointer, in which case no data is stored. A linked list of type n1 with a single node containing 8 and a pointer to itself might be represented by this data stream: 2 (named pointer value), 0, 0, 0, 1 (name), 0, 0, 0, 8 (four bytes for the integer 8), 3 (named pointer reference), 0, 0, 0, 1 (name).

Named data references are also useful when the same values appear in several places within the same data structure. A binary tree with common sub-trees can make use of named data references to reduce the length of its data representation, although such reduction is not necessary to render the data representation finite.

### 3.2 AIF routines

The AIF library contains conversion routines to construct AIF representations from native data of various types, including constructors that build representations of structured types from subordinate structures. To support circular types, the library provides routines for adding names to existing type descriptors. However, it does not contain methods to search down data structures and discover circularities; that task is performed by other software. A second set of conversion routines turn AIF representations into native representations.

The library includes basic arithmetic routines, such as mixed-mode division, which takes two operands in AIF representation and produces a result in AIF representation. The operands may be arbitrary data structures with compatible type descriptors. All integers, of whatever precision, are considered compatible, for instance, but arrays of 2 elements are not compatible with arrays of 3 elements. Arithmetic makes no sense except for arithmetic component types. However, two routines are more general: Two values may be compared for equality, and a value can be tested for equivalence to zero.

We use a recursive definition of comparison and zero-equivalence for compatible structured types, based on component-wise testing. Finally, the AIF library includes printing routines that present AIF representations in human-readable form.

### 3.3 Gdb extensions

Guard is implemented using a client-server architecture. The Guard client implements the logic required for relative debugging and communicates with one or more debug servers. Each debug server is responsible for operations that control the end application, such as setting breakpoints and examining variable data. In turn, each Guard debug server is implemented using `gdb` for the core services. `Gdb` is a standard debugger for imperative languages such as C, C++, and Fortran [13]. Compilers for supported languages produce debugging information in their executable files that `gdb` can interpret to associate source-line numbers with execution points and names and types with values. We have added a few features to `gdb` to make use of the features of the AIF library.

`Gdb` behaviour can be controlled by a set of run time parameters. For example, the user can enter `set print address` to request that `gdb` include the address of every datum that it prints. We have added two new settings.

First, `set print depth n` tells `gdb` to follow pointers to a depth of `n` (default 0) so the user can see the values that pointers reference. We have found that setting the print depth to about 3 makes ordinary program debugging much easier. In this case, any time we print a data structure, we see up to three levels of indirection with a single print command. This `gdb` enhancement is independent of the AIF library.

Second, `set print aif on` (default off) modifies the way `gdb` outputs all values. It uses the AIF library to construct AIF representations of values, going as deeply as necessary, and detecting data circularities as it goes, after which it outputs the type and data descriptors. The data representation is output in hexadecimal. This feature is of no use to the ordinary user, but it is crucial for Guard because Guard only communicates with the debug servers using AIF.

## 4 Smooth Particle Hydrodynamics Application

Smoothed/Smooth Particle Hydrodynamics, or simply SPH, is a technique that simulates non-axisymmetric fluid flows and particle motion in diverse subject areas such as astrophysics, fluid mechanics, explosion dynamics and geodynamics.

SPH is a particle method, but unlike typical Particle-In-Cell [7][6][9][10][11] methods based on Eulerian and Lagrangian approaches, it does not require a grid for calculating spatial derivatives in partial differential equations. Instead, derivatives are expressed as interpolation formulae involving a kernel function that is evaluated over a set of disordered points referred to as “particles”.

The number of particles required to obtain accurate approximations of the interpolation formulae is reduced substantially by choosing a kernel that decreases rapidly with distance. Then the interpolation formulae become finite sums over nearest-neighbour “particles”.

Although many types of kernels can be created, analogous to using different schemes in finite difference methods, the most common kernels are either spline-based or Gaussian. The former are more computationally efficient, while the latter provide a physical interpretation of an SPH equation. All kernels, however, are dependent upon an additional parameter `h`, which is referred to as the resolution.

