VATPA: A Simulation Environment for Message-Passing Concurrent Systems

Hui Liu, Maung T. Nyeu, David Y. Y. Yun and Woei Lin
Department of Electrical Engineering
University of Hawaii at Manoa
Honolulu, Hawaii 96822

Abstract

Low-cost, high performance personal workstations with high resolution display have been touted as vehicles for visual simulation. Creation and display of continuous movement of icons and objects provide realistic animation. We are currently involved in a visual animation tool for parallel algorithms (VATPA) for visualizing parallel systems. A parallel system is made up of many concurrent entities that exchange information by passing messages. VATPA creates graphic snapshots and movies to the systems' actions. There are two major subsystems in VATPA: (1) a programming subsystem is concerned with program coding and execution of computation and communication events; (2) an animation subsystem deals with visualizing images that correspond to simulation events. The animation subsystem provides a variety of options which allow users to create and tailor animation scenes to suit their needs. The behavior of process nodes at the run time is represented by color-code and the interprocesses message-passing is indicated by the moving message objects. The combination of color-coded processes and moving messages offers an visualizing environment where users can observe a variety of system characteristics which are difficult to see by tracing textual programs. With the animation tool, users can easily identify computational bottlenecks and communication-related programming errors, and tune the system to achieve higher performance.

1. Introduction

Analyzing parallel algorithms is a considerably difficult job [1-10]. It is drastically different from analyzing sequential algorithms. Execution of parallel algorithms typically involves a multitude of concurrent activities that are interleaved in arbitrary ways. Tracing these individual activities can easily overwhelm human abilities of understanding and reasoning the run-time algorithm behavior. Visualization has long been used as a communication tool for processing large quantities of image information in parallel, detecting and tracking complex patterns at incredible speed. In this paper, we present an animation tool for visualizing the complex, dynamic behavior of parallel algorithms. The key idea behind this tool design is to abstract run-time activities and turn them into graphical representations that move and vary in location, size and color. With proper abstractions, it establishes a technical foundation for monitoring, analyzing and improving complex parallel algorithms. It represents animated views in an informative and coherent manner. There are two unique attributes that distinguish our animation tool from others: (1) It provides a general simulation environment for visualizing parallel systems with arbitrary program structures and sizes; and (2) It allows the visualization for computation status of parallel processes and their run-time interactions.

2. VATPA Organization

VATPA provides a general simulation environment for parallel algorithms with arbitrary configurations. It is based on a general model that can adapt to various program states. The model defines an animation framework that (1) abstracts operations and events in an algorithm, (2) provides animation actions to simulate those operations and events, and (3) maps the abstract operations to the corresponding animation actions. The proposed model is a message-passing paradigm, which offers a simple, consistent way to describe dynamic interactions among concurrent actions in animated views. Using this tool, users do not need to write programs for calling graphics subroutines. Nor do users need to employ snapshot image updates using underlying static graphics packages. This enables users to concentrate on observing essential events and actions at a highly detailed level and on a continuous basis.

VATPA is composed of two major subsystems: the programming subsystem and the animation subsystem. Figure 1 shows the block diagram of the organization of the VATPA. These two subsystems are described as follows.

2.1 Programming Subsystem

The programming subsystem is concerned with the implementation of an algorithm. It is based on the Communicating Sequential Process (CSP) model [10]
that has been widely considered as a natural way of presenting parallel algorithms. In the CSP model, a parallel algorithm is considered as a collection of processes that run in parallel and exchange data asynchronously for information sharing and coordination. The programming subsystem provides a programming language, called POP [11], for algorithm implementation. Succinct but powerful POP control structures allow users to easily describe computation and communication steps of parallel activities in the forms of channel and process.

