Implementing an Interactive Visualization System on a SIMD Architecture

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

Abstract

The use of a massively parallel architecture to provide real-time performance of an application often requires the development of a version of that application tailored specifically to the given architecture [HIL93]. We found this to be the case during the implementation of a state-of-the-art visualization environment for a proprietary SIMD architecture. This paper discusses methods found to be useful for improving performance on the SIMD architecture and describes different strategies for the allocation of processors and processor memory. The processor allocation strategy found to be most useful is shown to correspond with the number of processors available. We also found that the requirement of displaying each rendered frame quickly, such that the display is updated in real-time, greatly affected our processor allocation strategy and resulted in a non-intuitive mapping function.

1 Introduction

Generating and rendering nontrivial display images can be a very computationally intensive task. For this reason, parallel architectures have been found to be extremely useful in aiding image generation. When the desired end result is a real-time interactive display, the demand for computational power is even greater. Unfortunately, merely having the appropriate hardware available does not guarantee the desired computational rate. This is a consequence of the complexity of such parallel systems. Currently, programming such systems requires that algorithms be redesigned for each architecture in order to realize the maximum performance provided be each system [HIL93].

A primary issue in developing an algorithm for a parallel architecture is how best to map data onto the individual processors. The allocation strategy used is one of the most significant factors affecting the communication pattern and the overall performance of the algorithm. This paper discusses several issues in designing an algorithm for a proprietary SIMD architecture, and shows how the need to display the resulting image greatly affected the data mapping strategy used. Other issues found to be important and useful for providing high performance on this architecture are also described. The algorithm of interest is the "Color Icon" developed by Haim Levkowitz [LEV91a][LEV93][LEV88].

2 Iconographics

The scientific data visualization environment designed at the University of Massachusetts-Lowell incorporates several state-of-the-art techniques for the rendering of scientific data. These techniques are based on the concept of "iconographic" displays. An iconographic display associates an icon¾a figure of arbitrary size¾with each pixel of the original image. The various visual attributes of an icon

are under data control. The exploratory visualization environment which incorporates these iconographic techniques is termed “Exvis” (Exploratory Visualization).

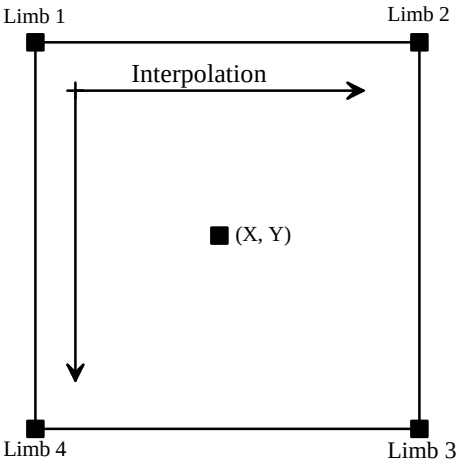

Fig. 1: Organization and characteristics of a Color Icon.

Because most data sets consist of more than a single image, we have developed icons that allow us to map multiple images (parameters) onto a single icon. This mapping ability allows us to merge the images into a single display, which hopefully provides more information than is observable from viewing the individual images side by side. An important concept here is that icons provide more information at the cost of screen resolution; for each pixel in an original image, an iconographic display has an area of pixels to contain each icon.

2.1 The Stick-Figure Icon

The first icon, developed by Pickett [PIC88], is termed the “stick figure” icon. A stick figure icon is formed by connecting line segments at their endpoints according to a set of geometric rules. The angles, lengths, and grayscale values of the line segments can be controlled by data. Two of the data variables, not necessarily independent of those controlling the icon features but hopefully largely independent of each other, control the position of each icon on the display surface. When icons are used to represent imagery data, icon position simply corresponds to pixel position. With sufficient density, the icons form a surface texture display, and structures in the data are revealed as streaks, gradients, or islands of contrasting texture.

2.2 The Color Icon

The stick figure icon relies primarily on the perception of line orientation to reveal information. Another approach is to use color perception. Color is a very effective way to merge multiple parameters of imagery data. For up to three parameters, the values of the pixels in the separate images control the three color channels of the corresponding pixel in the integrated display. Different mappings of values to color channels and different color models provide a wide range of options. Levkowitz [LEV91a][LEV93][LEV88] developed a generalized color icon to integrate multiple images using both color and geometric features. In this icon, a rectangle of pixels is used to represent each point in a data set. The color channels of the four corners of the icon may be independently data controlled¾providing 12 parameters (figure 1). For historical reasons the corners of the icon are termed limbs. Thus, we are defining a mapping between the data parameters and the icon limbs. The remaining pixels within the icon are interpolated to obtain the final coloring of the icon as a whole. The color icon provides straight color pixel integration when the width and height of the icon are both set to one pixel.

Fig. 2: Original images used to implement the iconographic technique for sensor fusion.

Another important issue related to the use of color icons is the color space in which the representations are taking place; each color space is represented by a color model. The simplest color model is the RGB color model, in which each color is represented by a red, green, and blue triplet of values. In addition to RGB, the color icon environment provides several other color models, including GLHS, MUNSELL, and CIE. GLHS is a Generalized Lightness, HUE, and Saturation color model developed by Haim Levkowitz [LEV91b][LEV93]. Interactive controls, provided with the GLHS model, allow the user to roam through a space of color models to find a model that provides the most effective presentation of the data.