The problem of determining nearest-neighbour particles can be overcome by dividing the problem into cells and only considering a limited number of cells near the home cell containing the particle of interest. The size of these cells depends upon the resolution. This layout is shown in Figure 1.

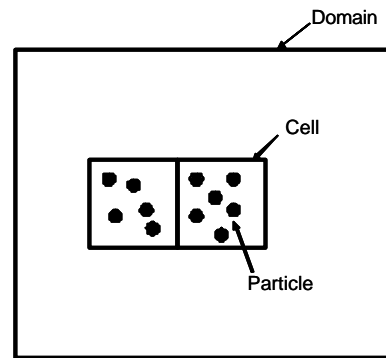
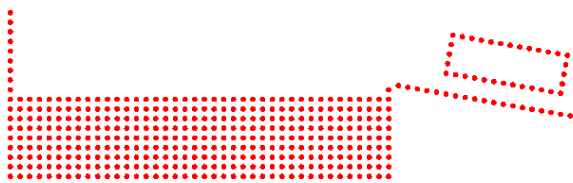


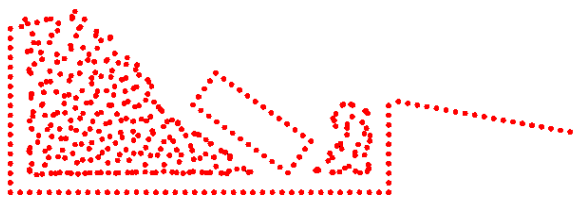
Figure 1 – Layout of Particles and cells in SPH Domain

For a description of the mathematics of the SPH technique, the reader is referred to [10][11].

SPH allows us to simulate a variety of different real world problems. In this paper, we describe the simulation of a fluid flow in a tank containing a solid block, as shown in Figure 2. Figure 2a shows the initial configuration of the block, fluid and particles in the fluid, whereas Figure 2b shows the final configuration. The simulation is able to describe accurately the motion of both the solid block the fluid in the tank.



(a)



(b)

Figure 2 – Initial and Final positions of a block impacting into a tank of fluid, as modeled in SPH

#### 4.1 Implementation issues

The SPH algorithm requires the storage and manipulation of the mass, position, velocity, thermal energy and initial density (sometimes not required) of each particle in the system. Further, every particle must keep track of its neighbours in the same cell. Accordingly, all of the particle properties are stored in a structure, a simplified version of which is shown in Figure 3, and neighbouring structures in the same cell are linked together using a single pointer chain.

```
typedef struct pointst {
    VECTOR    position;
    VECTOR    force;
    VECTOR    velocity;
    double    mass;
    :         :
    :         :
    pointst  *next; }

pointst  *all_particles[MAX_PARTICLES];
```

Figure 3 – SPH Particle Structure in C

The parallel implementation uses the same basic data structures, except the particles are distributed across the processes using a conventional (one-dimensional) domain decomposition; each processor only holds a subset of the

particles. A single process controls all of the particles in one grid cell, and the process also controls multiple grid cells. During each time step of the simulation, each process sends the physical data for those particles in the cells located at the boundaries to the adjacent processes. In addition, the processes exchange the data for those particles that have moved from their home cell to a neighbouring cell. The physical data for the particles in the other processors is stored in ghost bands around the domain.

Figure 4 presents the pseudo code for an SPH computation.

