Ad Hoc Visualization of Distributed Arrays

Alan H. Karp
Ming C. Hao
HPL-93-72
September 8, 1993

Parallel processing, debugging, visualization

We have incorporated data visualization in a debugger for parallel programs written with PVM. We use features of the debugger to write the requested data to a file for each PVM task, provide a simple menu for the user to describe the data distribution, and create a single file with the data in natural order. The user then invokes any existing visualizer to look at the data.

1 Introduction

Existing debuggers are fine tools for finding errors in simple programs; they are inadequate when dealing with large, scientific codes. The problem is in looking at the data. Today's debuggers allow the user to look at the program variables one number at a time or at arrays as a whole. As long as the arrays are small or the errors obvious, this view is all we need. As soon as we need to find a subtle error in an array of millions of elements, we are out of luck.

Only recently have vendors started to provide the kind of views of their data that applications programmers need. The Prism debugger[9] for the Connection Machine allows the programmer to view the data in a number of different ways, including surface plots and color maps. The debugger from MasPar[8] does gray scale and color maps. An interface to AVS[11] is provided for more sophisticated visualizations.

There are a number of data visualizers available, ranging from the comprehensive to the simple. Virtually all of these tools can read the data to be displayed from a disk file. Many debuggers have the ability to dump the output of a display command to a file instead of to the screen. Hence, a program variable can be visualized by using the debugger to write the data to a file and then invoking a visualizer. However, this simple scheme doesn't work well for irregularly spaced data, particularly in more than one dimension.

The problem is even worse for parallel programs. The current technology of debugging a distributed memory multi-computer consists of invoking an instance of the debugger for each task. If we dump the part of the array held by each node to a separate file, we can invoke a visualizer, but we may not be able to see what we need. It all depends on how the data is distributed.

A variety of data distributions are often used[1, 2, 7]. An array distributed by blocks has contiguous elements assigned to each task; one distributed cyclically has elements dealt in round-robin fashion as in a card game. The data can also be distributed in a block-cyclic manner where contiguous blocks of elements are dealt out to tasks, round-robin assignment being used for the blocks. Additionally, each dimension of a multidimensional array can have a different distribution. Even for block distribution in one dimension, finding the error in a distributed array may be difficult.

As an example, consider a one dimensional array of length 100 distributed over 4 nodes in blocks of 5 elements. Figure 1 shows a plot of the data held on each node. The array should look like a cosine function, but we purposely introduced a small error. It is not obvious where the error is.

If the application required visualization of this array, the programmer would have written the code necessary to gather the data together and plot it. During a debugging session, though, we don't know ahead of time what arrays the user will want to see. Hence, we want

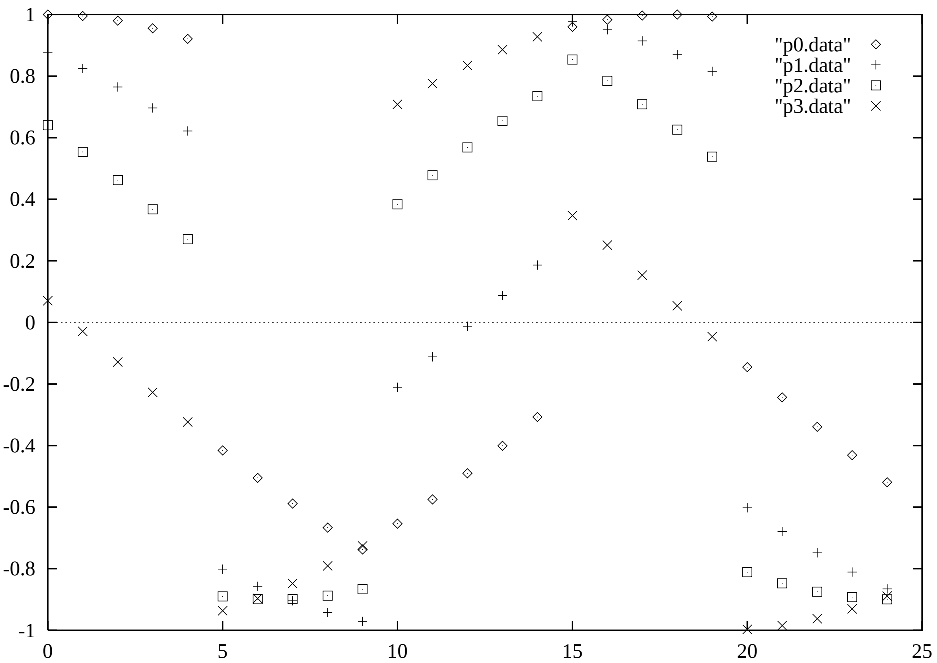

Figure 1: An array of length 100 distributed over 4 nodes with a block size of 5 has been dumped to 4 disk files and plotted with gnuplot[12]. Although there is an error in the data, it is not immediately apparent.

