What to Draw? When to Draw?
An Essay on Parallel Program Visualization

Barton P. Miller
bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

1. INTRODUCTION

Parallel programs are difficult to understand and the research community is determined to provide tools to help improve that understanding. As a result, there is intense interest in using visualization to understand the execution of parallel programs. But visualization is like the surgeon's knife: properly used it will quickly cut to the essentials; improper use brings colorful but disastrous results. The results from a visualization are almost always visually appealing, but I claim that they only occasionally are useful in debugging a program or improving its performance.

In this essay, I share opinions and experiences that I have formed while developing tools for understanding execution of parallel programs. I try to codify some of the implicit rules that have guided my research efforts. As a result, I will criticize some approaches to visualization, but I will not direct these criticisms at specific projects or tools (except my own). The intent of this essay is to provide a basis for evaluating our work and to encourage dialogue within the parallel programming tool community.

2. WHY IS VISUALIZATION IMPORTANT?

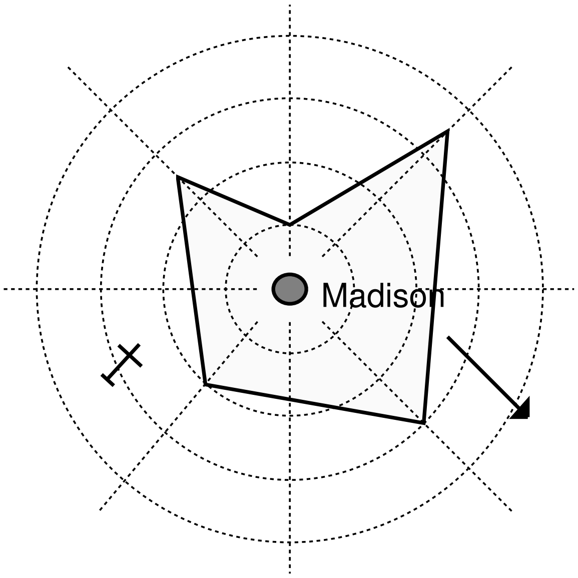
Almost everyone seems to agree that visualization is an important tool in understanding complex processes. Physical scientists have been using visualization successfully for several years. In the classroom, computer scientists have been using it to teach such subjects as data structures. As motivation, I present two examples from my experiences where pictures were worth much more than a thousand words (or even a megabyte).

The first example comes from the world of aviation. As a pilot flying in stormy weather, you will occasionally hear the air traffic controllers broadcast a message such as:

This work was supported in part by National Science Foundation grants CCR-8815928 and CCR-9100968, and Office of Naval Research grant N00014-89-J-1222.

There is an area of intense thunder showers covering an area bounded by
315 degrees at 25 miles from Madison to
360 degrees at 10 miles to
045 degrees at 35 miles to
135 degrees at 30 miles to
225 degrees at 20 miles,
moving southeast at 30 mph.

This message is obviously of great potential interest and you quickly want to decide if you need to point your aircraft in a different direction. After you hastily transcribed the message, you find your map and locate your position and direction of flight, then attempt to outline the area described in the broadcast. With digital telemetry, weather radar, and moving-map displays, instead you might see:



It only takes a few moments to evaluate your situation and take action.

A second example comes from a programming job that I had many years ago. We were building computer controls for a facility such as a refinery. Microprocessor-based graphic display computers were the main user interface. We had a bug in the system that appeared as random noise in the bottom right corner of the screen and worked its way across the screen. It continued across the next line, and the next, until it filled the entire screen. Any of our programming team that watched this view quickly exclaimed, *Hey, that's a stack overflow!* The visual manifestation was compelling.

Most experienced programmers have come across some similar circumstance. In both of the examples that I presented, the visual information was used to guide the pilot or programmer in their task. They were able to more quickly and easily do their job by using visual information.