1. **Initialise** positions of particles;
2. **Initialise** the domain decomposition;
3. **While** not end-of-simulation **do**
  - a. **Exchange** particle positions with neighbouring processors;
  - b. **Perform** the predictor part of the algorithm;
  - c. **Compute** cell composition;
  - d. **Setup** linked list of particles in cells;
  - e. **Perform** the SPH calculation
  - f. **Perform** the corrector part of the algorithm;
4. **End loop**

Figure 4 – Pseudo code for SPH calculation

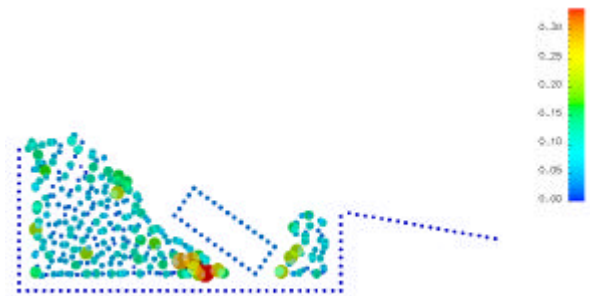
#### 4.2 Debugging SPH

During the parallelization of the original SPH calculation, we noticed that the results differed as we changed the number of processors in the decomposition. The differences were not large, so we attributed them to minor floating-point errors. This type of error is not uncommon when an algorithm is parallelized, because the order of partial sums and other reduction operators can vary, causing slight changes in the computation. Moreover, we did not expect more significant errors, because the parallel algorithm was essentially the same as the sequential one, and the numeric methods were not altered to take account of the parallelization.

Prior to the enhancements to Guard discussed in Section 4, it would not have been possible to use Guard to trace the source of this discrepancy. The particles are stored in a C structure, and these structures are linked together using a pointer chain. Furthermore, the array of all particles is actually an array of pointers to particle structures. However, the new features in Guard let us not only specify the type declaration, but also perform operations such as COMPARE on two different versions of the program. The normalization of the pointer chains is particularly important, because the virtual address values in the pointers almost certainly differs as the number of processors in the parallel algorithm is altered.

Our aim was to determine the cause of the divergence. Previous case studies have indicated that programmers have been too eager to attribute different behaviour in parallel programs to the order of reduction operators. In one such case study, we actually found four independent coding errors in the programs that were previously attributed to floating-point differences [14]. Debugging SPH was simply a matter of placing assertions on the key data structures until the source of the error could be located. In addition to assertions, we used scientific visualization techniques to highlight differences in the data. Again, previous case studies [1][2][3][4][14] have demonstrated the power of such visualization.

Initially we ran the two programs, one with 2 processors and the other with 4, and placed an assertion on the main data structure representing all\_particles at the end of the computation (after step 4 in Figure 4). Figures 5 (a) and 5 (b) show the two different final particle configurations, . Figure 5 (c) highlights the difference by scaling the glyphs for each particle according to the size of the Euclidean distance between corresponding particles in each version of the code. This is a better approach for examining the difference in particle positions.

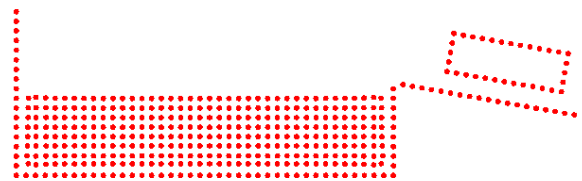


Difference (c)  
Figure 5 – Final particle configuration

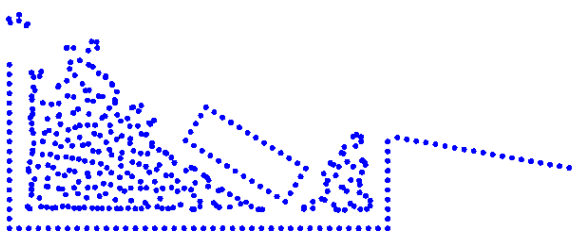
Figure 6 shows the same data but after the first time step only (after step 3f in Figure 4). Although it is extremely difficult to see the difference between the data by examining Figures 6(a) and 6(b), the difference is clearly seen in Figure 6(c).



