Developing Pedagogical Visualizations of Dense Matrix Operations on Interconnection-network SIMD Computers

Justin C. Miller
Computer Science Department
University of Wisconsin Oshkosh
neonprimetime.geo@yahoo.com

Abstract

Parallel algorithm animations provide graphical illustration of a parallel computer algorithm. Parallel algorithms can be difficult for students to understand, but it is possible with the right tools, to improve student's understanding of such algorithms. This paper presents specific detail for creating pedagogical visualizations of parallel algorithms for dense matrix operations on interconnection-network SIMD computers.

Section 1 will discuss the motivation behind building visualizations of parallel algorithms for dense matrix operations on interconnection-network SIMD computers. Section 2 will explain exactly which parallel algorithms this paper addresses. Sections 3 & 4 will discuss many strategies that should be used when creating these visualizations. Section 5 provides discussion about several specific algorithms. All of the strategies and discussions are based upon research I performed for my thesis as well as an informal case study I performed on ten computer science undergraduates.

Introduction

Drawing a parallel algorithm by hand can lead to confusion for several obvious reasons. One big problem with drawing parallel algorithms by hand is the amount of calculations and passing of data that occurs "in parallel". The instructor must do lots of erasing and re-drawing as quickly as possible in order to make things appear to occur in parallel. Another issue that adds potential for confusion is the common occurrence of an instructor making a small mistake while drawing. In a parallel algorithm, this small mistake could quickly manifest into tremendous confusion. In general, as mentioned by Bergin et al. (1996), it is very difficult to draw any procedure that involves two dimensions or more, such as matrices.

Since drawing by hand has so much potential for adding undesired confusion to a classroom setting, instructors can turn to creating a visualization of such an algorithm as an alternative. First, and most importantly, the drawing process of such a difficult algorithm could be automated. By having a predefined visualization to use in lecture, an instructor would eliminate the possibility of mistakes made by hand. Another added benefit is, after investing time into creating a visualization, the instructor can use the same animation from semester to semester. This would mean that a good section of a lecture is already created, thus making for quicker preparation time for lectures and allowing more time to do other things. Another added benefit of using a visualization in a lecture is the notion that it attracts student attention, as mentioned by Bergin et al. (1996). Also, if designed properly, these visualizations can not only be used during a lecture, but also outside of class for students to study from or even take online quizzes.

Many papers have been written that perform case studies on students and record their benefits from usage of visualization, like Stasko, Badre, and Lewis (1993), Naps, Eagan, and Norton (2000), Rößling and Freisleben (2000), Mayer and Anderson (1991). In general, the results show that while an animation definitely doesn't hurt a students understanding, it doesn't provide the "magic cure-all" that will make every student get perfect scores on exams. Instead, it seems that the animations provide perhaps a slight increase in performance, but maybe more importantly they provide added instructor benefits such as eliminating the potential for mistakes, making for quicker preparation time because of re-use of animations, and also catching the interest of students.

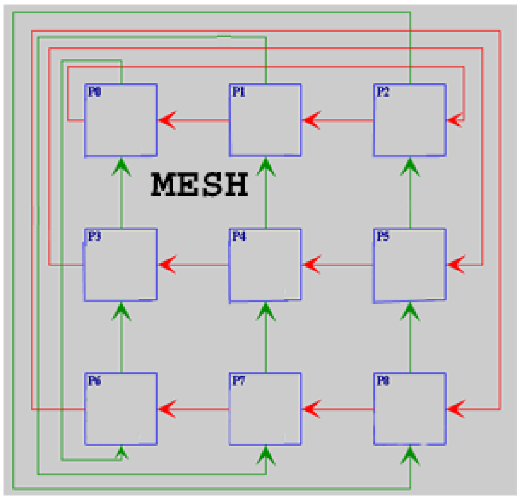
Which Parallel Algorithms?

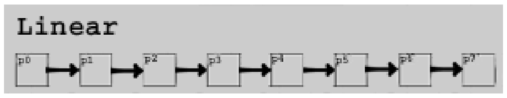
In this essay, the strategies that will be presented are for a specific class of parallel algorithms. This class includes only algorithms that are for dense matrix operations such as multiplication, transposition, and matrix-vector multiplication

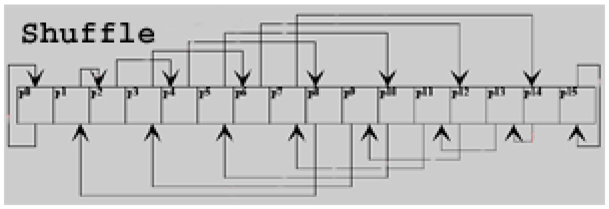
This class is also confined to algorithms that follow the SIMD model of computation as described by Akl (1989). This model allows for each processor to receive the same instructions (**S**ingle **I**nstruction), and the only difference from processor to processor will be the data it is working with (**M**ultiple **D**ata-streams).