![Figure 1. Organization of the VATPA.](image)

Parallel processes use channels to exchange messages. A channel is a two-way point-to-point link connecting two communicating processes. Or it can be viewed as a shared data structure that comprises a set of variables of various data types. Two processes communicate with each other by accessing these shared variables yet in a restricted way. A channel behaves as a read-only element to a receiving process. The transmitter can write only when the channel is empty, while the receiver can read only when the channel is full. In other words, writing fills the channel, and reading empties it. A process is the fundamental working element of sequential work. The process structure includes a set of local variable declarations and a program body. The set of local variables are used to store results due to local computations within the scope of a process. The body contains a group of statements and handles one prescribed activity. Functionally, a process is a group of statements that share the same context. These statements will be interpreted in sequence. They mainly fall into two classes: computation statements and communication primitives. Currently, the programming subsystem provides options for coding the computation in C programming language. Communication primitives allows processes to perform unidirectional point-to-point communication for transmitting data messages. Two types of primitives are used to read data from and write data into channels. Their syntax are of the forms

- send(channel identifier.variable1, variable2)
- receive(variable1, channel identifier. variable2).

For a "send" primitive, the value of variable2 will be stored into variable1 of channel. Similarly, for a "receive" primitive, the value of variable2 will be read from channel and stored into local storage variable1.

POP extends the index scheme, commonly used for structured data of array types, to the process level. An aggregate of parallel processes can be efficiently declared through a single indexed program instance. The use of index technique allows users to create a set of parallel processes simply with one single program instance. In POP, specifications of a set of indexed process require a minor augmentation in process header. This can be done merely by appending an index range expression to a process header. An index range expression, delimited by a pair of angle brackets, mainly contains a list of dimension parameters each followed by an index range. For instance, a process header

```
process mesh < x = 4,< y = 4 >
```

creates 16 processes: mesh[0][0], mesh[0][1], ..., mesh[3][3]. Although indexed processes share one version of program code, their run-time environments are physically separate from one another at the execution time. Within the program body of indexed processes, dimension parameters are used to distinguish these indexed processes from each other so that they can perform different operations due to location discrepancy. As an example, suppose

if (x == 0 && y == 0) then statement;

is contained in the program body of the indexed mesh processes. Among the 16 processes, only mesh[0][0] will interpret the statement. Channel declarations can also be indexed. However, the index scheme for channels is more complex than that for processes, because of the additional need of specifying the associations between channels and processes. The header of an index channel has a syntactic structure...
channel identifier < index ranges >: from source process [dimension specifiers] to destination process [dimension specifiers].

Index ranges have a form identical to the ones for an indexed process, i.e., each dimension specifier corresponds to one of the process dimensions. A dimension specifier could be a constant or a simple arithmetic expression with some dimension parameters. Associations of indexed channels and processes are determined by the use of common dimension parameters in index ranges and dimension specifiers. For example, the declarations

\[ hbus(x) = 2 > cy = 2 \times \text{mesh}(x)[y] \to \text{mesh}(x)[y+1]; \]
\[ vbus(x) = 2 > cy = 3 \times \text{mesh}(x)[y] \to \text{mesh}(x+1)[y]; \]

create 24 channels running horizontally and vertically to interconnect the 16 processes of the 4x4 mesh.

The programming subsystem contains a simulation environment that uses concurrency to emulate the parallel execution defined in a POP program. In the simulation environment, processes are physically executed and their execution threads are properly interleaved as if they were run on a multiprocessor. The programming subsystem provides the animation subsystem with three files as interface: process file, channel file and log file. For each algorithm, the animation subsystem reads these two files specifically describe process states and their interconnections for the algorithm. A log file stores event information of process execution. During the algorithm simulation, it records a sequence of events for each process, such as efficiency and utilization, and converts them into graphics primitives for display. These four modules create four classes of graphical objects: nodes, links, messages and meter bars.

(1) Nodes: A node is a graphical object that represents a collection of operations to be performed on a process. Throughout the frames of an animation, a node undergoes changes in color. Different colors are used to indicate various process states during execution. Typical process states considered in an animation are: blocked, active, and terminated. When taking a snapshot, a user can quickly comprehend the algorithm status at that particular time by examining individual nodes. Color is also useful in reporting some essential statistical measurements such as relative process load. For example, different colors are used to represent heavy load, medium load and light load. By observing differences in color, users can identify communication bottlenecks and overloaded nodes to improve the performance by load balancing.

