Algorithm Visualization through Animation and Role Plays

Jarmo Rantakokko
Uppsala University, Department of Information Technology
Uppsala, Sweden

`jarmo@it.uu.se`

1   Introduction

The motivation and expectations for using visualization of algorithms in the education is that the students will grasp and understand the abstract algorithms and concepts used in computer science courses easier, i.e. the algorithms will be more concrete for the students. Visualization is a useful technique for learning in any computer science course. In this paper, we focus on parallel computing and in particular on parallel sorting algorithms (Grama et al., 2003). Parallel computing is intrinsically more difficult than sequential computing, as it requires coordination of the parallel processes which can be very problematic to conceptualize. We want the students to understand how the processors work together in an algorithm and how they interact. As the students can *see* how the processors run simultaneously in parallel this will illustrate important concepts such as processor load balance, synchronization, and communication.

The use of pictures and visualizations as educational aids is accepted practice. Textbooks are filled with pictures and teachers often diagram concepts on the blackboard to assist an explanation. Animation includes one more dimension, explaining and illustrating the dynamic properties of an algorithm. Role plays go even further by actively involving the learners within the algorithm. Furthermore, animations and role plays bring a variety into teaching that is usually missing. There are different learning styles, i.e. Visual (V), Aural (A), Read/Write (R), and Kinesthetic (K) (Fleming, 2001). While traditional teaching and lecturing are more directed towards the Aural and Read/Write learners, animations are more directed towards the Visual learners and role plays towards Kinesthetic learners. Moreover, computer systems students have some preference to the visual learning style (Fleming, 2001).

In this paper, we discuss how students perceive these two visualization techniques, animations and role plays, for understanding abstract algorithms in parallel computing. We have used both techniques in the same course and then let students answer questions anonymously in writing. The students have also taken the VARK-test (Fleming, 2001) to correlate the answers with their learning style.

Computer animations are commonly used in traditional computer science courses, see e.g. (Naps et al., 2003a,b; Stasko et al., 1998), but little efforts have been made in parallel computing. Most visualization techniques are used by researchers for analyzing the parallel performance of their programs (Pancake, 1998). For educational purposes a group at the Georgia Institute of Technology led by John Stasko has developed the software package Polka for creating animations of parallel algorithms (Stasko et al., 1991, 1998). We have used this package to develop our animations. Another system that supports visualization of concurrent events is the Pavane system (Roman et al., 1992). A historically interesting note of using role plays is the experiment set up by Lewis F. Richardson in 1922 for computing a weather prediction (Richardson, 1922), long before the existence of parallel computers. Richardson simulates a parallel algorithm by letting individual workers do partial numerical calculations in parallel, as in todays parallel computers.
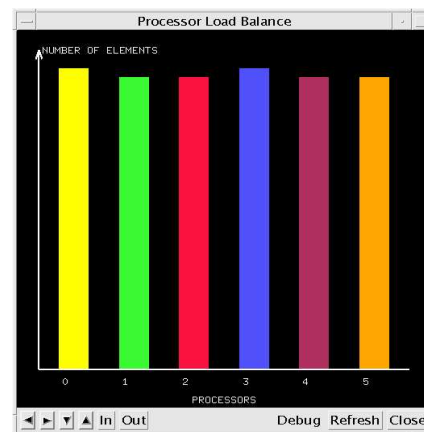
2   Computer Animations

We have developed two different animations for parallel sorting using the Polka package (Stasko et al., 1991, 1998). The animations are based on two parallelization strategies that are completely different and with different properties regarding communication, load balance, memory requirements, etc. The purpose is to let the students compare the algorithms, learn how they work and understand their inherent problems, i.e. the communication pattern, computation complexities and load balance. The students can work individually with the animations in a computer room exercise.