An example of the color icon is shown in figure 3. In this example, two grayscale images of a mill building at Logan Airport, shown in figure 2, are merged into a single display. The left grayscale image is in the infrared spectrum while the right image is in the visible spectrum. The goal in merging these two images was to bring out features of both in a single display while still maintaining the clarity of the original images. In other words, we wish to bring out the details and contrast from

Fig. 3: Image showing the two grayscale images of the mill building merged using the Color Icon.

the visible image and the heat signatures from the infrared image. The resulting image clearly shows the details of the visible image, such as the windows of the building, as well as which smokestacks are active.

2.3 Performance Requirements

Because iconographic techniques tend to be computationally expensive, several seconds may be required to render a single display using these techniques on a serial architecture. This rendering time, combined with the time required to find useful settings for the dozens of variables that control the rendering transformation, makes the process of exploring the possible representations of a database extremely time-consuming. Moreover, because many recent databases are very large and have hundreds of parameters, the task of fully exploring such databases appears overwhelming. Consequently, an interest in supercomputer implementations of these techniques has developed. It is hoped that these implementations will speed up the rendering time and substantially reduce the time required to fully explore a database. In [SMI91], Smith and Grinstein describe a supercomputer implementation of the stick figure icon. In this paper, we will discuss implementation details of the supercomputer version of the color icon and briefly outline the benefits and performance of this system.

Under a grant from the Supercomputing Research Center (SRC) of the Institute for Defense Analysis, we implemented a SIMD parallel version of the color icon on the SRC's proprietary architecture. One goal of this project was to provide a faster renderer to solve the problems described above. In addition, we hoped to create an environment which would allow the user to manipulate controls in near real-time, such that the combinations of controls and parameters could be explored much more quickly. The desire was to produce a system that would allow users to gain greater insight into how the parameters and controls affect the display, thus allowing them to make more intelligent and efficient use of the controls and parameter mappings.

3 Applicability to a Supercomputer Implementation

The serial implementation of the color icon generates a complete picture by rendering icons one at a time for all points in a data set. Because each icon in a rendered display can be calculated completely independently, there is the potential for extensive parallelization of the rendering routine; however, parallelization by itself does not solve all the problems of fast rendering. There remains, for example, the problem of determining how to parallelize the routines or allocate the processors. Several strategies for processor allocation will be discussed below.

4 The Terasys System

The hardware provided to us for this project is the "Terasys" system (figure 4), developed by the Supercomputing Research Center [MAR93][SWE92]. This system provides a bit-serial SIMD architecture in a linear array organization. The system we were provided with contains 4K processors, although up to 32K processors are supported. Each processor operates on a local 2K-bit memory. This memory limitation is minimized by the ability to specify the bit lengths of all parallel variables. The Terasys also provides a parallel prefix network, which improves the performance of some important operations. A SUN Sparc II plays hosts to the Terasys hardware, which resides on the S-Bus (the primary SUN I/O bus). The performance of the S-Bus was found to be a limiting factor for the real-time graphical displays [ERB95].

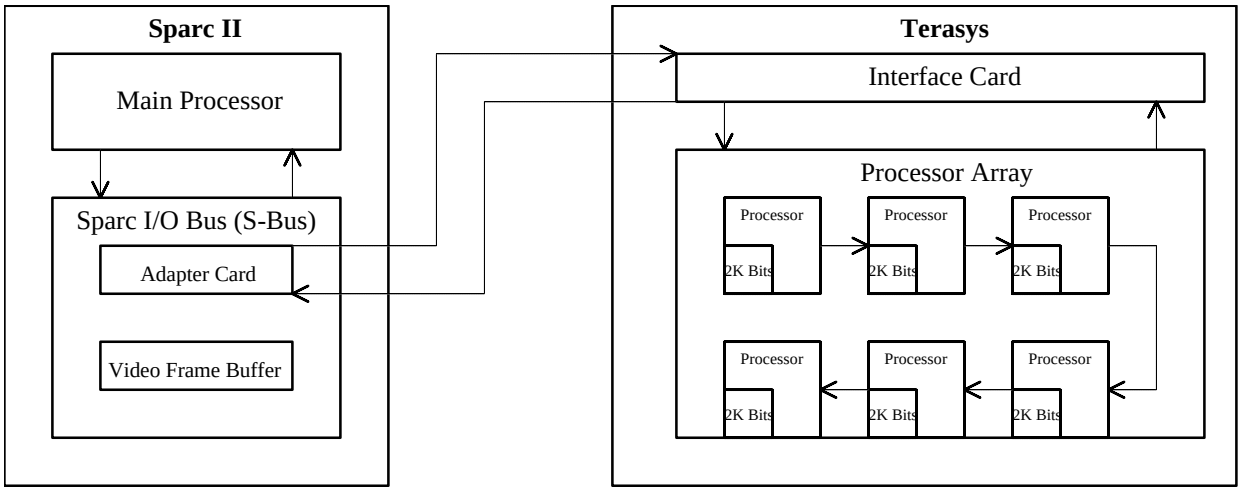

Fig. 4: Organization of the Terasys Hardware.

5 Implementation Issues

During the generation of the parallel version of the color icon portion of the Exvis environment, many implementation details beyond what must be handled in a serial implementation had to be dealt with. This section will describe some of the major issues that guided the design of the parallel environment.