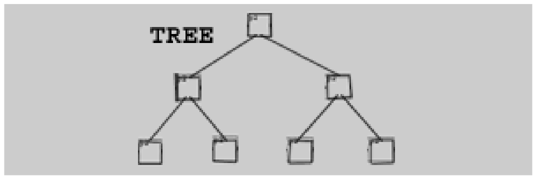
Finally, the class of algorithms discussed in this essay fall under the subclass of SIMD computers known as Interconnection-network SIMD computers as described by Akl (1989). Interconnection-networks, as opposed to shared memory, allow processors to communicate through a network of channels instead of through shared data. Within the class of interconnection-network SIMD computers, there are several simple networks Akl (1989) describes, and they are pictured below.

Figures 1-4: Each simple network is picture below









These networks define how the processors in a SIMD computer can communication. The common networks discussed in this essay are 1.) Linear Array 2.) Two-Dimensional Array or Mesh (wrap-around connections are optional) 3.) Tree Connection 4.) Perfect Shuffle Connection (neighboring connections are optional). Wrap around connections for mesh networks are defined as connections that go the last row to the first row and or the last column to the first column allowing those processors to communicate.

Since the simple networks of this particular class of parallel algorithms are so visual by nature, as seen in figures 1-4, their algorithms have great potential to be animated. The remaining parts of the paper will now discuss what strategies should be used when creating such animations. First, the paper will discuss several general strategies that apply to all algorithm animations. Then there will be a section describing several strategies specific to dense matrix operations on interconnection-network SIMD computers. Finally, there will be a section that analyzes the details of animating several specific algorithms.

General Strategies

Many papers have already been written describing some general strategies that should be used in all algorithm visualizations. Still, it is worthwhile to briefly cover some of these important general strategies that thus hold true for dense matrix operations on interconnection-network SIMD computers as well.

Use Everywhere

One major factor that comes into play is designing a visualization that will run across multiple platforms so it can be used during lectures, labs, or at a student's home. Luckily, this is not a major concern because tools have already been created to help build "use everywhere" visualizations. The tools used and discussed in this paper are Rößling and Freisleben's (2001) AnimalScript and Naps and Chan's (1999) JHAVÉ. Both are written in Java, thus they have the capability of being run on any machine that has a Java Virtual Machine, and they will look the same across all platforms. AnimalScript is a scripting language, and so one must simply write a program in any language, and have it output some script into a file. The Java-based Animal program will then use the script file to generate an animation. JHAVÉ is a web-based client-server tool that offers the capability of taking an AnimalScript file and introducing such features as HTML pop-up windows and stop-and-think/quiz questions.

Descriptive Text

Providing descriptive text, such as HTML pop-up windows in JHAVÉ or simple text on the animation screen in Animal, have been well discussed by Mayer and Anderson (1991), Rößling and Naps (2002), and Kehoe (1996). During each step of the algorithm, there should be some sort of descriptive text describing the action that is occurring. It is also important that if the descriptive text is going directly on the animation, it should to have a high-contrast text-box so that the text is easily readable. Since Animal allows you to do layering, the high-contrast text-box is very easy to create.

Step Through, Rewind, and Video Motion

It has been well documented by Naps et al. (2000) and Rößling and Naps (2002) that there is a need to have full forward and rewind capabilities. Full capabilities are defined as an animation that allows the user to manipulate the algorithm by going through one step at a time (forward or backwards) or playing it in full motion like a video. It is necessary to have rewind capabilities because, if a student has a question about a previous slide, the instructor won't have to re-run the entire visualization. Being able to step forward in the animation is necessary so that the instructor may go through the animation, one step at a time and explain what is occurring. Video motion is also a necessity because there are many concepts that become easier to grasp when they're seen occurring in a smooth motion. Fortunately, by using Animal and JHAVÉ, all the features mentioned above are provided.