(2) Links: A link provides a communication channel between two processes. When two nodes are connected by a link, they can pass some messages during the execution time. In other words, messages can only be transferred along links. A link typically designates a two-dimensional route in an abstract real-coordinate system.

(3) Messages: A message denotes a data object that moves from a source node to a destination node along a specified link. Each transfer gives an animation action. A message has a starting time to indicate when the message first appears on the screen. The life-span of the message is determined by the speed of the link along which it traverses and the amount of data it carries.

(4) Meter bars: A meter bar is a time-varying object that quantifies some measurement of interest. It is a one-dimensional strip that grows or decreases at one end. A meter is typically used for indicating program elapsed time or run-time performance levels.

The graphics editor is a module that interprets the user's editing commands for displaying, manipulating and restructuring the four classes of graphical objects mentioned above. It selects and scales objects according to editing specifications. The scene generator is a
collection of low-level graphics function calls to convert those abstract objects into visible images displayed on the screen. It is also responsible for displaying the scene synthesized by the graphics editor on the screen.

3. VATPA Implementation and User Interface

3.1. Operational Scenarios

VATPA uses a single high-level interface to maintain a consistent work environment. User interaction relies on mouse movement for menu selection. Upon entering the animation subsystem, a user sees a window with three functional areas: the display, the meters, and the menu. Figure 2(a) shows the layout of the three areas. The display shows algorithm animation scenes. The meters report elapsed time, average utilization and efficiency throughout the animation. The menu contains a set of options for piloting the animation and modifying the program structure. They control initiation and termination of an animation on the display. An animation session usually begins with two menu options for displaying nodes and channels. Upon request, the subsystem displays an initial structure of the program. The menu provides restructure options for changing the program structure. Users can use the mouse to rearrange nodes in a way they prefer. When a program has more than a hundred nodes, the display may become too crowded to see individual nodes and their interactions. In this case, users can apply a zoom option to magnify the initial program structure and apply scroll option to view other parts of the program structure. Combinations of the zoom, restructure and scroll options provide versatile ways of reconfiguring a program structure. Once the program structure on the display is determined, the animation is ready to run. In the animation phase, messages are displayed as white circles moving from their respective source nodes to destination nodes. In the mean time, nodes on the display can be chosen to show color-coded process status or utilization. There are three meters for reporting run-time measurements. The time meter indicates the current time of program execution. The other two meters show measurements of the overall utilization and efficiency of an algorithm.

3.2. Interface between Algorithm and Animation

The programming subsystem generates three files which are used to map a parallel system into animation scenes. A process file contains a list of processes' name. A channel file contains descriptions of channels and defines their associations with end nodes. A log file consists of sequential records which correspond to events during the execution of a parallel system. Main fields of the log file are listed below:

(1) Time: clock time when an event occurs;
(2) Node: index of a process node on which an event occurs;
(3) Status: event types including sending, receiving, computing, phase entering, waiting for channel, waiting for message, and terminated;
(4) March: communication step number;
(5) Phase: phase number of an event; and
(6) Channel: index of a channel which is involved in a communication-related event.

3.3. Modules of Animation Subsystems

The program structure module maintains a dynamic display file to record process nodes and channels which are displayed on the screen. It contains such detailed information as current positions, colors and sizes of nodes channels and messages. VATPA organizes process nodes in the display window by assigning each node a real-valued coordinate pair \((x, y)\), a radius and color. At the beginning of an animation session, it arranges nodes into a two-dimensional array. Then it updates the display dynamically through the animation. The message flow module is responsible for calculating message positions for visualizing message flows. A message flow designates a route from the source node to the destination node in a real-coordinate system. In the current version of VATPA, three consecutive message images are created for each message movement. This requires four artificial events:

(a) Draw the first message image;
(b) Erase the first image, draw the second one;
(c) Erase the second image, draw the third one;
(d) Erase the third image.

By sequentially operating the four events, VATPA creates an apparent motion effect to human visual system. After the four events are generated, they are inserted into the original event sequence provided by the log file. This allows message movements to take place concurrently. Composition is crucial to animation in which more than one object changes. When multiple message movements are composed in this manner, their respective actions in each frame appear perceptually simultaneous.