So why are tool builders for parallel programs having such a difficult time developing useful visualizations? A more cynical person than I would ask "are we really are trying to produce useful displays or are we just trying to be colorful?"

3. WHY IS VISUALIZATION DIFFICULT?

Drawing useful pictures is difficult. It is much easier to draw a visually attractive picture than it is to draw a useful one. You know that you have goofed when you hear: that looks really neat, what does it mean?

There are two basic problems in visualization. First you have to have a model of the system who's behavior you are trying to understand. Second (and only after the first problem is solved), you can then try to invent good ways to illustrate this model. Physical scientists have a physical reality behind their work. Even if the physical behavior is complex, they can still use their models to appeal to our intuition on how these behaviors should work. These scientists only have to worry about the second problem (how to illustrate the behavior).

Visualization of parallel programs is difficult because we cannot appeal to a physical model. In computer science, our constructs are mostly artificial (e.g., what does a process look like?). Even when we precisely specify our model (for example, the definition of a process), it may not appeal to our intuition. Unlike physical properties that (presumably) have a single physical reality, computer constructs can be realized in many forms. So a picture that appeals to one person's mental model may have little in common with the models held by others.

Computer scientists try to use some physical properties in visualizing their programs, e.g., the interconnection architecture of a message-based multicomputer. While this has some benefit, most programming tools try to abstract away from the physical model, gaining simplicity and portability. This level of programming abstraction limits the usefulness of such models.

Given a model, we still have to design a picture to represent it. Computer scientists and physical scientists deal with multivariate, time-varying data that has complex internal correlations. Both groups are looking for patterns that will describe essential behaviors at a level of abstraction that will neither obscure with detail nor obscure by being vague. If we can get past the first stage and find an intuitive model with wide appeal, then we might be able to share techniques with other fields.

4. HOW CAN WE EVALUATE A VISUALIZATION?

This section addresses two questions. First, what characterizes a useful program visualization? These characterizations can help to evaluate current techniques and may help in the design of future ones. Second, how do we evaluate new visualizations? Evaluation based on visual appeal is not satisfactory; we must be able to determine usefulness.

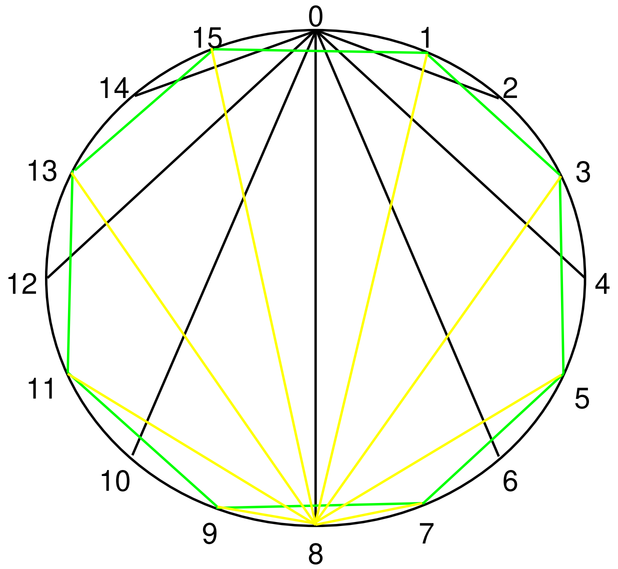
4.1. What are criteria for a good visualization?

Visualizations should guide, not rationalize:

“Guide” means that the visualization leads you to discover things that you did not already know. “Rationalize” means that it lets you illustrate things that you already know. If a visualization is used to demonstrate a system that is already understood (e.g., for marketing, sales, or teaching), then a rationalizing display is sufficient. If the visualization is meant to help a programmer discover new things about their program, it must guide. In research presentations and papers, it is important to make this distinction.

A major difference between guiding and rationalizing is the organization of the information. If the visualization tool can provide interesting and informative displays without detailed control by the user, it has potential to guide the user. If the user has to give the tool detailed instructions on how to select and organize the data, then the user may have to already know how the program is working to get a display on how the program is working.