Fancy Graphics != Quality Animations

One final important issue already described by Miller (1993) and Rößling and Naps (2002) is the concept that fancy graphics are not necessary to provide a quality animation. Instead, an animation should be simple, yet effective. Thus, it is necessary to only use a few simple fonts, and only a few high contrast colors in the animation. It has also been mentioned in a few papers that red-green color blindness is quite common, and therefore one should not attempt to use red versus green to distinguish important differences in an algorithm. By using tools such as Animal and JHAVÉ, you can easily manipulate the fonts and colors that you use in the animation.

Specific Strategies

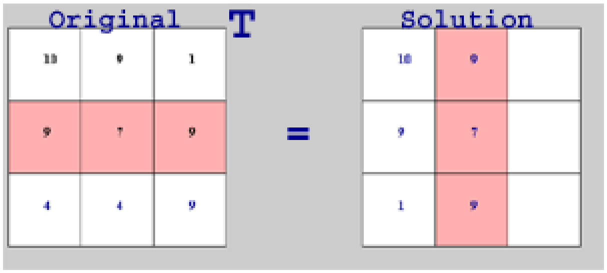
Along with the strategies mentioned in Section 3, there are more specific strategies that deal directly with developing visualizations of dense matrix operations on Interconnection-network SIMD computers. Through my thesis research and an informal case study of ten computer science undergraduates, I have determined that the following strategies should be applied to any visualizations of a dense matrix operations on Interconnection-network SIMD computers.

Show Original and Solution Matrices

In the attempt to visualize any dense matrix operation, one will most likely have part or most of the animation devoted to the many processors working on the algorithm. It is also very helpful to visualize the original matrices the algorithm is working with and keep it displayed throughout the entire animation. Also, it is a very good idea to provide an empty matrix that, by the end of execution, will hold the solution. For example, when visualizing a transposition one should devote most of the screen to the processors that are passing data and number crunching, but there should also be a visual assignment statement (Figure 5) with the original matrix on the left, and an empty solution matrix that by the end of the animation will contain the solution. For a multiplication, there should be another visual assignment statement that seems to be multiplying the two matrices and assigning it to the solution matrix. The purpose of having the original matrix animated on the screen is so before the algorithm starts execution, steps can be created that take the values in the original matrices and properly disperse them into the processors' registers. This can help the student understand how or why the processors receive initial values in their registers. The empty solution matrix is displayed because, for all networks except the linear tree, in general each processor will halt, holding a particular number that belongs in the solution matrix. This is very convenient then, because now each processor can place or pass its value to the correct location in the solution matrix. In general, the original and solution matrices should be smaller than the processors that are drawn, because the processors are obviously of more importantce.

Finally, for matrix transpositions specifically, having both the original matrix and solution matrix displayed can allow the animation to do a simple but effective proof that the transposition worked by highlighting each row in the original and the corresponding column in the solution matrix (see Figure 5). Since the Animal scripting language will allow you to move objects and change their colors, it is possible to do everything with the original and solution matrices that was mentioned above.

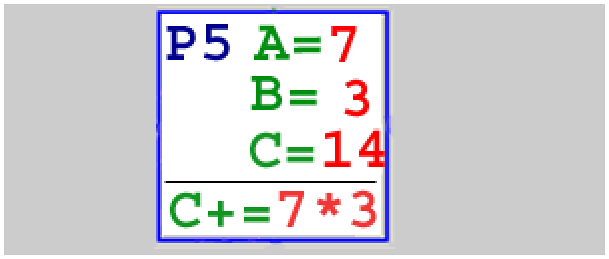
Figure 5: This is a visual assignment statement for a matrix being transposed. Currently, highlighting is occurring to prove or re-emphasize that the transposition worked properly.



Processors

When visualizing a parallel algorithm, it is necessary to display all the processors involved. These processors are connected in some network mentioned in Section 2. It is important to give each processor a label, for example P0, P1, P2, etc. The labels are needed so that in class an instructor can tell students to focus on a particular processor, and they will easily be able to find it and do so. In the matrix operations, a processor is usually required to hold some data in its local registers. Thus, in the animation it is necessary to show these registers and their contents at all times. For example, in a multiplication, there might be three registers: two for the numbers being multiplied and one for the sum accumulated by the multiplications. These registers can be visualized either by having a simple assignment statement inside the processor or by having a small box and label to designate a register. One final thing needed inside a processor is an "action window". This is an area in the processor that notifies the viewer what action the processor is taking at that current time. For example, if currently the processor is multiplying it's two registers (A and B) together and adding it to another register C, the action window could contain a statement similar to C += A * B, or as seen in Figure 6, it may be better to show the two numbers being multiplied, for example: C += 7 * 3. An action window is necessary in an algorithm where a calculation is occurring, but it could also be used in algorithms where the main action is passing and receiving data. In the cases where the processor is actually just passing and receiving data, the action window could show statements such as "passing data", "receiving data", or "reading from the input queue". The Animal scripting language will easily allow you to animate the processors in a readable, well organized way like mentioned above.

