

# Visualization of data for the debugging of concurrent systems

Robert F. Erbacher  
Georges G. Grinstein  
Institute for Visualization and Perception Research  
University of Massachusetts at Lowell  
Computer Science Department  
One University Avenue  
Lowell, MA 01854

## Abstract

*Debugging concurrent systems has been shown to be much more complex than for serial systems; this is further complicated by the number of processors that may be involved in any operation. The correctness of such systems are equally as important as serial systems. This places a considerable amount of extra demand on the debugging environment. Additional capability must be provided with the debugging environment to offset the complexity. We describe work related to the visualization of data associated with concurrent systems to aid users in comprehending the operation and correctness of their concurrent applications.*

**Keywords:** concurrent systems, debugging, visualization, parallelism, parallel debugging

## 1. Introduction

The use of concurrent systems is becoming increasingly widespread in all types of fields and occupations. We use the term concurrent system to refer to any type of environment allowing the execution of application code on multiple processors simultaneously, including, MIMD, SIMD, and distributed systems.

Developing efficient, error free applications for concurrent systems is much more complex, difficult, and time consuming than for serial systems. More focus has recently been given to the issues involved in developing applications for concurrent systems. Unfortunately, debugging tools are still weak.

It is important to understand the relationship between application comprehension, application debugging, and performance tuning as related to concurrent systems. Application comprehension deals with techniques geared towards aiding user understanding of how an algorithm works and what actions take place during different phases of execution. Application debugging takes the process a step further and provides capabilities to aid users in determining where in an application errors are occurring which effect incorrect results. Application debugging is very dependent on application comprehension; only user understanding of the program execution can provide effective debugging. Performance tuning deals with the analysis of program execution to determine what modifications may be made to improve execution throughput. As with application debugging, performance tuning is very dependent on application comprehension and can be considered a necessary step of application debugging. This results because applications are designed for concurrent systems in order to take advantage of their higher performance. If reasonable performance is not achieved, the program can in fact be considered incorrect.

Our research is concerned with the use of visualization to aid in the debugging of concurrent systems. The complexities of debugging concurrent systems as opposed to serial systems results from the possible multiple threads of execution in which different activities are occurring simultaneously. The user may be dealing with

hundreds or thousands of processors each with different data under consideration. Examining this quantity of information and its application execution characteristics can be difficult; but presenting it in a visual form, that is visualizing it, facilitates its interpretation.

The need for visual-based debugging in parallel systems is described by Stasko et al. <sup>1</sup> as follows:

“Parallel programming has an advantage over serial programming in that many computations can be performed simultaneously, thus reducing total computation time. Visual-based debugging systems offer this same advantage over traditional text-based ones: a larger quantity and variety of information (as opposed to a single stream of text) can be presented concurrently, which can be exploited by the human visual perception system.”

In this paper, we discuss the visualization and analysis of data within concurrent applications. By concentrating on the analysis of data, rather than on the processes or communication patterns, we provide the user with additional insights into the usage of the concurrent system and its potential problems.

We first provide a brief review of the key issues in the debugging of concurrent systems. We then describe an environment that combines analysis and visualization techniques, and provide example applications.

## 2. Previous Work

Extensive research has focused on techniques to improve program development technology for concurrent systems. Most of this research has been geared towards tuning the performance of concurrent programs (see <sup>2, 4, 5, 6, 7, 11, 12, 13, 14</sup>). Our research is primarily concerned with the debugging of parallel systems as it applies to errors in computation, rather than performance. Research in this area has relied on visualization techniques to aid users in understanding what is actually occurring as the program executes to help identify and explain the generation of erroneous data.

Most of the earlier environments were geared towards providing the user with statistical summary information on the execution of concurrent applications. This information typically included processor utilization, memory utilization, interprocessor communication, process scheduling, page faults, and instruction execution rate, and was provided in graphical form to make it more easily comprehensible. Example systems with these types of capabilities include PV<sup>10</sup>, RP3<sup>9</sup>, PIE<sup>11</sup>, ParaGraph<sup>5</sup>, and Pavane<sup>8</sup>.