An example of that was presented at a workshop several years ago. A tool was developed that had a circular display. Process ID’s (integers) were arranged around the circumference of the circle. Colored lines connected the processes to represent communication levels; red showed intense communication between a pair, violet showed little communication. In this presentation, the speaker showed a figure with a clear geometric pattern in the communication.



It was impressive that information about the dynamic structure of the program was readily apparent from the picture. The speaker was then asked: "how did you know how to arrange the processes around the edge to produce a useful picture?" The answer was something like: "well, I knew something about the structure of the program, then played with it for a while until I got something that I recognized." The disappointing part of this answer was that it seemed that they were using the visualization to show things that they already knew. This example is not an isolated one; it has been repeated at many meetings.

Visualizations should be appropriate to your programming model:

If a programmer uses procedures, explicit process creation (e.g., fork) and explicit synchronization (e.g., semaphores), then your displays can be in terms of these operations. Metrics such as blocking time for a process and context switching rates make sense for these objects. If the programmer is using a language such as Fortran-D with automatically parallelized loops, explicit data distribution and alignment, then results based on processes would be meaningless. It would be more useful to present the data in terms of blocking time for an array update, based how much time was spent moving specific parts of other arrays (a data view, rather than a control view).

Scalability is crucial:

Visualizations should have few inherent limits. A picture that works well for up to 16 processors may be sufficient for the current system, but this can change quickly with new versions of the software or hardware. Any visualization tool that is based on processes, threads, or procedures will have to deal with hundreds, even thousands, of objects. A display might deal with scale by inherently being able to organize large amounts of data. This type of display is hard to find. Alternatively, a display might deal with scale through alternate views; the user starts with a course level of detail, then adds new displays with finer details for selected parts of the program.

In IPS-2, we use a display that is inherently scalable (to very large numbers of objects), easy to understand, and easy to draw. Unfortunately, it is not a picture but a table! Our tables that present procedure profile data sort the data by having the procedures with the largest values at the top and the rest in descending order. Given thousands of procedures, you still have the most significant ones at the top. If you want to look further down, a single scroll bar is sufficient.

If you want to display a profile table graphically, you might encode this data in a pie chart. But this display can quickly get cluttered with small wedges and too many labels. A histogram is another reasonable alternative, but this display is also subject to clutter with a large number of objects.

Color should inform, not entertain:

Color should either (1) increase the information density, (2) accent important cases, or (3) aide in identification. In our performance tool (IPS-2), we have a display that we call the “time histogram”. This display plots the values of one or more metrics over a time period (often called a “strip plot”). Our experience has shown that a user can distinguish up to three curves in black & white (using different line styles) and up to six or seven curves in color.

IPS-2 also displays data for each procedure in a sorted profile table. We have several different ways of calculating profiles, but the basic table display is the same. Even though each different kind of profile is labeled with a different banner label, it is easy to misread these labels. We give each different kind of profile table a different banner color, allowing you to quickly identify the type of information that you are using. Color can also be useful to correlate related information contained in different displays (e.g., by giving it the same color or shape).

Color is appealing, but should be used only when it adds information. Avoid (as described in the New Hacker’s Dictionary) pictures that look like “angry fruit salad”.

A visualization should be interactive:

A visualization that tries to provide all information in one view can be overwhelming. Parallel programs are complex and there are many aspects to the behavior of the program. The visualization should help you decide what information you need to see next. The complexity of understanding a visualization should be less than the complexity of the program.

There are three dimensions over which you can refine your visualization. First, you should be able to refine your view by looking at a different part of the program or system. Second, you should be able to change your level of detail, such as moving from the procedure level down to basic blocks or up to processes. Third, for a given part of the program and level of detail, you should be able to select different types of performance data.