an *ad hoc* scheme that gathers the data without requiring the user to modify the application.

The work described here is part of an effort to build an Interactive Visualizing Debugger (IVD) for distributed memory, parallel programs[5]. A key feature of IVD is that it is independent of the underlying tools. For example, any performance visualizing system can be used without modification. In keeping with this philosophy, we have designed the data visualization to work with existing debuggers and data visualizers in the same way that users have been visualizing their data on uniprocessors. We provide simple schemes to dump the data held by each node to disk and for the user to describe the data distribution. Next, we use the data distribution to produce a single file containing the data to be visualized and invoke the user's favorite data visualizer.

2 Reconstructing the Data

In order to put the data back together again, we need some information. First, we need to know which task produced which output file. Next, we need to know which part of the array was held by each task. Finally, we need to know the sequence in which the tasks were assigned.

This last point is worth explaining. We are working with programs written using PVM[10] for distributing work over a cluster of workstations. PVM programmers make a call to a routine to spawn new tasks. This routine returns an integer containing a unique identifier for each task. If I want to send a message to the 7'th task, I would code `pvm_send(tid[7],...)`. This is the way the programmer thinks of the tasks – `tid[7]`, not the actual task id contained in the `tid` array. We must allow the user to describe the data distribution in terms of the index in the task array, not the numeric values.

There are two possible ways to determine which file was produced by which task. If our version of PVM is being used, the spawn routine captures the order of the tasks in the task array and informs IVD. If this modified version of PVM is not available, the user must initialize the data visualization by having the debugger controlling each task write that task's `tid` to its output file. The user also dumps the contents of the `tid` array from one of the tasks. We can now read the individual `tid`s and determine the index of each task in the array.

Next, we need to find out how the data is distributed. Our present implementation assumes that the array is distributed in a regular grid with arbitrary blocking in each of up to 4 dimensions, allowing us to follow the time evolution of 3 dimensional objects. For each dimension, the user gives the length, the block size, and the number of tasks. For the example we are using, there is one dimension of length 100. There are 4 tasks and 5 contiguous elements are given to each tasks in turn, *i.e.*, the block size is 5. A pure, block distribution would have a block size of 25; a pure cyclic one, a block size of 1. In addition, we allow the segments to overlap by an arbitrary amount. When the segments overlap, we accommodate toroidal distribution by allowing the last segment in each dimension to overlap the first.

Figure 2 shows the menu presented to the user. The data requested is similar to that needed to describe a data distribution in ScaLAPACK[3]. There are several features of note. Since several arrays are often distributed the same way, we save data distributions on disk. When data visualization is requested, a list of all saved data distributions is displayed. If one is selected, the menu is filled in with the corresponding values. The user may then change any of the values, including the distribution name, and save the new definition. A default distribution is used if the user does not select one from disk.

Most debuggers will dump the data to disk in storage order. Since C stores data in row major order and Fortran uses column major order, we need to know the number of dimensions and the storage order. In addition, some data distributions can not be described in our notation, so we allow the user to specify an explicit distribution. In this way we can let the user visualize arrays which do not have regular data distributions.

We also allow for cases in which some tasks hold no part of the array. For example, in a master-slave program, the master task usually does not participate in the calculation and holds no relevant data. We allow the user to select a subset of the tasks to reflect this type

Distribution name	block5		
Number of dimensions	3		
Order	Row	Column	Explicit
Group	Slaves		
Read	Save	Use	

Dimension	1	2	3	4
Length	100	100	50	—
Block Size	5	6	50	—
# Tasks	4	5	1	—
# Overlapped	0	2	0	—
Overlap ends?	—	y	—	—

Figure 2: A data distribution menu filled out for a three dimensional array. See the text for a description of the various distributions.

of distribution using the grouping feature of IVD, similar to that in the proposed message passing interface[4].