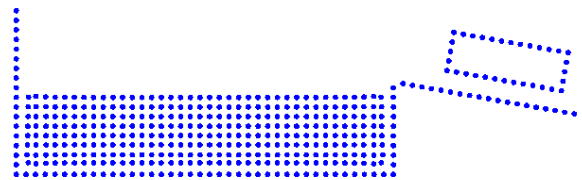
2-processor version (a)



2-processor version (a)



4-processor version (b)



4-processor version (b)



Difference (c)

Figure 6 – Particle configuration after one time step

Further assertions helped us refine the region in which the computations diverge. Specifically, Figures 7(a) and 7(b) show the particles and the linked lists that indicate neighbouring particles. In these visualizations, a link is shown by drawing a line from one particle to another. Although it is difficult to see the difference by observing the images, Figure 7(c) highlights the difference by colouring the links black if they are identical in the two codes; otherwise the two different links are superimposed on each other. At this point, the visualisation highlights the problem, because the black regions correspond to particles allocated to processors 0 and 2, whereas processors 1 and 3 have differences. Further examination of the code, indicated that the computation which determined the composition of particle cells is dependent on the number of processors, and thus each version of the code generates a different cell structure. To ensure that the two programs generate identical solutions we modified the calculation so that the cell composition was independent of the actual number of processors. Importantly, Guard was instrumental in localizing the region that was responsible for the different behaviour in the two versions of the code.

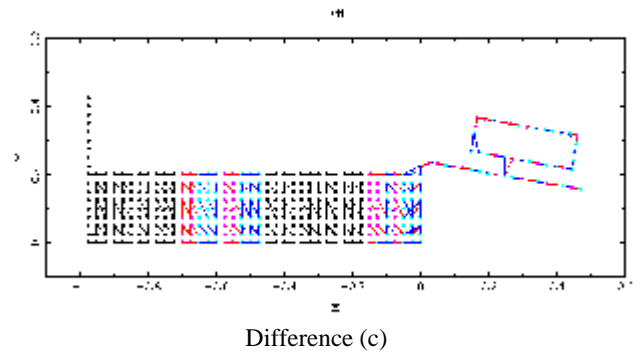
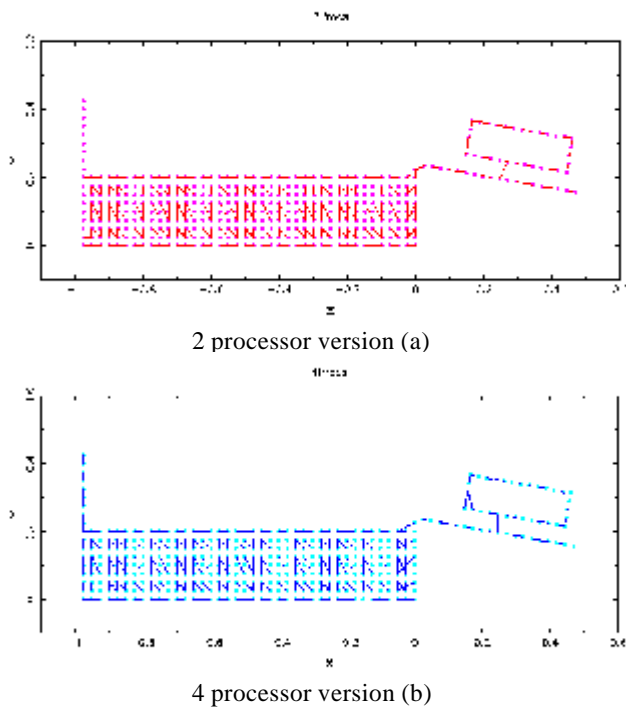


Figure 7 – Inter particle links

## 5 Conclusions

This paper has described the enhancements to Guard to enable it to represent, manipulate and compare complex dynamic data structures. In particular, we have described the changes required to the Architecture Independent Form (AIF), including normalization of memory address pointers. These enhancements are novel and can be used in other tools that manipulate such dynamic structures.