Figure 6: This processor (P5) is currently performing a multiplication and storing the sum. Notice the three registers and their values denoted by assignment statements. In this example, the action window contains the statement C += 7 * 3, and the sum (21) will soon be added to register C's total.



Communication Channels

Communication channels are vital to the visualization of any algorithms done on an Interconnection-network SIMD computer. The communications channels, usually animated as a line connecting two processors, define how processors can communicate with each other. Depending on the network (described in section 2) the algorithm is built for, the channels will be located in different locations. In order to help the viewer better understand how the channels work, arrows heads should be placed on the channel in the direction the data will be passed. If a pair of processors both pass to and receive from each other (bi-directional), it is necessary to then draw two separate lines instead of drawing one line with arrow heads at both ends. The reason is that during the animation numbers will literally be sliding along these lines, and it will become confusing if more than one number is traveling across the same line at the same time. The Animal scripting language allows you to move objects from one point to another, and therefore it is quite easy to make numbers smoothly slide across a channel from one processor's register to another.

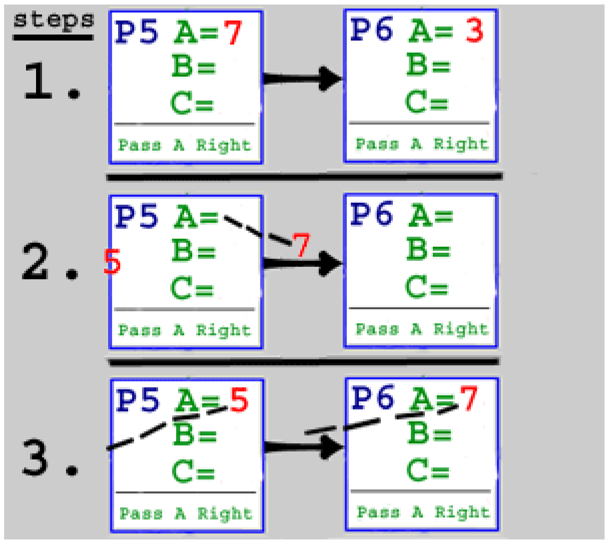
Input Queues

In some algorithms, certain processors are receiving input from some sort of queue. In these situations, processors do not receive initial values immediately from the original matrix. Instead, the initial values are passed directly from the original matrix to the queue and stored there until the processor needs them. In order for the processor to then grab the next element from the queue, there should be some sort of communication channel drawn between the queue and the processor. Then, when the processor wants to grab the next element, the element can slide smoothly across this channel into the appropriate register location where the processor can perform its calculations. Once again, since the Animal scripting language allows you to move objects, creating and manipulating an input queue is made quite easy.

Slide by Slide Abstraction

As discussed in several papers, there needs to be a certain level of abstraction when visualizing algorithms. In particular, when visualizing a parallel algorithm, all the communication delays and differences in processor speeds must be abstracted. What this means is that all processors must be synchronized so that they all pass data at the same time and they all do calculations at the same time. Fortunately, in AnimalScript “slides” can be created by putting curly braces around several script statements. A “slide” forces all operations within the curly braces to occur at the exact same time. In Figure 7, the next instruction is for every processor to pass its data in register A to the right. Since this action was created on a “slide”, the animation will show *every* processor passing its data right at the exact same time. When creating slides using the Animal scripting language, I initially used multiple threads to solve the algorithm and pretend things were happening in parallel. What I learned was that for the purpose of creating animations and “slides” in particular, it is much easier to just write a purely sequential solution to the parallel algorithm and use the “slides” to make it look like it’s happening in parallel. If you are using threads, then because of their unpredictable scheduling, the animation can begin to look chaotic and confusing.

Figure 7: Processor P5 is passing it’s ‘A’ value to the right.
During the 3 steps, the dashes show where a number has been.
The dashes are not in the actual animation, just in this picture.