5.1 Generality of Routines

The serial implementation was organized such that routines were very general and reusable. The goal was to make it easy for users to adapt these routines and generate their own icons with minimal difficulty. Unfortunately, these characteristics do not coincide with the requirements of a parallel implementation. First, it is beyond the capabilities of current compiler technology to adequately parallelize serial code for an arbitrary parallel system. It is therefore necessary to generate an algorithm for each specific type of parallel hardware. In addition, the purpose of implementing a parallel version is generally to provide a high performance environment; an interactive real-time environment in our case. These two considerations dictated the elimination of some of the generality which allowed re-use of routines in order to provide the most efficient implementation possible.

5.2 An Interface for Dynamic Interaction

Because the original user interface had been designed for a static display, generated by a slow serial implementation of our visualization environment, there was no provision for dynamic display updates. The user interface therefore had to be modified to handle dynamic updates. It was also necessary to provide a mechanism by which parallel data that had become outdated was updated before generation of the next display update.

The modifications to the user interface to handle dynamic updates were fairly simple. The system already provided slidebars for most of its interaction; we merely needed to modify the way the slidebars were used. Slidebars are effective for this purpose because they allow the user to rapidly and smoothly scan across a range of values. This, consequently, allows the user to use the full real-time interactive capabilities of the system.

Because loading data onto the Terasys hardware is slow, it was essential to minimize the amount of information to be reloaded when the display is calculated. This was accomplished by providing flags

which are set by individual interaction routines to specify what data had become invalid. An alternative to having the routines set flags would be to have the routines reload the data directly; however, this approach would have added unnecessary overhead by loading data more than once if a particular routine was called multiple times before an updated display was required. This in turn would have made the interface slower (due to pauses to reload data) and consequently more difficult to use interactively.

5.3 Floating Point Computation

The first major issue that had to be dealt with was the performance of floating point computation. This type of computation appeared to be performed so poorly on the Terasys hardware that future versions of the compiler will not support this type of computation on the Terasys itself (e.g. the data will be sent to the host processor for processing). Floating-point performance is very likely a problem with all bit-serial processors as they do not have specialized components devoted to floating point computation. The CM-2 Connection Machine solved this problem by providing a shared floating point unit for every group of 32 bit-serial processors. This arrangement provides better performance than forcing the bit-serial processors to perform the computation; however, it still degrades performance overall sine the need to share a system resource among multiple processors produces a bottleneck. The scalability of the system is thus decreased. While much research has been done on the Connection Machine and authors discuss the use of the floating point units [MIS94][SCH94], the fact that the scalability, and consequently the performance, of the system are decreased when floating point is used and methods of dealing with this situation are not discussed.

Another alternative is to modify the algorithm such that it uses only integer arithmetic. This modification would have been relatively straightforward for the RGB color model but a major undertaking for many other algorithms. Generating such an algorithm for the GLHS color model [LEV91b][LEV93] in particular would have been difficult, and beyond the scope of the grant. As an alternative, we noted that the result of the GLHS algorithm, as with most image processing algorithms, is a sequence of three color channels (red, green, and blue), each with a value in the range of 0..255. Accurate results thus require only sufficient precision to provide values in this range. Consequently, not all the accuracy normally provided by a full floating point implementation may be needed by a given algorithm. This is the case with our rendering routine. Our alternative was therefore to implement a form of fixed-point computation.

We simulated fixed-point computation with signed integers, using macros to perform the required arithmetic operations. This method is advantageous in that it provides greater performance with little modification to the algorithm itself. Architectures which allow the bit lengths of the variables to be controlled, as with the Terasys, may gain added speedups if the bit lengths of variables are specified to be the minimum required for the accuracy needed by the algorithm. This allows operations to spend as few cycles as necessary in processing a given variable. Unfortunately, fixed-point computation is much less standardized than floating point and is unlikely to appear in compilers. Thus, this modification must be hand coded. In addition, this method will work only for algorithms in which the accuracy required can be predicted beforehand and limited to a range that makes fixed-point feasible without affecting the results. After making the modifications to our system to incorporate fixed point computation, rather than floating point, the throughput of the system was increased substantially.

5.4 Processor Allocation

The actual implementation of the rendering algorithm allocates processors in a slightly unusual manner. This is a result of the need to read the result data from the processor array for display. The most basic allocation strategy would have each processor generate one complete icon. In this