While this simple scheme doesn't cover all possible data distributions, most of the common ones can be described. (Explicit distributions described below cover the rest.) For example, in Figure 2 we see that dimension 1 is distributed in blocks of 5 elements over 4 tasks with no overlap. Dimension 2 is in blocks of 6 elements over 5 tasks with neighboring tasks sharing 2 elements. Since periodic overlap is specified, the last task shares 2 elements with the first. The third dimension is not distributed. Since only 3 dimensions were requested, we do not allow the user to enter data for Dimension 4. Figure 3 illustrates an array distributed in two dimensions over 4 tasks with overlap in both directions. Note that the points in the intersections of the shared blocks are held by 4 tasks.

An explicit data distribution provides a lot of flexibility. It is implemented using a special form of the data distribution description saved on disk. If the user specifies that the order is `explicit`, then we assume the rest of the file tells us where to find the data. Each subsequent line in the distribution file for an explicit description contains the index of the task holding a word in the complete array and the coordinates along each dimension. While the entries for data held by different tasks can be interleaved in any order, entries for data held by a single task must appear in the order they are dumped by the debugger.

For example, 8 elements of a one dimensional array distributed over 3 tasks could have the explicit distribution shown in Figure 4. Since the data to be visualized are not equally spaced, we have used the explicit form to allow us to associate the value of the independent

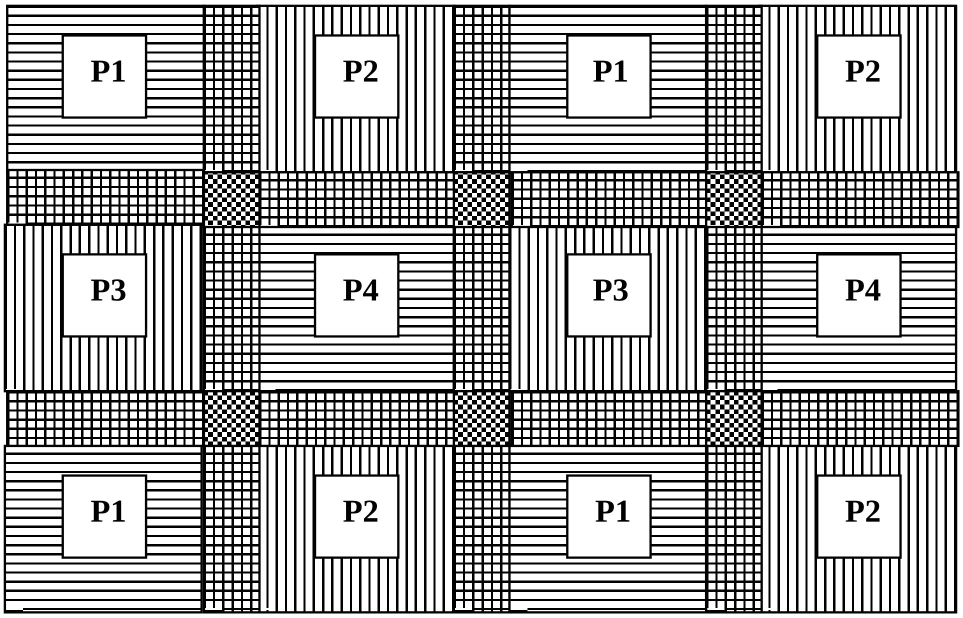

Figure 3: Two dimensional array distributed over 4 tasks with overlap in both directions.

variable with each data point. We can also use an explicit distribution to allow us visualize an array distributed so as to improve the load balancing. Note that the explicit distribution need not be known before run time if the user's code produces a valid data distribution description.

We now have all the data in files, one per task; we know which file corresponds to each entry in the task id array; and we know the data distribution. We can now read the individual files and produce a single file with the data in the same order the debugger would have produced had we dumped it during a uniprocessor run.

3 Examples

Figure 1 shows how difficult it can be to find a small error by looking at the pieces of the array held by each task. Figure 5 shows the entire array reconstructed using the procedure described in Section 2. The data points in error are easy to spot. Simply counting the points tells me that task 2 is the culprit. We can now go about identifying the source of the error.[1] A more sophisticated visualization could have used a different color or plot symbol for each task because the output file we create has the task id associated with each data point.