Tracking Data

In some instances there may be a desire to follow a particular data element throughout the execution of a program. This can be accomplished by giving this particular data element a different color that all the other data elements and perhaps even bolding it. This would be particularly useful when visualizing the mesh transposition algorithm that has one of the more confusing paths the data must follow. In general, tracking data becomes very useful in helping a student understand the path data travels during execution. Since animal allows you to change the colors of any object, tracking data can easily be accomplished.

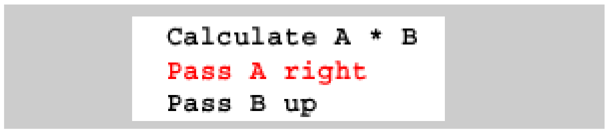
Matrix Size

One crucial issue when creating an animation is choosing a good default size for the matrices. Obviously, as the matrices get larger, the screen fills up and therefore it becomes quite hard to even follow the animation. Also, as the matrices get larger, the complexity increases and the animations become more confusing than helpful. Because of these reasons, it may be necessary to have a maximum size that your animation can handle. This maximum size is the largest matrix sizes that can be manipulated in a "visually pleasing" and understandable way. Normally, command line parameters or pull-down menus in JHAVÉ will allow the user to choose the size of the matrix being manipulated. Still, it is necessary to choose one default size that is most "visually pleasing" and easy to follow. Thus, the default size would kick in if no command line parameters were given or if the user just wanted to view the generic animation. Typically from my studies I've discovered that 3 by 3 or 4 by 4 matrices are good default sizes.

Pseudo Code

Many papers such as Rößling and Freisleben (2000), have talked about the importance of showing pseudo code that traces the execution of the program. Since these are SIMD (that is single instruction) computers, showing pseudo is possible because all the processors share the same instructions. It is important, just like the descriptive text mentioned in Section 3, that the pseudo code is placed on a high contrast "text-box" so that it is readable. Of course, it is also then necessary to highlight the current line of pseudo code that is executing. During my studies, I found that pseudo code was only helpful for certain algorithms. Specifically, the algorithms such as in Figure 7, where every processor is performing the same generic action (like passing data right) then the action windows should be substituted with on section of pseudo code. Pseudo code sections can be created easily with the Animal scripting language or it can be displayed in a JHAVÉ HTML pop-up window. One thing to note is that, as of the time of this writing, if text is placed in a JHAVÉ HTML pop-up window, it is static text and therefore highlighting of the active line cannot be accomplished.

Figure 8: Sample pseudo code section that could appear during an animation. Notice that the second line is highlighted, which means it is active.



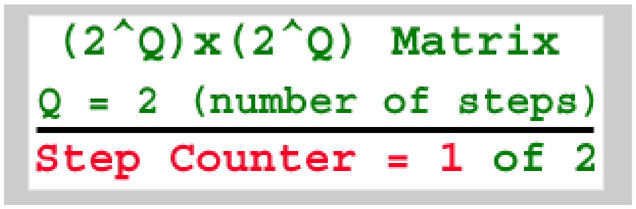
```
Calculate A * B
Pass A right
Pass B up
```

With many of the algorithms though, different processors end up executing different parts of the code at the same time, as will be seen in the next section. For example, if the code told odd numbered processors to do one thing, and even numbered processors to do another, then the odd and even numbered processors would not be executing the same line of code, and thus pseudo code wouldn't be feasible. With these algorithms, it is then necessary to use the action window, so that it can be specifically shown what each processor is doing.

Step Counter

Finally, each algorithm executes a certain number of times relative to some variables (usually the matrix size). It is nice to follow these algorithms step-by-step and so it is logical to have some text that displays what step the algorithm is currently executing. It may also be useful to display the equation that determines how many steps the algorithm runs in and also display the values of the variables this equation relies on. Once again, it is important to put all this text on a high-contrast "text-box" to make it readable. The step counter can easily be created using the Animal scripting language.

Figure 9: A step counter that includes vital information to the algorithm. The step counter would update on each pass of the algorithm.



```
(2^Q)x(2^Q) Matrix
Q = 2 (number of steps)
Step Counter = 1 of 2
```

Algorithm Specifics

Now it's time to give specific attention to the different parallel algorithms that fall under the category of dense matrix operations on Interconnection-network SIMD computers. In this section, there will be a brief discussion of the issues that arose while creating visualizations of the different parallel algorithms I worked with during my research.