For SIMD systems, the use of code views is often used to provide context under which the application is executing (i.e., PV<sup>10</sup> and PIE<sup>11</sup>). Code views also aid the user in more precisely identifying where in the application an error is occurring. Since a MIMD and distributed application may contain any number of different threads of program execution, this technique is difficult to use effectively. Faust<sup>4</sup> and Node Prism<sup>15</sup> extend this concept by displaying call graphs which provide a history of function calls leading to the current execution state.

Several systems provide techniques for simple analysis of data itself, rather than information strictly related to processor statistics. Prism<sup>14</sup> and IVE<sup>3</sup> allow the data values associated with a selected variable to be displayed graphically using a color scale to represent data value ranges or intensities. Prism also provides for thresholding and other simple interactive techniques to allow the user to focus on specific data of interest.

While not a visually oriented environment, the Poker<sup>16</sup> environment supports the display and modification of values associated with concurrent variables. Since there can be thousands of instantiations of each variable,

this capability can be extremely difficult to implement and use. Consequently, it has seen little use in more recent systems.

Maritxu <sup>17</sup> differs in that it focuses on the processor rather than the process and harnesses the human perceptual system to help in identifying unique characteristics in large animated displays. By displaying large amounts of information simultaneously in a single display, the user can very quickly identify trends or anomalies in the data collected from the processors. This is done through the use of “icons”, which are groups of pixels in association used to represent multiple characteristics or parameters of a data object under consideration (processors in this case). This technique provides increased information at the cost of screen resolution.

Finally dynamic presentations can be quite effective at revealing trends and behavioral phenomena that may not be revealed at all in a static display <sup>9,10</sup>. Most displays to-date have provided static presentations.

### **3. Visualization of Data**

We provide several techniques for the visualization of data. The goal of these techniques is to aid the user in comprehending what is occurring during the execution of an application. Most of today’s systems have been developed with a focus on the analysis of concurrent applications to aid in performance tuning. Our work is geared toward debugging. This involves determining how, and potentially why, generated results do not match the expected results. Since incorrect data values are being generated, we emphasize data visualization rather than processor visualization.

The user is presented with a screen, in a grid-like format, consisting of information, such as the status of the processors, their utilization, and their message characteristics. This represents a visualization of data associated with specific processors in a concurrent system, i.e., a processor centered approach. An alternative approach is to focus on the data independently of the processor. For example, when the user select elements for further investigation, the data elements get selected, not the processors with which they happen to be currently associated.

#### **3.1. Data element encoding**

We provide several visual techniques for encoding the data elements. The most familiar technique is a grayscale encoding. This is similar to grayscale images often used to represent images. We map the data values under consideration onto a range of 0..255 grayscale values. In this way, the value of a data element controls the intensity of a pixel or icon on the display screen. While a grayscale encoding may initially appear to provide sufficient capability in this type of environment, there are cases where this is not true. An example in which grayscale encoding actually fails when presenting even simple data is provided in section 5.1.

A second encoding we provide is a pseudo-color scheme where a colormap file defines the color values to be used for different ranges of data values. This helps highlight various regions of data values and provides more identifiable changes between value ranges than can be done with grayscale. In this technique, the data values under consideration are mapped onto a range of 0..N, where N is the number of color elements defined in the colormap file. This technique also provides more control over the number of color entries, rather than being limited to 255 as with grayscale encoding.

To present a very wide distribution of values, we also provide a truecolor mapping. This technique allows significantly more ranges of data values to be represented independently. In this technique, the data values under consideration are normalized separately and sent unmapped to the frame buffer as is.