[1] In this case it is easy. We added 0.1 to all the points computed by task 2.

```
Data distribution name:  scattered
Number of dimensions  :  1
Order (row/col/exp)   :  e
 0   -0.5
 0   -0.2
 1    0.0
 1    0.2
 2    0.25
 2    0.5
 2    0.9
 1    0.8
```

Figure 4: An example of an explicit data distribution of a one dimensional array. The two numbers in each subsequent line are the task holding the data and the value of the independent variable for this one dimensional array. The values of the dependent variable will be read from the file produced by the task holding them.

The ability to visualize the array as a whole is even more important for higher dimensional problems. In Figure 6 we can spot two errors. The small bumps near the right hand side were introduced intentionally. The discontinuity, which is clearly much larger, is an actual bug that was not noticed until this plot was made.

Higher dimensional arrays can also be visualized by tools more advanced than gnuplot. For example, a 3 dimensional array might be displayed as a movie showing how 2D slices change with time. Visualizing the data would bring the brain's power to detect visual patterns that would be undetectable by looking at lists of numbers.

Errors such as the ones illustrated here occur in real programs all the time. The code produces reasonable results, but they are wrong. One of us (AHK) encountered just such a situation 20 years ago when modeling pulsating stars.[6] A model was evolved for 50 hours of CPU time until it reached a limit cycle. A surface plot of velocity versus time and position showed that one of the layers in the interior stood out above the others. It turned out that a typographical error had been made in specifying the grid spacing. When the error was corrected, the amplitude of the pulsation increased significantly and a wave that had erroneously been absorbed reflected back to the surface. This error would have been nearly impossible to find without good data visualization. Had the visualization been integrated with the debugger, the error would have been found much sooner.

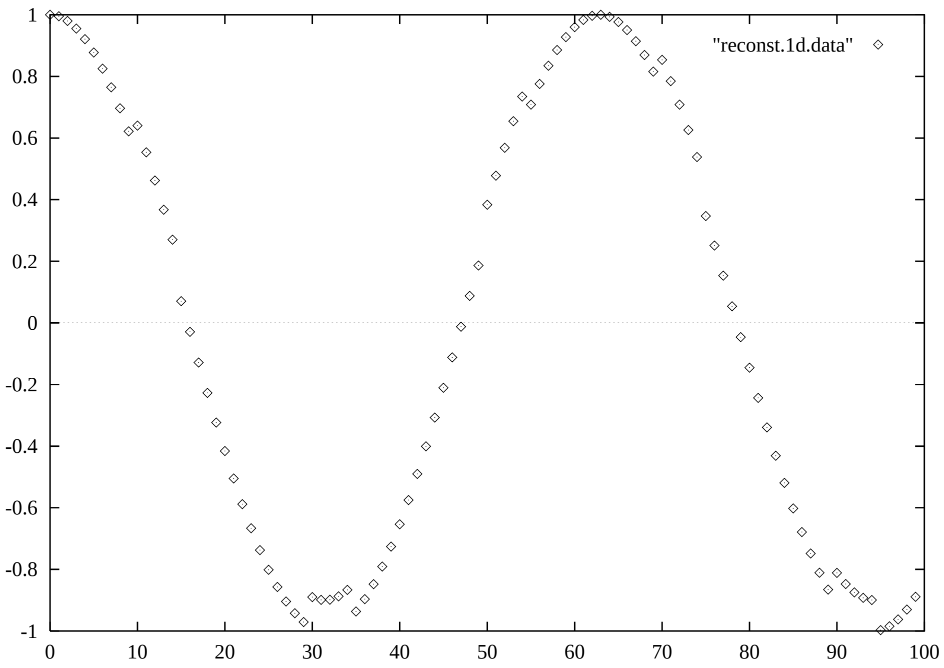

Figure 5: An array of length 100 distributed over 4 nodes with a block size of 5 has been dumped to 4 disk files. These files were combined to produce a single file with the data as it would have been dumped on a uniprocessor run. The error is easy to spot.

4 Conclusions

Data visualization should become a part of all interactive debuggers. So far it has not, but users have been able to work around this deficiency on uniprocessors. Our work makes it easier for users to visualize data on distributed memory parallel systems.