Mesh Multiplication

To accomplish a mesh multiplication, such as the ones described in Akl (1989) and Chaudhuri (1992), the mesh network described in Section 2 must be used (without wrap-around connections). These algorithms require that certain "edge" processors have an input queue (as mentioned in the previous section) that holds the numbers from the original matrices. In these algorithms, typically the processor would grab from the input queue as soon as the processor had an empty register to fill. In order to make the visualization easier to follow though, the "edge" processors should not grab a number from the input queue until they receive a number from one of their neighbors. Thus, the processor seems to only grab data when it's preparing to do some calculations on that number. The processor receives both the numbers it needs, multiplies them, adds the total to its other register, and passes the numbers to the appropriate neighbors.

It is also interesting to take a look at the scenario that occurs when a processor is preparing to pass a number to its neighbor, but it turns out to have no neighbor (it's in the last column or row). When this occurs in the visualization, the number can be considered "dead", and therefore it needs to be discarded. The best way to discard a number is to just make it disappear using the AnimalScript hide feature.

Torus Mesh Multiplication (Canon's Algorithm)

The torus mesh multiplication as described in Lester (1993) and Chaudhuri (1992) uses the mesh network that includes wraparound connections as described in Section 2. The wraparound connections can be tricky to design visually. To make the design as easy to understand as possible, the wraparound channels should be drawn together and travel around the entire matrix (see Figure 2). This way, the concept of each processor passing data through the wraparound connection can be grouped together and seen easily.

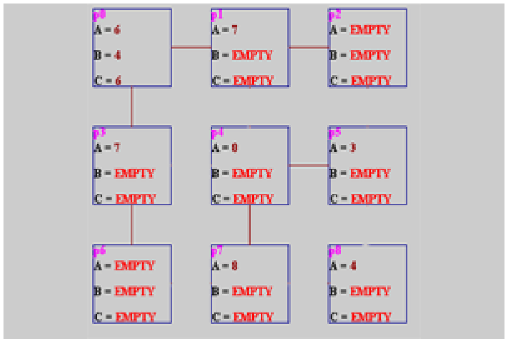
In the torus mesh multiplication, the initial distribution of the numbers and the rearrangement that occurs before algorithm execution is crucial. Because of the importance as well as the complexity, it is very beneficial to animate the pre-execution rearrangement. The rearrangement can be accomplished by giving each processor its initial value and then having that processor pass to its neighbors along the communication channels as described by the algorithm until the rearrangement has concluded.

Mesh Transposition

The mesh transposition, as written about by Akl (1989) and Chaudhuri (1992), uses a simple mesh network as described in Section 2. Actually though, there are certain channels that are not used during execution and thus don't need to be drawn (as seen in Figure 10). The algorithm because easier to understand when only the communication channels that are used are included.

Also, in this particular algorithm, data is being passed from processor to processor until it reaches its destination processor. In an algorithm like this one, it is necessary to distinguish between a number than is still active and being passed and a number that has found its destination processor. These numbers can be distinguished by changing the color of the number when it reaches its destination processor. Also, in an algorithm like this one, when a particular processor has done all of its passing and is finished executing, it can change color as well.

Figure 10: The Mesh Transposition only requires a
few communication channels, and
the rest can be eliminated from the animation.