You might be able to display data from across one of these dimensions in a single visualization, for example, many performance metrics for a specific set of procedures. But trying to vary more dimensions in a single visualization can result in unmanageable detail.

Visualizations should provide meaningful labels:

A visualization should provide a means for identifying the program or system objects that are being described. The simplest approach is to identify objects by explicitly labeling them in the display. In this way, both performance data and identification are simultaneously visible. Alternatively, you can provide an interactive facility that lets the user use a mouse to request the presentation of labeling information.

Default visualizations should provide useful information:

The programmer should not require sophisticated knowledge nor have to select from a myriad of parameters to create a display. This rule follows from the “guide, not rationalize” rule. If you give users the opportunity to carefully select many interesting views, they can as easily select a confusing one. A visualization tool should provide a set of useful default views. These views should, in most cases, be sufficient for programmers to find their problems.

Avoid the “watchmaker’s fascination”:

When you remove the back cover from a mechanical watch, you see a fascinating and elegant collection of gears, levers, jewels, and springs. There is an almost hypnotic pleasure in watching the action of the mechanism.

Unfortunately, it is difficult (though not impossible) to tell the time by looking at the gears. The users of a watch prefer a more abstract and simplified display; they understand the time abstraction and have no idea how to map the low-level gear abstraction on to it.

There is a great temptation with visualization tools to show all the low-level details. As with the watch, typically the user is unaware of the meaning of most of these details and would have to work hard to understand them (though, they may look pretty when displayed).

Visualization controls should be simple:

Make sure that the display controls on your visualization tools have fewer buttons and gauges than an F-16 fighter. The trend in aircraft over the past 20 years has been to reduce significantly the number of things that the pilot has to watch and understand at any given moment. If our tools follows the previous rule, then we have no (or little) need for complex controls.

A tool will often have dozens of options that can be set, but only a relatively few combinations of these options may be sensible. The controls can be much simpler (and much easier to obtain useful results) with fewer choices.

If you want to provide an escape mechanism for those users who want (or think that they want) these controls, hide these options until requested. A nice technique is to have a “back-up control panel” that is not needed or displayed in most cases. The determined users can request it to appear and then use its features. After they are done, they can hide it again.

4.2. How do we evaluate our visualizations?

Once we produce a new visualization tool, we need to provide a fair evaluation of its usefulness. It is not sufficient to try the tool on a familiar program and see if it gives the expected results (though this is a reasonable first step). If our evaluation of a new visualization tool is based on studies of toy programs that we wrote ourselves, the results may be self-serving. *We must test our tools on real programs.* Taking a new parallel program that was written to accomplish a real computation allows us to understand the strengths (and weaknesses) of a tool in finding real problems.

We also must have other people use our tools on their programs. Users do the most unusual things. When we design a visualization, it fits our own mental models. We then must allow other programs to use the tool to see if the model has a broader appeal.

In IPS-2, we designed an interesting feature that automatically divides a time-histogram into phases. The goal is to have IPS-2 split the execution into time periods so that the performance of each period (phase) can be studied separately. This idea was well received by audiences and reviewers, but proved useless in practice (manual selection of phases proved much more useful). Only experience with real users could provide this type of feedback on our design.

A difficult question is: how do we get real users to base the progress of their research on a flakey research prototype? To provide this level of testing and experience, we have to polish our tools beyond the fragile state sufficient to use them in our own research group. This extra polish takes time and resources that might be spent developing new ideas or writing more papers. The funding agencies for such research need to be sensitive to these requirements.

5. SUMMARY

We should not stop working on parallel program visualization; it is a difficult problem and requires a lot of creative effort. But we should not be seduced by pretty pictures nor should we mistake attractive results for useful ones. We should encourage new ideas but, at the same time, ask hard questions about the usefulness of these ideas.

ACKNOWLEDGEMENTS

I am grateful to Bruce Irvin and Jeff Hollingsworth for their comments and suggestions on this essay.