Graphical Assistance in Parallel Program Development

Kang Zhang* Wanli Ma**
*Department of Computing, Macquarie University, NSW 2109, Australia
**Computer Sciences Laboratory, Australian National University, ACT 0200, Australia

Abstract *Researchers have proposed many visualisation tools that assist the development of parallel programs. A number of graph formalisms or notations – which we will call graph models – have been used to visualise various aspects of parallel programs and their executions. This paper attempts to classify and compare these graph models which provide different information at different stages of parallel program development.*

1: Introduction

There have been several interesting taxonomies and surveys of the systems using computer graphics to assist program development [1]. Yet limited work has been done on classifying or summarising the role of computer graphics in parallel program development, although an increasing number of parallel programming environments that support graphical visualisation have been developed. The aim of this review is to systematically examine the role of computer graphics in different stages of parallel program development. We restrict ourselves to the use of graphics to aid the understanding of parallel programs and their executions. As Miller puts it: "visualistion should guide, not rationalise. 'Guide' means that the visualisation leads you to discover things that you did not already know. 'Rationalise' means that it lets you illustrate things that you already know" [2]. We focus on the graph models that meaningfully interpret parallel computations, rather than on explanatory presentations that improve visual aesthetics. Therefore, we are not interested in a presentation that incorporates visual events or aspects that have no direct counterparts in the computation being depicted.

We propose a model that classifies parallel program visualisation systems according to the purpose of using graphics at different stages of parallel program development. This classification method relates to the definition of "scope" in Myer's taxonomy [3], and "aspect" in that of Stasko and Patterson [1], but tailored for the parallel program development cycle. We find that there are three main stages where computer graphics plays a guiding role: program construction, debugging, and performance tuning. As illustrated in Figure 1, at different stages, graphics plays different roles and may have different notations. Any of the three stages may be entered more than once during the development life cycle. The purpose of using graphics in different iterations of a cycle of the same stage may vary depending on the progress with the program development.

At the first stage, the user may use a graphical editor to build a graphical program which is directly executable with its operational semantics; or to draw a diagram and then generate an intermediate textual program of an existing language syntax for execution. The program may be optimised during this design stage with the aid of graphics. This type of system is called a *visual programming* system [3]. Another type of system, a *program visualisation* system, allows program graphs to be generated from textual

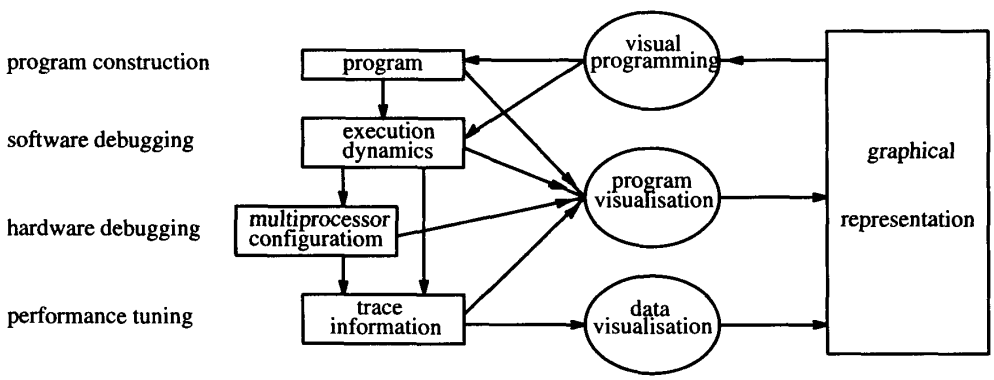

Figure 1: A classification model for using graphics to aid parallel program development

0-8186-6660-9/94 $04.00 © 1994 IEEE

programs. At the debugging stage, the execution dynamics at the program level and its reflection on the multiprocessor configuration at the machine level may be visualised. Program animation techniques and some structure– or process–oriented diagrams may be used to help debugging. The final stage of parallel program development is usually concerned with the performance tuning. The trace information recorded during the program execution can be visualised using various graphical notations which meaningfully depict the program behaviour. The graphical assistance at both the debugging and tuning stages is also classified as program visualisation. Using different conventional visual formats, such as bar charts, to visualise the pure statistical data that profiles the performance of a program is called *data visualisation*. This paper attempts to summarise various *program visualistion* methods used at different stages of parallel program development.

2: Program Construction

The use of graphics at this stage may serve two reciprocal purposes, for instance, using a graphical editor supporting a pre–defined notation to generate visual programs to perform desired functions, or displaying the structure of the program graphically to show either data dependencies or control flow. The former purpose is classified as visual programming and the latter program visualisation, according to Myer [3]. There are several widely used graph models for constructing programs, such as Petri nets, Program Dependence Graphs (PDGs), Process Graphs, and form–based notations. Many systems use more than one type of graph model, each of which may represent a different conceptual model of the problem. The criteria for comparing various graph models for program construction are described below:

Functionality – what is the primary purpose of using a particular model at this stage of program development.