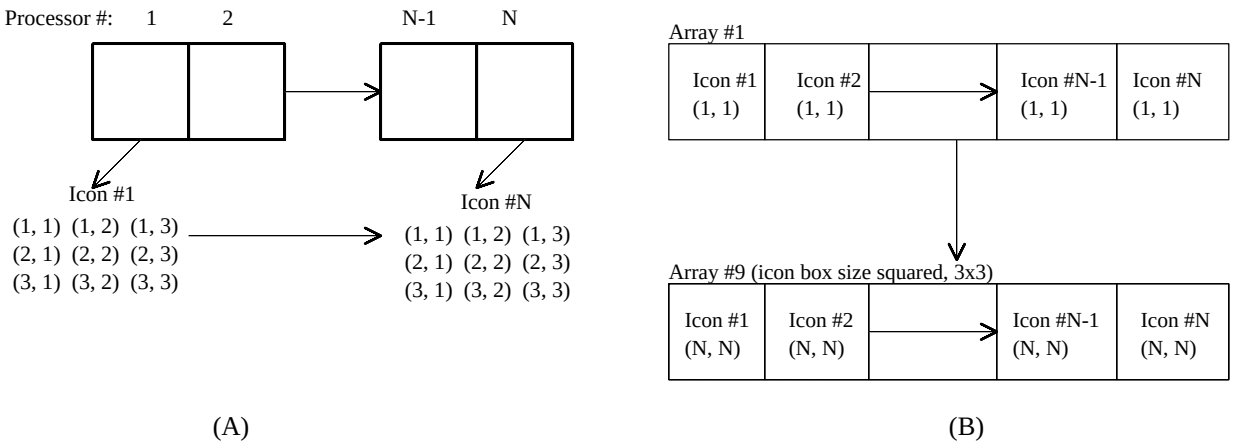

Fig. 5: Simple data layout for a 3×3 icon. One icon per processor.

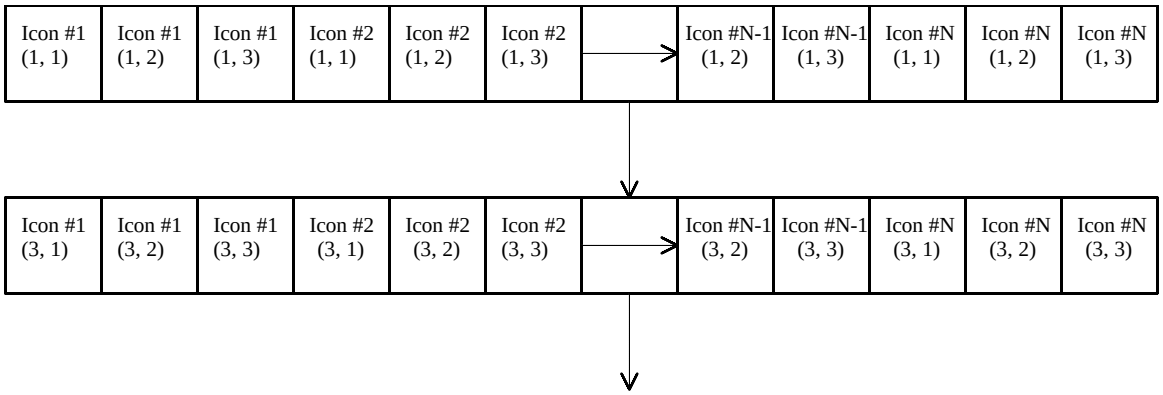

Fig. 6: Result data after reorganization for display. Example for a 3×3 icon. N is the width of the result image.

Icon Box Size	Computational costs for a single rendered frame using the RGB color model. This includes display time. No data reorganization	Computational costs for reorganizing data for display of a single frame, using the RGB color model.
1×1	"0.016666"	"0.000000"
2×2	"0.041665"	"0.016666"
3×3	"0.083330"	"0.049998"
4×4	"0.141634"	"0.066664"
5×5	"0.216658"	"0.133328"
6×6	"0.299988"	"0.233324"

Fig. 7: Costs associated with reorganizing data for display in a simple implementation.

approach, all the data values and computation associated with an icon are dedicated to a single processor. This strategy eliminates all inter-processor communication and should therefore generate a relatively efficient algorithm. The problem is that each processor will contain a whole icon (figure 5-A), which can have a size up to 5´5 pixels. Thus, we have 25 pixels in the same processor, of which each row of 5 must be displayed in adjacent positions on the screen. With this arrangement the general library routine which reads all the values from a particular array in all the processors will not place the data in an appropriate form for display; the data will be in the form shown in figure 5-B. Consequently, the data will have to be reorganized for correct display. Figure 6 shows the desired organization. This reordering can be done either before or after computation. Performing this operation after computation adds an enormous overhead that will be incurred every time the image is updated (figure 7). Another method is to read the result data pixel by pixel rather than using a single all-encompassing read. Unfortunately, each read requires an enormous amount of overhead, due to the need to decode the data. This decode time is reduced significantly with mass reads.

5.4.1 Previous Work

Much of the previous work done on data mapping issues is concerned primarily, if not solely, on the computational costs of the algorithms themselves. Thus, most, if not all, of the heuristics previously developed attempt to minimize non local communication [HIL93][MIG91][WIE92]. Hillis and Boghosian even suggest that it may be beneficial to perform extra arithmetic in order to reduce communication [HIL93]. The primary reason for this concentration is that the time required to generate a single display, even on a supercomputer, for the applications under investigation forbid the generation of real-time displays. Consequently, the cost of displaying the resulting images is not of significance.

Extensive work has been done to ensure that appropriate load balancing is done on MIMD architectures to maximize the amount of parallelism achieved [CAM94]. This research, as with the other research areas discussed, has not taken any requirements for the resulting data (e.g. its organization) into account.

In [SCH94], Schechter et al. describe the use of massively parallel systems for the visualization of ultrasonic pulses in solids. While they are applying massively parallel computers to visualization, they don't discuss the impact of data layout on the display of their results. Their routines operate such that there is an initial startup time, during which most of the computation takes place and after which the application executes in real-time. Additionally, they used a CM-2 with a dedicated frame buffer for their real-time visualization, and thus the display time of the resulting data was very likely less of an issue than in our case. This is an area in which our research differs, we are using a supercomputer attached to a standard workstation and are attempting to generate real-time performance from the limited capabilities of this system.