We have described the application of relative debugging for determining why two different versions of an SPH program generated different results. Prior to performing the case study none of the application developers was able to explain the discrepancy in these results, and were of the belief that the divergence was due to different partial orderings in the parallel computations. However, Guard enabled us to demonstrate that the contents of important data structures were actually different in both versions, and that the difference was due to a single variable, namely the number of processors being used. When we altered this critical computation, we could force the two programs to compute exactly the same result. This outcome is important, because developers frequently attribute floating point issues as the cause for the different behaviour without a rigorous examination of the actual cause. Guard is an important tool because of the relative ease with which it can perform the comparisons between different versions of codes.

The paper also highlighted the importance of scientific visualization techniques in showing where the contents of data structures differ. In this work we have presented new visualizations that show small differences in particle position that cannot be detected by simple visual inspection of the particle configurations. Furthermore, we were able to show the dynamic links between particles, and the differences in these links.

At present, it is necessary to produce new visualization scripts for each problem, and this can be time consuming. In the future, we would like to achieve better integration

between Guard and production scientific visualization packages.

### Acknowledgements

This work is supported by a research grant from the Australian Research Council. We wish to acknowledge support from the Monash Cluster Computing laboratory (MC<sup>2</sup>).

### References

- [1] Abramson D., Foster, I., Michalakes, J. and Sosic R., "Relative Debugging: A new paradigm for debugging scientific applications", the Communications of the Association for Computing Machinery (CACM), Vol 39, No 11, pp 67 - 77, Nov 1996.
- [2] Abramson D., Foster, I., Michalakes, J. and Sosic R., "Relative Debugging and its Application to the Development of Large Numerical Models", Proceedings of IEEE Supercomputing 1995, San Diego, December 95.
- [3] Abramson, D.A. and Sosic, R. "A Debugging and Testing Tool for Supporting Software Evolution", Journal of Automated Software Engineering, 3 (1996), pp 369 - 390.
- [4] Abramson, D.A. and Sosic, R. "A Debugging Tool for Software Evolution", CASE-95, 7th International Workshop on Computer-Aided Software Engineering, Toronto, Ontario, Canada, July 1995, pp 206 - 214. Also appeared in proceedings of 2nd Working Conference on Reverse Engineering, Toronto, Ontario, Canada, July 1995.
- [5] Abramson, D.A., Sosic, R. and Watson, G. "Implementation Techniques for a Parallel Relative Debugger ", International Conference on Parallel Architectures and Compilation Techniques - PACT '96, October 20-23, 1996, Boston, Massachusetts, USA
- [6] Benz, W. "Computer Physics Communications", **48**, 97 (1988).
- [7] Gingold, R.A. and Monaghan, J.J. "Monthly Notices of the Royal Astronomical Society", **181**, 375 (1977).
- [8] <http://www.opendx.org/>.
- [9] Lucy, L.B. "Astronomical Journal", **82**, 1013 (1982).
- [10] Monaghan, J.J. "Annual Reviews of Astronomy and Astrophysics", **30**, 543 (1992).
- [11] Monaghan, J.J. "Computer Physics Reports", **3**, 71 (1985).
- [12] Sosic, R. and Abramson, D. A. "Guard: A Relative Debugger", Software Practice and Experience, Vol 27(2), pp 185 – 206 (Feb 1997).
- [13] Stallman, R. Debugging with GDB – The GNU Source Level Debugger, Edition 4.12, Free Software Foundation, January 1994.
- [14] Watson, G. and Abramson, D. "Relative Debugging For Data Parallel Programs: A ZPL Case Study", IEEE Concurrency, Vol 8, No 4, October 2000, pp 42 – 52.
- [15] Watson, G. R., "The Design and Implementation of a Parallel Relative Debugger", PhD Thesis, Monash University, Melbourne, October 2000.
- [16] Watson, G. and Abramson, D. "The Architecture of a Parallel Relative Debugger", 13th International Conference on Parallel and Distributed Computing Systems - PDCS 2000, August 8 - 10, 2000.