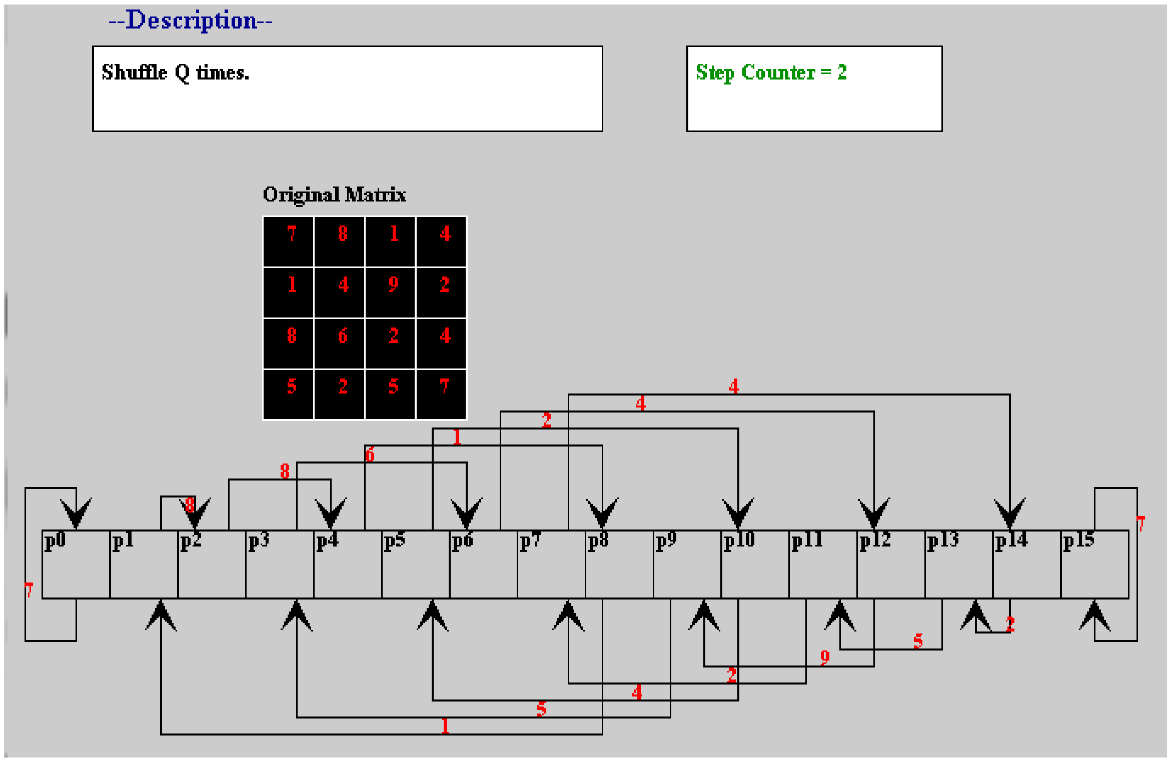
Perfect Shuffle Transposition

The perfect shuffle transposition, described in Akl (1989) and Chaudhuri (1992), is the animation I used during my small case study. The algorithm is can be thought of as very visual in nature, and therefore it can be animated quite elegantly. Initially, the animation should have the original matrix as well as an empty matrix displayed on the screen as seen in Figure 5. Now, in order to disperse the initial values to the proper processors, a very simple but effective trick can be played. First of all, the processors are supposed to be placed in 'perfect shuffle format', as displayed in Figure 3. Instead of just making the processors appear out of nowhere, the processors can grow out of the original matrix. Basically, the animations should actually draw two copies of the original matrix, one on top of the other, so that initially it appears that there is only one original matrix drawn. Then, for each element in row one, the second copy of the matrix should break apart and move below into perfect shuffle format. The process should then continue for each row until all of the initial values have been dispersed. This helps the student understand where the initial placement of values came from.

Next, it is necessary to explain how the processors are connected with the communication channels because this can be a confusing item with the perfect-shuffle network. Each communication channel should be drawn separately, and it along with it should be drawn a small text box explaining why this channel connects one processor to another. Also, just fitting all the communication channels on the screen can be a tricky process. The channels must be drawn very similar to Figure 3 so that they do not just become a bunch of confusing lines that cross each other. It is a good idea to put the first half on the top of the linear array and the second half of the channels below. Also, since this algorithm requires the number of processors to be a power of two, the matrix that is being transposed almost has to be a 4x4 matrix. The reason being, a 2x2 matrix is too simplistic and an 8x8 matrix becomes too confusing with all communication channels and processors required.

Finally, after the actual shuffling has occurred, the data is ready to return to the solution matrix. This can be accomplished by simply reversing the initial placement process. Every processor will slide back over the appropriate position in the solution matrix. What you will then end up with is the animation looking almost exactly as it did when it began. The only difference is that the solution matrix will now be full.

Figure 11: The perfect shuffle transposition animation in the process of shuffling.



--Description--

Shuffle Q times.

Step Counter = 2

Original Matrix

7	8	1	4
1	4	9	2
8	6	2	4
5	2	5	7

Perfect Shuffle Multiplication

The perfect shuffle multiplication described in Chaudhuri(1992) looks similar to Figure 3 and the perfect shuffle transposition, except it also includes communication channels connecting each processor to its neighbor on the right. The trickiest part of animating the shuffle multiplication, is fitting all the communication channels so that they do not look like just a bunch of crossing lines. The setup is going to be almost exactly the same as the perfects shuffle transposition, except that there needs to be spaces between each of the processors in order to show the connecting communication channels.