Crow et al. discuss much of the previous work that has been done on the CM-1 and the CM-2 as related to graphics and rendering [CRO92]. While the CM-2 provides a high speed frame buffer connected directly to the backplane of the supercomputer, the CM-1 required use of the host computer's framebuffer. The situation with the CM-1 is similar to the situation discussed in this paper. They make note of the fact that each display of a comparably sized image requires five seconds on a CM-1 as compared to the .2 seconds it takes our system. Unfortunately, they do not discuss any of the issues related to this display time.

None of these works, unfortunately, discuss situations in which it may be beneficial to use other heuristics than the most commonly established norms. The computational cost of most systems being analyzed are too demanding to be able to consider the details/importance of the data layout issues as they apply to the real-time display of the resulting data.

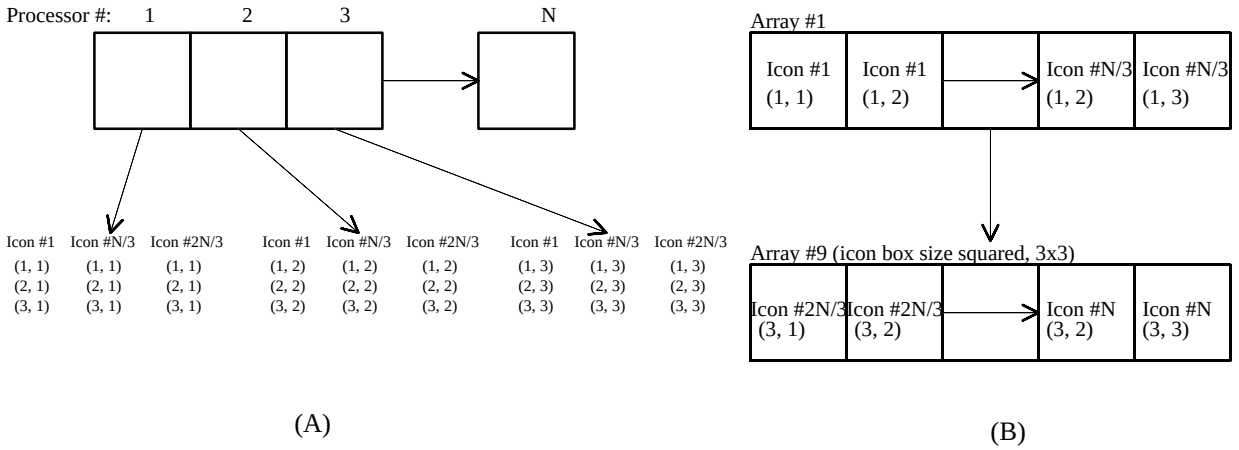

Fig.8: Modified data layout for a sample 3×3 icon. Each icon is divided among multiple processors.

5.4.2 Data Layout as Organized for Display

An alternative is to reorganize the data before computation is done, e.g., when the original data is first loaded onto the processor array. This approach incurs a penalty whenever the data must be reloaded onto the processor array, but it incurs no additional overhead when the image is updated. This latter method is by far the most efficient for our needs and was thus chosen for implementation. In our scheme, the data is reloaded onto the processor array when either the parameter-to-limb mappings change or the selected region of the database under investigation is changed. Manipulation of any other control does not require the data to be reloaded and thus does not incur a penalty.

Our allocation scheme is based on the width in pixels of each icon box. The data mapping onto the processor array is performed as follows: All data values associated with an icon are placed in a single processor. Each successive icon is separated by a number of processors equal to the width in pixels of the icons. This skipping of processors will force the mapping algorithm to wrap around the processor array, which is performed a number of times equal to the width of the icons. After the algorithm wraps around the processor array, it increments the starting processor; allocating an equal amount of data to each processor. When interpolation is performed, the data values will need to be sent both to the left and to the right to get the results in the desired position. Interpolation is performed as follows: The processor which contains the relevant data calculates the four corner values of the icon. The column of values appropriate to the processor is then calculated. Figure 8-A shows the organization of the resulting columns each processor is responsible for. The data is then passed to the left such that any preceding columns of the icon are calculated. The data is passed to the left and then to the right an appropriate number of times such that all columns in each icon are calculated. After reading the data values from the Terasys hardware, the data is organized as in figure 8-B.

It is important to note that if the number of processors is not evenly divisible by the width of the icons then the algorithm will only use the maximum number of processors that is evenly divisible by the width of the processors. This removes special cases in which we must handle communication which wraps around the processor. We are losing a bit of performance by doing the interprocessor communication but not nearly as much as if we had to completely reorganize the data.