We use color to encode process status. There are two coding schemes. The first one is an activity coding scheme. It uses different colors to represent different activity types. We consider seven types: sending message, receiving message, in computing, phase entry, waiting for channel, waiting for message, and terminated. However, we divide these seven types into three groups and simply use three colors for representation because of human's visual capability. Experiments show that too detailed information often makes users easily tired. The three groups are (1) active (sending, receiving, in computing, phase entry), (2) blocked (waiting for channel or message); and (3)
terminated. The second scheme is a phase coding scheme. When programming an algorithm, a user may divide it into a different number of phases. These phases are coded with different colors. Each phase entry is considered as an event. When a node enters a different phase, the process status module notifies the scene generator module to change the color for the node.

The statistics module visualizes performance of individual processes as well as the average performance of an algorithm at the run time. An accumulator is associated with each process throughout an animation. It computes process busy time, utilization and speedup. Color is used to encode utilization level of process nodes at the execution time. We categorize node utilization into 5 levels based on the same reason mentioned above. They are: over 90%, 90-70%, 70-50%, 50-30% and below 30%. Two global accumulators are used to calculate utilization and efficiency and they are instantly displayed in the form of time-varying meter bars. In this way, the animation scene generator maps abstract objects into graphical objects on the screen and provides an animated display of parallel program execution.

4. Applications of VATPA

VATPA has been successfully used by some students for implementing projects in the graduate course of parallel processing. VATPA offers the following but limited features to the end users.

4.1. Usefulness

VATPA offers user a simple and easy way of coding a wide range of parallel algorithms with minimum efforts. The performance statistics generated by the simulation tool not only help the user to observe the performance of a parallel program but also to understand how the performance changes with respect to different parameters such as, number of processors, communication topology, granularity and load balance. The moving messages show the interactions and data flow. In a parallel algorithm, communication costs may dominate the program performance. By comparing the busy time and computation time, the effect of the communication costs can be revealed. In a nutshell, VATPA provides a basic, yet versatile, tool for fast visualization and evaluation of the performance of parallel algorithms.

4.2. Examples and Experiences

Here we present a parallel implementation of Chelosky algorithm of matrix decomposition to illustrate the use of VATPA as an animation tool. Figure 2(a) -- (d) show four snapshots of the animation of a 10x10 matrix decomposition. Note that there is a wavefront during the execution of Chelosky algorithm, which consists of three bands corresponding to the three phases of square root, divide and elimination operations, respectively. The width of the wavefront gradually increases as it passes from the top left corner to the bottom right corner of the array. This is difficult to observe analytically. But it can be easily seen through the animation.

VATPA has been available for users for over a period of one year. Many interesting and complex algorithms have been implemented using VATPA. It has been used to implement parallel algorithms with different communication topologies, e.g., mesh, binary tree, hypercube, etc. VATPA has showed its use through a powerful, user friendly environment. Users have suggested for more debugging facilities. VATPA is still being continuously modified with the additional new and useful features.

5. Conclusion

VATPA is intended to be an animation tool for visualizing the dynamic behavior and performance of parallel algorithms. It offers users a simple and easy way to simulate parallel algorithms with arbitrary configurations. It has been successfully used by a few of graduate students for their project implementations in the graduate course of parallel processing at University of Hawaii. The programming and animation models are based on the Communicating Sequential Processes (CSP) concept that considers a parallel algorithm as many concurrent entities that exchange information by passing messages. This model captures many key features for characterizing parallel algorithms, such as communication structure. Color-coded nodes along with moving messages offer a lively environment for users to observe many high-level algorithm characteristics, e.g. varying-width wavefront of the Cholesky's decomposition algorithm. Computational bottlenecks and communication-related programming errors can be easily pinpointed by observing the animation scene. These features along with dynamic performance meters provide users with a powerful tool to design parallel algorithms, to measure performance, to debug errors and to improve performance.

6. References


Figure 2. Snapshots of Chelosky Algorithm Animation