The classic presentation grid layout need not be mapped to processor layout. In this direction, we provide a mapping in which the data element ID (often the processor ID with which the element is currently associated) is used to identify placement on the Y axis and the data value itself identifies placement on the X axis. Both the data element ID and data element value are scaled to a usable range for this display (i.e., 0..255). This provides a better mechanism for observing distributions of values and is similar to histogram visualizations and scatterplots.

### **3.2. Data selection**

We provide several techniques for selecting data values. Our initial approach was to merely color the entire region with a single user defined color. This did not allow for the underlying selected values to be discerned. Consequently, we began selecting data sets by adding a fixed value to their color to add a “highlight”. We are in effect using a separate color scale for the selected region than we are using for the unselected portion of the image. Applying this technique to the standard grayscale encoding discussed previously generates a grayscale image with the selected region appearing as a green color scale. This allows for immediate identification of the selected region but does not prevent the underlying data values from being discerned.

Internally, the actual data values are not modified. Instead, a tag is provided with each data element. By checking this tag, data elements can be identified as selected and how they should be highlighted. Multiple regions can also be selected using different schemes. This allows elements from each selection to be differentiated from each other as well as from unselected regions.

## **4. Visualization of Data Operations**

To date, the visualization of operations on concurrent systems has been limited to the visualization of locks, waits, etc. While this does provide useful information to aid in the performance tuning of concurrent systems, additional information is needed to aid in debugging. Our research currently aids in determining where an error is occurring by using visualization techniques that examine the operations being applied to data. The goal is to provide a system that will not only identify where in an application incorrect data is generated but what operations lead to the incorrect generation of results, hence beginning to answer the question ‘why’.

Accomplishing this portion of the debugging process requires that source code be examined, either as a separate process or through tools provided by the visualization environment. This forces the user to perform a context switch from the visual display the user has been examining to the application code that is textual in nature and requires that the user begin performing a different type of mental processing. Our goal is to provide a unified environment in which a visual display of the operations being performed replaces the analysis of the application code. In this way, the user will at most need to switch from one visual technique to another, thereby reducing the mental load typically required.

## **5. Examples**

In this section, we will present examples of the visual techniques we have incorporated, as well as two examples of applications in which our visual techniques provided a significant difference in the understanding of the applications in question. The first example discusses issues in visualizing an even-odd exchange sort. The second example discusses how a matrix transposition algorithm presented a hardware error.

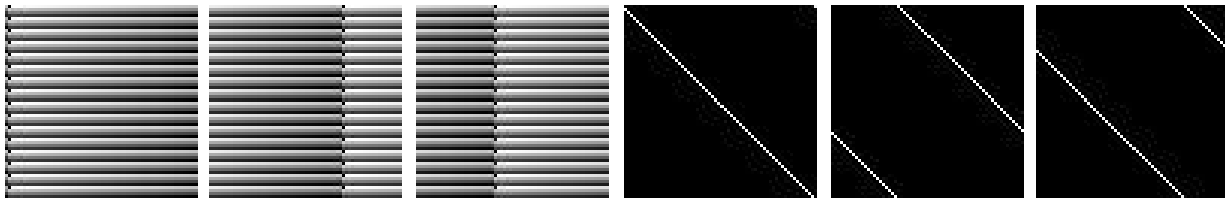
We have implemented several demonstration programs, including: matrix multiplication, matrix convolution, even-odd exchange sort, and matrix transposition. These examples were implemented on a proprietary 4K

processor SIMD architecture called Terasys. Terasys is based on the PIM (Processor In Memory) chip. Each processor contains 2K bits of local memory on which it may operate.