Tree Matrix-Vector Multplication

The tree matrix-vector multiplication, as described by Akl (1989), is a very interesting algorithm. In many ways, it is similar to the mesh multiplication and matrix-vector multiplication mentioned earlier in this section, except that it's in the form of a tree. There are input queues, just like a mesh multiplication, that will have to be drawn for the leaf nodes. Since the children are allowed to pass to their parents, each non-leaf processor will have two incoming communication channels that need to be drawn.

One thing to note is that the root of the tree performs a special task in the animation. As soon as the root calculates a final answer, it should send it to the proper position in the solution vector. This ends up being different than the other algorithms then, because the solution matrix needs to actually be filled as the algorithm is running instead of waiting until the algorithm terminates and then doing it all at once.

Linear Matrix-Vector Multiplication

The linear matrix-vector multiplication, as described by Akl (1989) and Chaudhuri (1992), requires input queues similar to the mesh multiplication described earlier in this section. Linear matrix-vector multiplication also requires one to deal with the issue of "dead" numbers like in the mesh multiplication. The linear matrix-vector multiplication actually ends up being just a simplified version of the mesh multiplication and therefore it is very easy to animate.

Summation

Through my studies, I have come to the conclusion that an instructor can greatly benefit from the usage of visualizations of parallel algorithms in the classroom. Simply attempting to draw them by hand can only lead to more confusion than intended. Sections 3-5 of this paper provided instructors with a variety of strategies for creating quality animations. Animations created using these strategies can then be used in lecture or outside of class for studying or quiz purposes. From my research and informal case study, I have concluded that the animation of parallel algorithms for dense matrix operations on interconnection-network SIMD computers can be very beneficial in a classroom setting and deserves more attention in the future.

Downloads

AnimalScript can be downloaded at http://www.animal.ahrgr.de/
JHAVÉ can be downloaded at http://csf11.acs.uwosh.edu/

References

Akl, S. G. (1989). *The Design and Analysis of Parallel Algorithms.* Englewood Cliffs, NJ: Prentice Hall.

Bergin, J., Brodile, K., Goldweber, M., Jimenez-Peris, R., Khuri, S., Patino-Martinez, M., McNally, M., Naps, T., Rodger, S., & Wilson, J. (June 1996). An overview of visualization: its use and design: Report of the Working Group on Visualization. *Proceedings of 1st Conference of Innovation and Technology in Computer Science Education,* 192-200.

Chaudhuri, P. (1992). *Parallel Algorithms: Design and Analysis.* Brunswick, Victoria: Prentice Hall.

Lester, B. P. (1993). *The Art of Parallel Programming.* Englewood Cliffs, NJ: Prentice Hall.

Mayer, Richard E., and Anderson, Richard B. (1991). Animations Need Narrations: An Experimental Test of a Dual-Coding Hypothesis. *Journal of Education Psychology, 83(4)*, 484-490.

Miller, B.P. (June 1993). What to draw? When to draw? An Essay on Parallel Program Visualization. *Journal of Parallel and Distributed Computing, 18,* 265-269.

Naps, T., Eagan, J., and Norton, L. (Mar. 2000). JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. *31st SIGCSE Technical Symposium on Computer Science Education*, 109-113.

Naps, T., and Chan, E. (Mar. 1999). Using Visualization to Teach Parallel Algorithms. *30th SIGCSE Technical Symposium on Computer Science Education*, 232-236.

Rößling, G., and Freisleben, B. (Feb. 2001). AnimalScript: an Extensible Scripting language for Algorithm Animation. *32nd SIGCSE Technical Symposium on Computer Science Education*, 70-74.

Rößling, G., and Freisleben, B. (Mar. 2000). Experiences in Using Animations in Introductory Computer Science Lectures. *31st SIGCSE Technical Symposium on Computer Science Education*, 134-138.

Rößling, G., and Naps, T. (June 2002). A Testbed for Pedagogical Requirements in Algorithm Visualizations. *Proceedings of 7th Conference of Innovation and Technology in Computer Science Education*, 96-100.

Stasko, J., Badre, A., and Lewis, C. (April 1993). Do Algorithm Animations Assist Learning? An Empirical Study and Analysis. *Amsterdam Netherlands, Proceedings of INTERCHI '93 Conference on Human Factors in Computing Systems*, 61-66.

Acknowledgements

I would like to acknowledge the assistance of Dr. Naps, my college professor and thesis advisor during my undergraduate career at the University of Wisconsin Oshkosh.