2.1 Parallel bucket sort

The first animation shows a parallel bucket sort algorithm for sorting a sequence of integer numbers in the interval [1, 100]. The numbers are represented with bars of corresponding height. We define *k* buckets by dividing the interval length uniformly into *k* subintervals, e.g. [1,25], [26,50], [51,75], [76,100] using four buckets on the interval [1,100]. The elements (numbers) are filtered in a sequential step into the equally spaced buckets. The buckets are then assigned to different processors and sorted in parallel with Quicksort. When running the animation, the user is prompted for the number of buckets, the number of processors, and how to distribute the buckets to processors. In addition, the animation is independent of the random number sequence. The user can choose from different random number sequences as input or even provide her own data file. In the animation we have two windows, one dynamic view of how the elements are sorted (Figure 1a), and one static view of the processor load balance (Figure 1b).
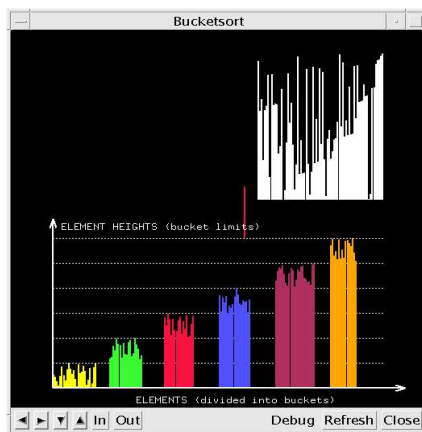
| (a) Physical sorting view | (b) Processor load balance |

Figure 1: The *bucketsort* approach for parallel sorting.

In a first step, the elements are moved from a single stack to the different buckets. The buckets are coded with different colors depending on which processors they belong. Next, the elements are sorted simultaneously in the different buckets creating an illusion of parallelism. The processor with the highest work load will continue sorting the longest time, illustrating the effect of the load imbalance. Different random number sequences will give different work loads in the buckets. The workload can then be balanced with, e.g., using more buckets than processors and distributing the buckets cyclicly to the processors or by providing a file with pre-described non-uniform bucket widths.
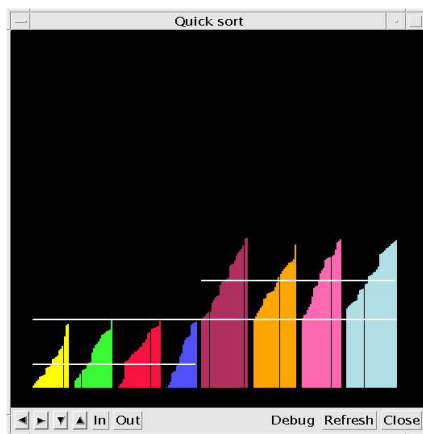
2.2 Parallel Quicksort

The second animation illustrates a memory efficient and fully parallel algorithm. The data is kept distributed all the time and there are no large serial sections as in the filtering operation above. In this approach, we divide the elements equally among the processors and perform a local sort within each processor. Then we exchange data pairwise between processors to get a global sorting order. The algorithm is outlined in Figure 2.

In this animation, the user can also choose the number of processors and the random number sequence as input. Here, the processor load balance depends on how the pivot elements are chosen in step *3.1*, Figure 2. There are different pivot selections strategies available for the user to choose as input. The animation supports two different views of the algorithm. The dynamic view shows how
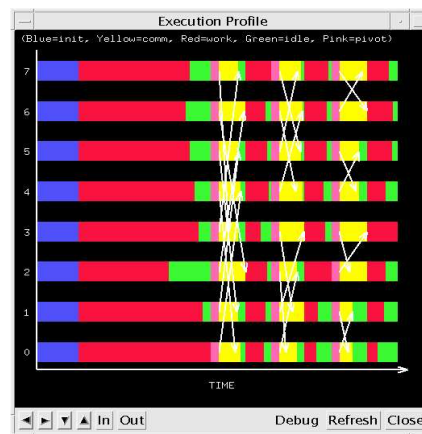
```
Parallel quick-sort

   1.  Divide the data into p equal parts, one per processor.
   2.  Sort the data locally for each processor.
   3.  Perform global sort.
      3.1 Select pivot element within each processor set.
      3.2 Locally in each processor, divide the data into two
          sets according to the pivot ('≤' or '>').
      3.3 Split the processors into two groups and exchange data
          pairwise between them so that all processors in one group
          get data less than the pivot and the others get data larger
          than the pivot.
      3.4 Merge the two lists in each processor into one sorted list.
   4.  Repeat 3.1 – 3.4 recursively for each half (log₂ p steps).
```

Figure 2: Parallel Quicksort. The elements are divided equally among the processors and sorted locally. Then, in a number of steps, the processors split their data into two parts according to a pivot and exchange data pairwise with a merging step to get a global sorting order.