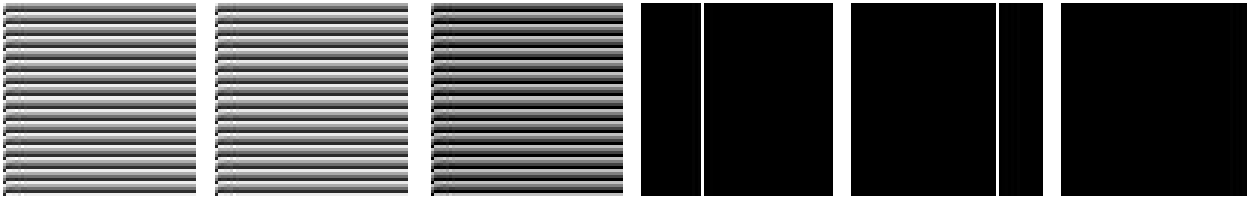
### 5.1. Matrix multiplication

The first example is of a matrix multiplication algorithm using Cannon’s method. The algorithm works by repeatedly rotating the first input matrix to the left (west) and the second input matrix up (north), see Figure 1. Next, the temporary matrices, seen in Figure 2, are generated. The first three temporary matrices are generated from the first input matrix by taking the first column and repeating it across the entire matrix. The second three temporary matrices are generated from the second input matrix by taking the first row and repeating it down the entire matrix. The values in each of the temporary matrices are then multiplied together (not a matrix multiplication) to generate an intermediate value, which is then added to the result matrix. The results matrix is shown in three stages in figure 3.

In the first example of this technique we display grayscale images that are designed to help show the execution of the algorithm. The first input matrix consists of sequential numbers in the range of 0..255 in increasing order, which is repeated several times to fill the processor array. The second input matrix consists of the identity matrix. Even these few static images provide insight into the execution of the algorithm. These images make it possible to envision the operation of the algorithm and how the different variables are passed between processors during each phase. The actual execution of this demonstration generates a smoothly updating dynamic display that provides even greater feedback as to how the execution is performing.



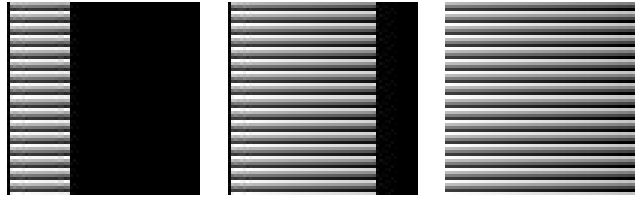
**Figure 1.** These images show the two original matrices being multiplied. Three examples images are provided of each dataset. The examples are taken at different points in time, showing the movement of the data values through the interconnection network. The first three images are examples of the first input matrix. The second three images are examples of the second input matrix.



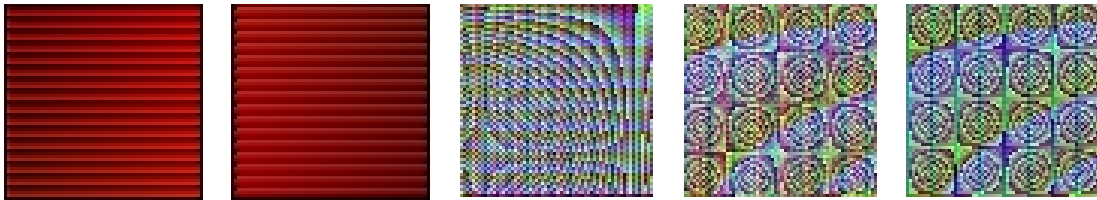
**Figure 2.** These two sets of images show the temporary matrices being used to assist in the calculation of the multiplication. Three points in time are represented. The two images are multiplied together on a processor by processor basis to get a new column of the result. The first three temporary images are generated from the first input matrix. The second three are generated from the second input matrix.