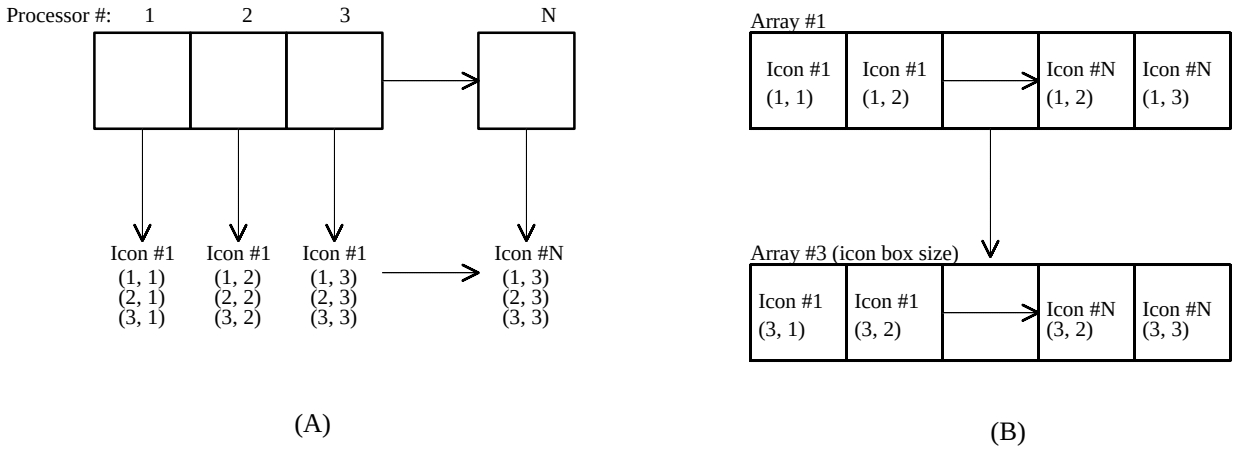

Fig. 9: Modified data layout for a 3×3 icon with a large number of processors. One column of pixels per processor.

5.4.3 Influence of System Size on Data Organization

Another issue is encountered when a larger number of processors is available. The mechanism we are using works well because the current machine we are developing on contains only 4K processors. Here we make a tradeoff between the cost of displaying an image and the image size. Having each processor generate its own icon provides a usable image size but slows the display down slightly; instead of 64´64 pixel images we are generating 320´320 pixel images. The performance loss is acceptable because of the need for the increased display area. Should a machine with a larger number of processors be made available to us, say 32K processors, then it may be more beneficial to take an alternative approach. The main influence here is the cost of actually displaying the data. As mentioned previously, this is currently the greatest bottleneck. Adding additional processors will increase the display time linearly while the computation time will remain constant. Thus, for a large number of processors it may be more beneficial to reduce the execution time slightly by having each processor generate a one-pixel result instead of an entire icon. This would provide an image of 178´178 pixels, which is rather small, but will only take a fraction of the time to display of an 890´890 pixel image.

The conclusion we draw is that a combination of the two methods may be useful, namely, using the current method, except that each processor only calculates one column or row of N pixel values¾as opposed to N columns or rows of N pixels as is done now. The best implementation of this method has not yet been determined, but difficulties will arise due to the dependence between rows and columns and the need to organize the data for display. The need to organize the data for display implies the allocation of one column per processor. In this approach, all data values associated with an icon are duplicated on a number of processors equal to the width of the icon. All associated processors are adjacent. Interpolation is then performed by having every processor calculate the values of the four corners of its icon and then calculating the appropriate column of values. This requires redundant computation¾the corner values of each icon are calculated multiple times¾but would still improve computation time greatly. Figure 9-A shows the resulting data as mapped onto the processor array, while figure 9-B shows the resulting data organization after being read from the Terasys hardware.

This combined method will remove most of the interprocessor communication and will reduce the number of operations each processor must execute. Thus, the computation time required for rendering will be reduced and the maximum size of the image will be more limited than in the currently implemented method. This should not be problematic due to the cost of displaying a larger

image. Given the high cost of sending data to the display under the current system, the improved computation time may not be noticeable. The reduced size of the image window will actually provide more of a speedup. Should a better display mechanism be provided, e.g., a framebuffer directly on the supercomputer, then the improved computation time will have a much greater impact.

5.4.4 Impact on Compiler Technology

There are two areas outside of algorithm implementation which are affected by the results of these data mapping paradigms, compiler technology and dynamic load balancing. Automatic parallelizing compilers are of particular interest and are looked upon as an important future goal to aid in the acceptance of parallel systems by the general scientific communities [CAM94]. Both compiler technology and dynamic load balancing algorithms [MIG91][HIL93] for parallel systems need to take the impact of these results into account. Generally, these technologies rely on the assumption that an algorithm will perform more efficiently if the interprocessor communication is reduced [MIG91]. While this may be true for most computational purposes, our example, in which we wish to display the resulting data very quickly, contradicts this assumption.

As compiler technology and dynamic load balancing algorithms for parallel systems improves, the compiler will allocate processors and map data more effectively. Unfortunately, the current trend seems to be to direct this attention primarily at the computational costs of the algorithm. Additional work needs to be done on the impact of outside influences (e.g. the need to display the results). Consequently, both of these technologies should provide mechanisms by which the implementor can adjust the heuristics for outside influences, which would normally not be considered. Badrodia and Mathur discuss an environment which provides some of these capabilities. This is the UC environment, which is based on the concept that a different mapping strategy could greatly improve performance [BAG91]. Consequently, this environment provides several routines (permute, fold, copy, and reduce) which allow the implementor to modify the default data to processor mappings generated by the compiler. This is a good beginning but extensive research remains to be done in this area.

5.5 Processor Memory Usage

As mentioned previously, the Terasys hardware is limited in that each processor is currently only provided with 2K-bits of local memory. This is a serious limitation that greatly limits the applicability of algorithms and makes implementing new applications on the hardware extremely complex. The task of carefully allocating this memory space is complicated even more by the fact that some of the space is allocated by the compiler for system and temporary variables. To put this in perspective it is useful to note that the Connection Machine CM-2 supports 8KB of memory per processor, providing 32 times more memory per processor than the Terasys.