Our tool lacks some of the features we could have added had it been fully integrated with the debugger as is Prism[9]. File I/O is slow compared to having the visualizer and debugger work together on data stored in memory. We plan to provide an interface for tool developers that would more closely integrate the debugger and visualizer.

On the other hand, this weakness of our tool is one of its strengths. Since our interface is independent of any of the tools used, we did not have to write a debugger or a visualizer. Had we written our own, they would have been primitive compared to those currently available. With IVD, any debugger that can redirect its output to a file and any visualizer that can read data from a file can be used. This feature makes our tool completely portable and as flexible as most people will ever need. It also made the tool much simpler to develop.

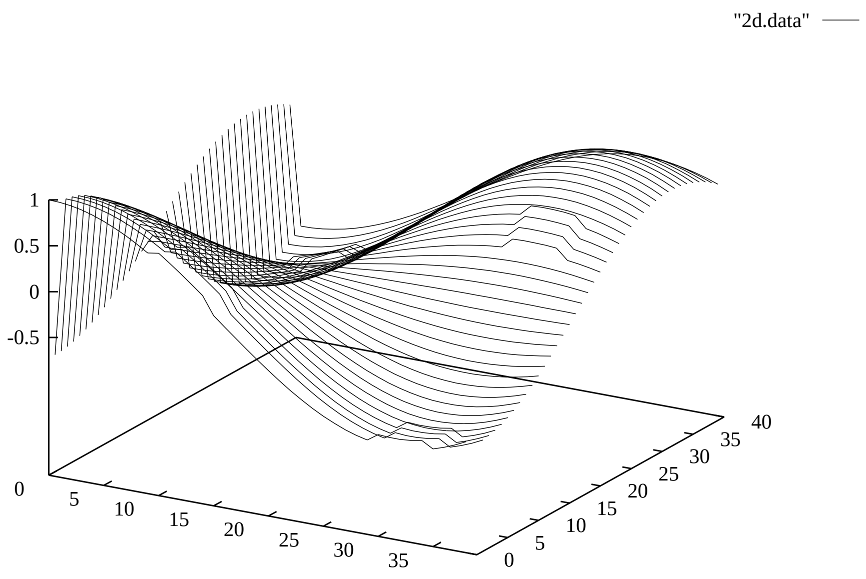

Figure 6: A two dimensional array with two errors. The small bumps near the right hand side were introduced intentionally to make a point. The discontinuity near the left hand side is an actual bug that was not noticed until this plot was made.

Acknowledgements

Thanks to Milon Mackey for explaining how PVM spawns tasks across multiple processors, and Rich Title of Thinking Machines, Bob Brown of Silicon Graphics, Danny Franklin of Parasoft, and Jeff McDonald of MasPar for help finding debuggers that incorporate data visualization. Milon Mackey and Vineet Singh suggested significant improvements to the paper.

5 References

[1] David Callahan and Ken Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[2] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, August 1993.

[3] J. Choi, J. Dongarra, R. Pozo, and D. Walker. LAPACK Working Note 55, ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. Technical Report CS-92-181, Computer Science Department, University of Tennessee, November 1992.

[4] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A Proposal for a User-Level, Message-Passing Interface in a Distributed Memory Environment. Technical Report CS-93-186, Computer Science Department, University of Tennessee, January 1993.

[5] Ming C. Hao, Vineet Singh, Alan Karp, Milon Mackey, and Jane Chien. IVD: An Integrated Performance Visualizing and Debugging Tool for Parallel Applications. In *ONR/ACM Workshop on Parallel Debugging*, San Diego, CA, May 1993.

[6] Alan H. Karp. Hydrodynamic Models of a Cepheid Atmosphere: I. Deep Envelope Models. *Astrophysical Journal*, 199:448, 1975.

[7] C. Keobel and P. Mehrota. An Overview of High Performance Fortran. *Fortran Forum*, 11(4):9–16, December 1992.

[8] Jonathan B. Rosenberg and Kent L. Beck. A Massively Parallel Programming Environment. Submitted to IEEE Computer.

[9] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title. Data Visualization and Performance Analysis in the Prism Programming Environment. *IFIP Transactions A*, A–11:37–52, April 1992.

[10] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[11] Craig Upson, Thomas Faulhaber, Jr., David Kamins, David Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics & Applications*, July 1989.

[12] Thomas Williams and Collin Kelley. *GNUPLOT Version 3.2*. NASA Ames Research Center.