A second example of the matrix multiplication is provided in Figures 4 and 5. This shows a true color example in which the first input matrix consists of sequential numbers in the range of 0..4096 in increasing order (e.g., each processor has a unique value, unlike the first example). The second input matrix contains the same data except the values are in decreasing order. The matrix multiplication is performed in the same fashion as in the previous example of matrix multiplication. The patterns present in the result image are an artifact of the

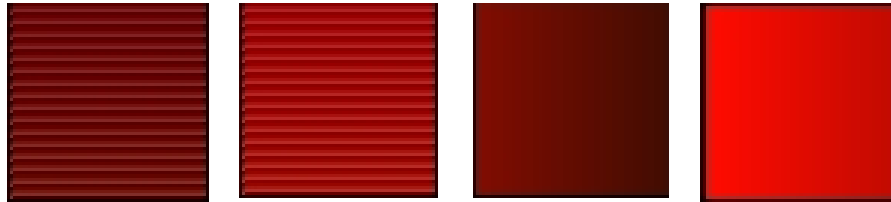
input matrices we chose as well as the effects of using a true color display. Observing these patterns as they develop clearly shows that visual displays can present information in a fashion that cannot be done with textual based displays. An error occurring in these result images would be clearly observable as an inconsistency in the display that does not fit the principal pattern.



**Figure 3.** The result of the matrix multiplication is seen in the process of being generated.



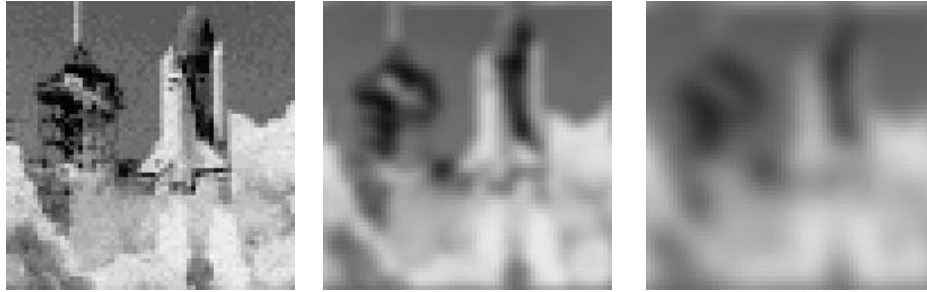
**Figure 4.** The two input matrices are shown on the left. Three examples of the result matrix over time are shown on the right. These images were originally truecolor.



**Figure 5.** The first two images show examples of the temporary matrices associated with the first input matrix. The second two images provide examples of the temporary matrices associated with the second input matrix. The image pairs were taken at different times.

## 5.2. Matrix convolution

Another demonstration we have implemented consists of a standard  $3 \times 3$  image convolution low-pass filter. Examination of these three static images shows the effects of the convolution over time but does not show effectively how the convolution modifies the image at different stages. In our dynamic implementation of this algorithm it can be seen how the convolution smooths the image characteristics and dims brighter portions. In effect, the dynamic implementation provides a greater understanding of the effects of the convolution, which should allow users to use this type of operation more effectively. Providing a dynamic implementation to users when they are attempting to use such a tool will allow them to more accurately control the kernel variables associated with the convolution and generate better results. The effectiveness of these results has led us to begin developing an environment for the interactive computational steering of concurrent applications. Figure 6 shows examples of the matrix convolution algorithm.

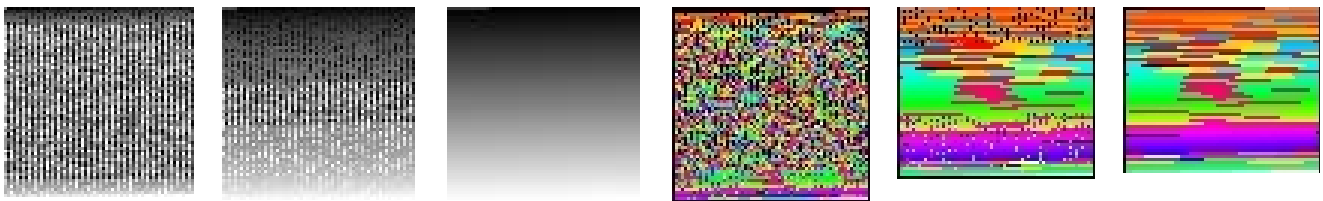


**Figure 6.** Example of a matrix convolution. The first image shows an unmodified example. The second image has been convoluted twice. The third image has been convoluted ten times. Images are scaled to three times normal size.