(a) Snapshot during global sorting   (b) Execution profile

Figure 3: Views of the parallel quicksort animation.

the elements are physically sorted and moved between different processors (Figure 3a). All parallel activities are animated simultaneously, creating an illusion of parallelism. In the second view, an execution profile is grown simultaneously as the animation continues (Figure 3b). Here, different activities are color coded in bars for respective processor. Arrows represent the communication. The arrows are drawn from the sender to the receiver processor and their stopping point indicates how much data is communicated. From the execution profile it is possible to extract both computation and communication load imbalances.

3   Role Plays

Role plays are well suited for visualization of parallel algorithms. Each student can take the role of a processor in the algorithm. All bottlenecks, such as communication and load imbalance, in an algorithm become very obvious and concrete. The role plays can be performed within class.

Again, we have used different sorting algorithms for demonstration. The first algorithm is a pipelined version of *bubblesort*. The students are given different parts of an unsorted deck of cards. They lay out their cards on the table. The first student starts its first iteration moving the largest card by pairwise compare-exchange to her right while the other are idle. When she comes to the end of

her cards in the first iteration she must compare and exchange cards with her neighbor and can then start the next iteration. The other students proceed in the same way but have to wait awhile in the first iteration before they can start. Similarly, the processors/students to the right will finish sorting first and will then be idle. This algorithm requires frequent communication and synchronization and it also has a large load imbalance.

Next, we let the students perform the *odd-even transposition* algorithm. Here the students are also given parts of the deck of cards but now they first sort their own cards (locally). Then, they exchange all their cards alternately with their left and right neighbor. In each step, one student in a pair merges his cards with his neighbor's cards and gives back half of the cards, keeping the other half. The algorithm proceeds in *p*-steps (where p is the number of processors/students). This algorithm is much faster and requires less communication than the pipelined bubblesort algorithm. Still, there is some load imbalance, with half of the processors doing the merging while the other half of the processors are idle.

Finally, we let the students simulate the parallel *quick-sort* algorithm, as shown in Figure 2. This algorithm minimizes the communication compared to the two previous ones and terminates in $log_2(p)$ steps. Any load imbalance comes obvious from bad pivot choices giving some students more cards than the others.

4   Students

The target course, *Programming of parallel computers*, is a C-level course given in the third year. The students come from different programs, Master of Science, Computer Science, Natural Science, and exchange students. The students are mostly male (85%). The pre-requirements for the course are that they have at least taken two programming courses. This is their first course in concurrent programming techniques. The course is given with traditional lectures and has mandatory computer room exercises.

5   Results

In addition to a course evaluation we have asked the students to fill out a survey with the following questions: Grade 1-5 (where 1=not useful and 5=very useful) traditional lectures (L), textbook (B), animations (A) and role plays (R) with respect to:

1. Understanding parallelism and algorithms

2. Motivation for learning

3. Help for programming assignments

4. Which combination of teaching methods would you benefit most of, e.g. (L)+(B)+(A)

The results from the survey are summarized in Figure 4. The scores for the fourth question are calculated by counting the number of (L), (B), (A), and (R), respectively, then dividing these numbers by the number of students and finally multiplying with five to normalize with the other scores.

The results show that traditional lectures outperform the other teaching methods in all aspects. But it is interesting to see that the students feel that both animations and role plays give better understanding for the algorithms than the textbook and that the students prefer animations before the textbook for learning (in combination with lectures). Comparing the two visualization techniques clearly shows that computer animations are experienced by the students as having higher impact on learning than role plays. The results of the survey depend very much on factors such as quality of lectures, book, animations, enthusiasm of lecturer, and preference to learning style.