The memory limitations of the Terasys forces the use of memory to be an important issue. The method used to render the image generates certain constraints on what data must be in memory at a given time and how much memory must be available for the result data. Crow et al. describe the amount of memory required for image synthesis algorithms and the limitations of the CM-1 with respect to memory [CRO92]. The CM-1, though limited in its memory capacity, provided 4K bits of memory¾twice that provided by the Terasys. Beyond the memory required by the rendering method, the available memory may be used in different ways. A tightly coupled system would have only a minimal amount of information on the Terasys at any one time. This information would be what is required to generate a display of the currently selected region of the database. This approach simplifies coding significantly. An alternative is to have the renderer loop through the entire database, rather than only the currently selected region. This provides the advantage of being able to show the entire image at once or being able to pan over the remainder of the image with trivial

display times. The authors believe that this is not a significant enough advantage to outweigh the cost of having all other control changes take excessive amounts of time, especially considering the fact that the user is often primarily interested in only a particular region. Perhaps having a switch to change between the two methods would be beneficial. In this way, the user could generate usable control settings using quick updates on a selected region and then render the entire database, which could be panned around without further computation.

A more interesting modification that could be made to the current implementation would be to buffer the information in the Terasys memory more intelligently (e.g. apply database techniques to the memory hierarchy). Because the amount of Terasys memory is limited, only the parameters needed for the current display are maintained on the hardware. When a parameter is changed, all of the parameters are reloaded when the next redraw is requested. The simplest modification would be merely to load the parameters that have changed and leave the rest of the parameters alone; however, because the memory locations of the parameters will rarely be in the expected positions of the simple implementation, this approach would require the additional overhead of maintaining the locations of the parameters in a table. An added improvement to this implementation would have us provide more than the minimal amount of space for the needed parameters. This would allow us to maintain a few additional parameters in the buffer.

An interesting issue is how to handle this buffer such that the parameters in the data set currently being analyzed are buffered efficiently. In other words, when we already have as many parameters on the Terasys hardware as memory allows¾the buffer is full¾then we wish to remove only those parameters that are least likely to be used again. In addition, efficient mechanisms must be provided to allow the renderer to retrieve parameters from the buffer. Since implementing this buffer will add an additional level of overhead on top of the renderer it remains to be seen to what extent providing this buffer actually improves performance.

6 Performance

The Terasys hardware proved to be capable of performing the calculations required for our visualization system fast enough to generate pseudo real-time displays. The actual display of those images proved to be severely limiting and greatly decreased the frame rate. This is mainly a result of the S-Bus, which is incapable of the throughput needed for this type of application [ERB95]. Since both the Terasys hardware and the graphics framebuffer are on the S-Bus, we must pass the data over the bus twice. The first pass brings the data from the Terasys hardware to the CPU, and the second pass sends it from the CPU to the framebuffer. On our system, the current implementation of the algorithm generates a 64´64 icon image (320´320 pixels). This image is calculated on average in less than .05 seconds. The actual display of the image adds approximately .15 seconds. Thus, while we could calculate 20 frames per second without display we are limited to only 5 frames per second with display. This image calculation time can also be improved on larger systems with a different implementation of the rendering algorithm, as discussed above.

Improving the display time is a much more complicated issue. Because both the Terasys hardware and the framebuffer are sitting on the S-Bus (figure 4), a system-friendly display routine will require that the data to be displayed make two passes over the S-Bus. The first pass sends the data from the Terasys hardware to main memory, where it is then passed to C, which displays the image. We were able to increase performance by bypassing C altogether and sending data directly to the framebuffer. We are still limited by the need to send data across the S-Bus twice; eliminating this need would improve performance dramatically. Another alternative would be to incorporate a framebuffer directly onto the Terasys hardware. This would completely eliminate the need to transfer displays over the S-Bus.

7 Applications and Examples

In a recent presentation we were attempting to describe, in detail, the effects of the different controls and color models that affect the results of the color icon to individuals without strong backgrounds in computer graphics. The parallel version of the color icon provided a mechanism by which they could get a feel for the effects rather than having to learn purely conceptually. The parallel version was of particular use when describing the LHS color model, which the individuals had no experience with. This worked particularly well in conjunction with a description of the LHS color space. The color space description allowed them to attain a general idea of how the fields work, but actually allowing them to change the amount of lightness in the image and watch the image brighten and darken provided them with a truly useful understanding of the process. Hue and saturation worked especially well as they are much more difficult to conceptualize without experience.

This experience was very useful in that we were able to get feedback concerning the value of manipulating controls in real-time. One of the goals is to provide a system where the individuals can make more intelligent decisions as to what parameters or controls to change to affect the display in particular ways. This capability should help users generate more useful displays in less time.

8 Future Work

The computational cost of the serial algorithm for rendering each color icon varies not only with the color model selected, but also according to the data values given to the rendering routine. The reason for this is that the rendering algorithm depends on conditionals. Because each icon is generated from different input data, the serial version requires varying computational time for rendering each icon. By contrast, the SIMD paradigm forces all processors to execute in lockstep, and thus shows identical computation time for each icon generated. Every processor must step through each instruction in a conditional even if the instructions are not to be executed because the conditional evaluated to false. This behavior wastes cycles that could be spent on other tasks. The renderer is thus best suited for MIMD parallelization, which should offer a speedup over the SIMD implementation described here, since a MIMD parallelization will not waste as many cycles. This analysis is based on the computation time required to generate icons and ignores the issues of displaying the data. Consequently, future projects should include an implementation of this environment on a MIMD architecture, should one be made available to us, in order to assess whether any speedup over SIMD execution is actually gained.