### 5.3. Even-odd exchange sort

Figure 7 shows two examples of an even-odd exchange sort. Our initial implementation of the even-odd exchange sort represented the elements being sorted as grayscale intensities with the value of the data element controlling the intensity (Figure 7¼first three images). The elements were organized in a square grid with the upper left corner of the grid containing the data element held by processor ID #0. The lower right corner contains the data element held by processor ID #MAX\_ID. This display was updated after every exchange in the algorithm and clearly shows the process by which darker (lower valued) data elements are moved to the upper left while brighter (higher valued) data elements are moved to the lower right. Due to the simplistic nature of this display we were able to update the display at near real-time.

The second example was created using a random pseudo color map applied to the data values (Figure 7¼second three images). Each example consists of three images taken at different points during the execution of the sort. These static images provide insight into the algorithm’s progress that is not identifiable without a visual representation. In particular, the first example shows that the high and low ends of the processor values begin to become more ordered much sooner than the middle range does. Meanwhile, the second example shows that it is actually the middle range that eventually completely sorts first. The dynamic implementation of this example shows these characteristics more clearly than these static examples. Again, these examples provide insight into characteristics of the algorithm that go beyond what could be learned from examination of the source code or textual data. We are identifying patterns in the execution through patterns in the visual display.



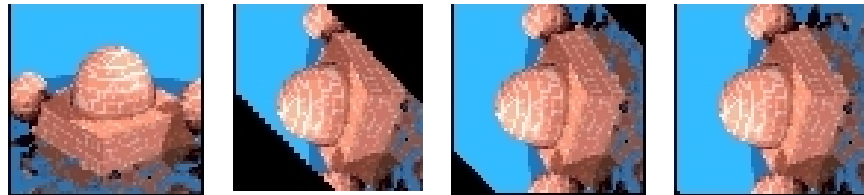
**Figure 7.** Examples of an even-odd exchange sort. The first three figures present a basic greyscale representation. The second three images show random pseudo color maps applied to the image. They are represented here as greyscale; the difference from the pure greyscale images can still be discerned.

### 5.4. Matrix transposition

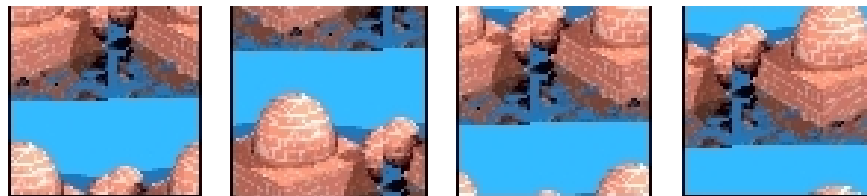
This final example shows the execution of a matrix transposition algorithm. Figure 8 shows the initial image before transposition and three examples of the result image as it is being generated. Figure 9 shows two

examples each of two temporary working matrices (an upper working matrix and a lower working matrix). The upper working matrix is rotated northeast and the lower working matrix is rotated southwest during every iteration. A diagonal mask is associated with each working matrix. The mask associated with the upper working matrix is shifted north every iteration while the mask associated with the lower working matrix is shifted south. As the mask values are shifted they do not rotate to the other end of the processor array as the working matrices do. Processors with positive mask values during any iteration are added to the result matrix. This gives the impression of the matrix growing from the center diagonal to the northeast and southwest corners.