The students also took a VARK-test (Fleming, 2001). The results from the test, Figure 5, show that this particular group of students had some preference to the Read/Write and Kinesthetic learning styles. This does not correlate with the results of the questionnaire where the textbook got low scores while lectures got high scores.
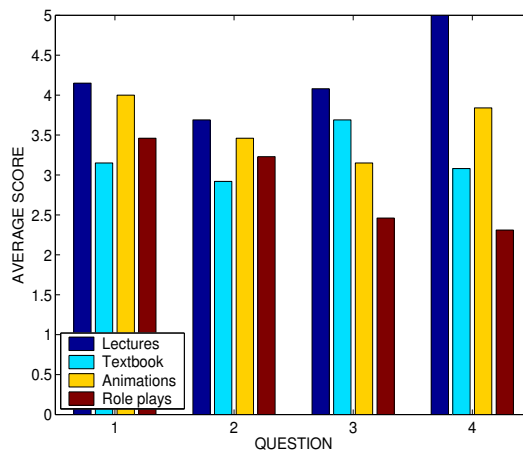
Figure 4: Average scores from the survey, 13 students answered the questionnaire.
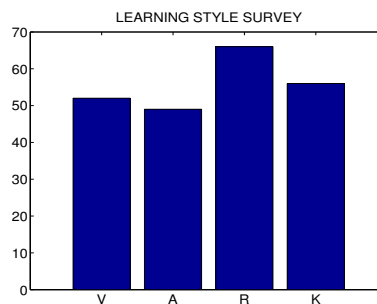
Figure 5: Total scores from the VARK-test. 15 students answered the questionnaire.

The course evaluation, which the students also answered anonymously, was very positive giving an average score 4.3 (out of 5.0) for the course as whole, 4.6 for the lectures and 4.1 for the laboratory exercises including the animations. The textbook got an average score 3.6 which can still be considered good.

6   Discussion

We have used both computer animations and student role plays to visualize abstract algorithms in parallel programming. Both techniques increase and facilitate learning and understanding of the specific algorithms. They also illustrate general concepts, such as load imbalance, synchronization and communication, in a very concrete way. In this particular study, they are experienced by the students as giving better understanding for the parallel algorithms than the textbook.

There are some fundamental differences between these two visualization techniques. Computer animations allow the students to work in their own pace and let them re-run the animations as often as they want to. They can also experiment with different input parameters and data testing different scenarios and behavior of the algorithms. Role plays on the other hand make the inherent concepts more concrete and real. For example, a load imbalance forces one participant to work more than the others. Role plays do not require any technical equipment or software development, maintenance, etc. Developing computer animations requires a lot of work and it can be difficult and time consuming to learn how to do this (Naps et al., 2003a). A problem with role plays is that they can be difficult to perform in a large group of students. It can be impossible to let all participate and then some of the students may not be able to follow or feel left out and be unengaged. Also, while role plays promote understanding of details it can on the other hand be difficult to see the big picture of the algorithms.

In the student survey, animations got better scores than role plays. But the best results were given to the lectures. One explanation is that the impact of the other factors, e.g. quality and enthusiasm,

has higher impact than the individual learning styles. Moreover, the lectures are a mixture of different teaching methods, including discussions, writings and visualization with pictures on the blackboard.

References

Niel Fleming. Vark, a guide to learning styles. `http://www.vark-learn.com/`, 2001.

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Introduction to parallel computing, second edition. *Pearson Education Limited*, 2003.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudoplh Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and Angel Velazquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *inroads 35:2*, pages 131–152, 2003a.

Thomas L. Naps, Guido Rößling, Jay Andersson, Stephen Cooper, Wanda Dann, Rudolph Fleischer, Boris Koldehofe, Ari Korhonen, Marja Kuittinen, Charles Leska, Lauri Malmi, Myles McNally, Jarmo Rantakokko, and Rockford J. Ross. Evaluating the Educational Impact of Visualization. *inroads 35:4*, pages 124–146, 2003b.

Cherri M. Pancake. Exploiting visualization and direct manipulation to make parallel tools more communicative. *Lecture Notes in Computer Science 1541*, 1998.

Lewis Richardson. Weather prediction by numerical process. *Cambridge University*, 1922.

Gruia-Catalin Roman, Kenneth C. Cox, Donald Wilcox, and Jerome Y. Plune.G. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing, Vol 3, No 1*, pages 161–193, 1992.

John Stasko, William F Appelbe, and Eileen Kraemer. Utilizing program visualization techniques to aid parallel and distributed program development. *Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical report GIT-GVU-91/08*, 1991.

John Stasko, John Domingue, Marc H Brown, and Blaine A Price (editors). Software visualization: Programming as a multimedia experience. *MIT Press, Cambridge, MA*, 1998.