Work needs to be done to develop a true real-time implementation. The primary task here would be to improve the display time such that it more closely matches calculation time. In addition, the user interface needs to be examined more rigorously with test subjects to determine its strengths and weaknesses in a real-time environment.

9 Conclusions

The use of floating point computation on a SIMD architecture is an as of yet unresolved issue that requires further examination. Several approaches, both hardware and software, have been discussed¾all have benefits and limitations.

The current basis for most of the data mapping strategies being used for algorithm implementations can be summarized as follows: “the problem should be divided in such a way as to let each processor work as independently as possible” [WIE92]. We have shown that for a system which requires real-time display of the resulting data, this data mapping strategy may not be appropriate. We provided examples of how to better map the data onto processors for this type of application. These

results are, on the other hand, partly dependent on the fact that the display (and any required data reorganization) are the primary source of execution time for the application.

We have discussed different methods of using available memory on an architecture with very limited memory resources. We also discussed the use of database technology to assist in the use of this memory. These are issues that do not present themselves in most systems as most systems generally provide more adequate resources.

We have shown that a pseudo real-time implementation provides necessary and beneficial characteristics to the application which make it more comprehensible and therefore more useful.

References

[BAG91] Rajive Bagrodia and Sharad Mathur, "Efficient Implementation of High-Level Parallel Programs," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 142-151, 1991.

[CAM94] W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland, "Massively parallel methods for engineering and science problems," *Communications of the ACM*, April 1994, Vol. 37, No. 4, pp. 31-41.

[CRO92] Franklin C. Crow, Gary Demos, Jim Hardy, John Mclaughlin, and Karl Sims, "3D Image Synthesis on the Connection Machine," *Proceedings of the Conference on Scientific Applications of the Connection Machine*, World Scientific Publishing Co., pp. 260-281, 1992.

[ERB95] Robert F. Erbacher and Georges G. Grinstein, "Performance issues in a real-time true color data display," *Proceedings of the SPIE '95 Conference on Visual Data Exploration and Analysis II*, San Jose, CA, February, 1995.

[HIL93] W. Daniel Hillis and Bruce M. Boghosian, "Parallel scientific computation," *Science*, August 13, 1993, Vol. 261, No. 5123, pp. 856-863.

[LEV88] H. Levkowitz and G. T. Herman, "Towards a Uniform Lightness, Hue, and Saturation Color Model", *Proceedings of the SPIE Conference Image Processing, Analysis, Measurement, and Quality*, pp. 215-222, 1988.

[LEV91a] H. Levkowitz, "Color Icons: Merging Color and Texture Perception for Integrated Visualization of Multiple Parameters", *Proceedings of the Visualization '91 Conference*, IEEE Computer Society Press, San Diego, CA, October 22-25, 1991.

[LEV91b] H. Levkowitz and G. T. Herman, "A Generalized Lightness, Hue, and Saturation Color Model", Department of Computer Science, University of Lowell, R-91-004, 1991.

[LEV93] H. Levkowitz and G. T. Herman, "GLHS: A Generalized Lightness, Hue, and Saturation Color Model", *CVGIP: Graphical Models and Image Processing*, Vol. 55, No. 4, July, pp. 271-285, 1993.

[MAR93] Jennifer Marsh and Mark Norder, "Pim Chip Specification", Technical Report SRC-TR-93-088, Super Computing Research Center Institute for Defense Analyses, Bowie, MD. 1993.

[MIG91] Serge Miguet and Yves Robert, "Elastic load-balancing for image processing algorithms," *Parallel Computation - First International ACPC Conference Gmunden*, Springer-Verlag, pp. 439-451, 1991.

[MIS94] Manavendra Misra and Terry Nichols, "Computation of 2-D Wavelet Transforms on the Connection Machine-2," *Proceedings of the IFIP WG10.3 Working Conference on Applications in Parallel and Distributed Computing*, pp. 3-13, 1994.

[PIC88] R. M. Pickett and G. G. Grinstein, "Iconographic Displays for Visualizing Multidimensional Data", *Proceedings of the 1988 IEEE Conference on Systems, Man and Cybernetics*, Beijing and Shenyang, People's Republic of China, 1988.

[SCH94] R. S. Schechter, H. H. Chaskelis, R. B. Mignogna, and P. P. Delsanto, "Real-time parallel computation and visualization of ultrasonic pulses in solids," *Science*, August 26, 1994, Vol. 265, No. 5176, pp. 1188-1192.

[SMI91] S. Smith, G. Grinstein, and R. D. Bergeron, "Interactive Data Exploration with a Supercomputer", *Proceedings of the IEEE Visualization '91 Conference*, IEEE Computer Society Press, San Diego, CA. 1991.

[SWE92] H. Douglas Sweely, "Terasys Demonstration Hardware Manual", SRC No. 101.93, Super Computing Research Center Institute for Defense Analyses, Bowie, MD. 1993.

[WIE92] Kathleen K. Wiegner, "Parallel thinking," *Forbes*, Feb 3, 1992, Vol. 149, No. 3, pp. 92-93.