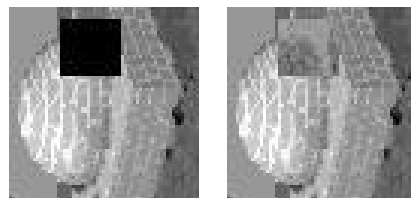
The example images shown in figures 8 and 9 clearly show the growing effect of the result image and the rotation that is being applied to the working matrices. Figure 10 shows the two selection methods discussed in section 3.2. The first image shows a region of data elements selected with a solid colored sub-region. The second image shows a region selected using a different color scale, providing a highlight of those processors%allowing the data values in the selected region to be determined.



**Figure 8.** Example of a matrix transposition. The first image is the input image. The remaining three images show stages in the creation of the result image.



**Figure 9.** Example of a matrix transposition. The first two images show two examples of the upper working matrix, taken at different times. The second two images show examples of the lower working matrix, taken at different times.



**Figure 10.** Examples showing a selected region of the image. The first image shows a solidly selected region. The second image contains a highlighted region.

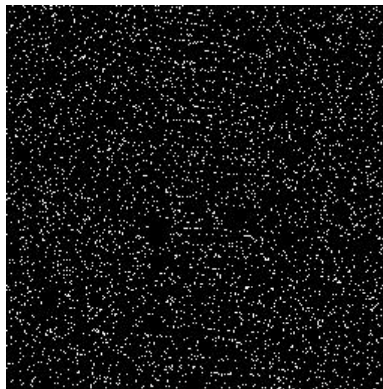
## 5.5. Successful visualizations

This section discusses two examples in which our visualization techniques provided exceptional feedback that greatly impacted our understanding of applications being examined. In effect, these examples show that visual

perception of data can provide beneficial information not available through other techniques and that the results are identifiable.

### 5.5.1. *Even-Odd exchange sort*

This example discusses additional issues of the even-odd exchange sort, as discussed in section 5.3 (Figure 7). Upon examination of the animation generated from this application, it appeared that the display contained many more brighter data elements moving to the lower right than it contained darker elements moving to the upper left. Our first instinct was that the random numbers being generated to fill the unsorted array were being generated incorrectly or with preference to higher values. This turned out not to be the case. Instead, after switching to a pseudo-color representation (Figure 7's second three images), as opposed to the grayscale representation, we observed that the distribution appeared normal—equal numbers of elements moving in both directions. This appears to be the effect of the human eye being attracted to brighter values more than to darker values. Since the display is being updated at a fairly high frame rate, the eye is picking out the brighter elements but not the darker elements, giving the impression of a skewed distribution. To verify these results we implemented a scatter plot version of the exchange sort (Figure 11), which clearly shows an even distribution of values. These effects are observable in the static grayscale images presented in this paper.



**Figure 11.** An example of a scatter plot visualization of an even-odd exchange sort.

### 5.5.2. *Matrix transposition*

A second example we implemented was of a matrix transposition. This was discussed in section 5.4 (Figures 8 and 9). This algorithm was implemented and tested on one machine but was later moved to another machine for demonstration purposes. When moved to the new machine, an artifact was observed in which a black blotch appeared in the initial diagonal and grew to cover a fairly large area. This was caused by a break in the parallel prefix network, which is used for communication between processors. This break resulted in all data elements being sent through the break resulting in zero values. Since this error was then propagated through other communications, it covered a fairly large area upon completion and was very easily observed.

## 6. Conclusions and Future Work

We have discussed concurrent systems and what visualization tools are currently available. We have identified key problems needing to be addressed and provided examples of tools that concentrate on the data aspects of the computation, an as of yet unexplored area. Using this approach, analysis of data quickly shows characteristics that may not be as easily discerned with other techniques. We showed that initial impressions generated by visual techniques should not be taken for granted. Visual techniques that rely much more

heavily on early vision perception can be a hazard as these can sometimes generate displays that the human perceptual system interprets incorrectly (as was seen with the even-odd exchange sort). Issues related to how the human perceptual system works must always be taken into account when developing visual techniques and considered when viewing the results of such techniques. Lastly, we showed that even simple techniques incorporating the analysis of data can be beneficial in identifying characteristics of the computation.

More work needs to be done concerning the analysis of data and data operations within a concurrent system. Interactive techniques must be developed that provide more detail of the underlying computation. We are extending our environment to incorporate interactive computational steering techniques to make every feature of the computation dynamically controllable by the user.

## References

1. William Appelbe, John Stasko, and Eileen Kraemer, "Applying Program Visualization Techniques to Aid Parallel and Distributed Program Development," Georgia Institute of Technology, College of Computing, Technical Report GIT-GVU-91-08, 1993.
2. C. Bateau, Y. Maheo, and J.-L. Pazat, "Parallel Program Performance Debugging with the Pandore II Environment," *Parallel Computing: Trends and Applications; Proceedings of the International Conference PaeCo93*, Elsevier Science Publishers B.V., 1994, pp. 241-248.
3. Mark Friedell, Sandeep Kochhar, Mark LaPolla, and Joe Marks, "Integrated Software, Process, Algorithm and Application Visualization," *Journal of Visualization and Computer Animation*, 1992.
4. V. A. Guarna, D. Gannon, D. Jablonowski, A. D. Malony, and Y. Guar, "FAUST: An Integrated Environment for Parallel Programming," *IEEE Software*, Vol. 6, No. 4, 1989, pp. 20-27.
5. M. T. Heath and J. A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, Vol. 8, No. 5, 1991, pp. 29-39.
6. Michael T. Heath, Allen D. Malony, and Diane T. Rover, "The Visual Display of Parallel Performance Data," *IEEE Computer*, Vol. 28, No. 11, November 1995, pp. 21-28.
7. Michael T. Heath, Allen D. Malony, and Diane T. Rover, "Parallel Performance Visualization: From Practice to Theory," *IEEE Parallel & Distributed Technology: Systems & Applications*, Vol. 3, No. 4, Winter 1995, pp. 44-60.
8. Takeshi Horie and Morio Ikesaka, "AP1000 Software Environment for Parallel Programming," *Fujitsu Scientific and Technical Journal*, Vol. 29, No. 1, 1993, pp. 25-31.
9. D. N. Kimelman and T. A. Ngo, "The RP3 Program Visualization Environment," *IBM Journal of Research and Development*, Vol. 35, No. 5-6, 1991, pp. 635-651.
10. Doug Kimelman, Bryan Rosenburg, and Tova Roth, "Strata-Various: Multi-Layer Visualization of Dynamics in Software System Behavior," in *Proceedings of Visualization '94*, IEEE Press, 1994, pp. 172-178.
11. T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung, and C. E. Fineman, "Visualizing Performance Debugging," *IEEE Computer*, Vol. 22, No. 10, 1989, pp. 38-51.
12. Cherri M. Pancake, "Graphical Support for Parallel Debugging," *Software for Parallel Computation*, Springer-Verlag, 1992, pp. 216-228.
13. Bernhard Ries, R Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards, and W. Smith, "The Paragon Performance Monitoring Environment," *Parallel Computation; Second International ACPC Conference*, Springer-Verlag, 1993, pp. 233-248.
14. Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title, "Data Visualization and Performance Analysis in the Prism Programming Environment," *Programming Environments for Parallel Computing; Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, Elsevier Science Publishers B.V., 1992, pp. 37-52.

15. Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title, "A Scalable Debugger for Massively Parallel Message-Passing Programs," *IEEE Journal of Parallel & Distributed Technology*, Summer 1994, pp. 50-56.
16. L. Snyder, "Parallel Programming and the POKER Programming Environment," *IEEE Computer*, Vol. 17, No. 7, 1984, pp. 27-37.
17. Eugenio Zabala and Richard Taylor, "Process and Processor Interaction: Architecture Independent Visualisation Schema," *Environments and Tools for Parallel Scientific Computing*, Elsevier Science Publishers B.V., 1993, pp. 55-71.