Code generation – whether it is easy or possible at all to generate the textual form of the program from the graphical layout constructed in the first stage. This criterion does not apply to the parallel visual languages which are directly executable on multiprocessor machines without the need to generate textual programs. The parallel languages of the latter category are rare at the present and are not covered in this review.

Scalability – whether a graph model supports hierarchical constructions so that different levels of program details can be viewed or constructed through 'zoom–in' and 'zoom–out' effects. Ideally, a graph model should allow a given display space to visualise a parallel program of any size.

Formalism – whether a graph model is built on a formal basis so that certain program properties are provable or derivable from the graphical syntax and underlying semantics.

Parallelism – how easily the viewer can identify parallelism from a given graph and whether parallel execution is supported by the graphical semantics.

Vocabulary – how many types of graphical primitives are required to construct a graph. The vocabulary is regarded as small if at most five primitives are needed, medium if it is necessary to remind the user through a menu or legend of the primitives, and large if the number of primitives is dependent on the number of language constructs or dependent on other factors.

A comparison of the graph models against the above criteria is shown in Table 1.

Graph model	Functionality	Code generation	Scalability	Formalism	Parallelism	Vocabulary
Petri net	*modelling/verification*	*very difficult*	*low*	*strong*	*indirect*	*small*
Form–based	*structured design*	*easy*	*low*	*weak*	*indirect*	*large*
Process graph	*design/mapping*	*difficult*	*high*	*weak*	*direct*	*small*
PDG	*optimisation/transformation*	*easy*	*high*	*medium*	*direct*	*medium*

Table 1: Comparison of graph models supporting parallel program construction

3: Debugging

Parallel programs may be debugged at both software and hardware levels. At the software level, debugger displays provide various views of parallel program states. These include inter–process communication, procedure call order, etc. At the hardware level, debugger displays show graphically run–time characteristics of a multiprocessor system when executing a particular program. These characteristics include the access pattern of memory, inter–processor communication, processor utilisation, etc.

Program animation plays an important role in debugging parallel programs. We can consider the spy–point and trace–based debugging approach to be magnified or frozen animation. Animation can be performed on the existing

graphical program structures built in the first stage. Typical graph models used in this category include Petri nets and Process Graphs. Almost all the models mentioned in Section 2 can support animation in one way or another. In addition to the four graph models we discussed before, we also compare Causality Graphs [4] and Space–time Diagrams. The criteria for comparing various graph models for debugging are described below:

Orientation – which aspect(s) of the program behaviour a graph model can explicitly provide for debugging purposes.

Characterisation – which particular characteristic of the program a model serves the best to visualise for both debugging and optimisation purposes. This criterion distinguishes one graph model from another by focusing on one characteristic which can be visually described either by the graph structure or through animation.

Granularity – how the primitive graphical components correspond to the complexity of operations. In other words, whether a graph node represents a program statement, a subroutine, a process of computation, or a processor. Graphs of large granularity are suitable for high level (e.g. communication) debugging and those of small granularity are suited for code level debugging.

Comprehensibility – whether a graph model and its animation are easy to understand and their implied meanings of program behaviour are straightforward to comprehend without textual interpretation (our judgement of various graph models against this criterion might be biased by our own experiences).

Table 2 compares various graph models against the above criteria.

Graph model	Orientation	Characterisation	Granularity	Comprehensibility
Petri net	*state transition*	*concurrency*	*medium*	*low*
Process graph	*communication*	*load balancing*	*large*	*high*
Dependence graph	*depencency/sequencing*	*critical path*	*small to large*	*high*
Causality graph	*communication/sequencing*	*time distribution*	*medium/large*	*high*
Space–time diagram	*communication/synchronisation*	*race condition*	*large*	*medium*

Table 2: Comparison of graph models supporting parallel program debugging and animation

4: Performance Tuning

There are two major approaches to displaying performance data. One is event–oriented display, and the other is system–oriented display. Event–oriented display depicts all the interesting events of a program, including types of events and when they happen. This kind of display has a strong relation to the original program and mainly serves the purpose of debugging, as discussed in Section 3. System–oriented display visualises the execution details of a given parallel program in terms of the behaviour of the system components that support the execution. The displays may show the hardware status or operating system activities. The graphical notations and the data displayed are not directly related to the original program.

Time, or a chronological clock, plays an important role in performance visualistion, since computations are always performed within space and time. How time is represented in a graphical notation usually determines the presentation style. There are two common methods of representing time, i.e., explicit and implicit representations. When the chronological order of events are explicitly displayed along a time axis, the user can view the historical data and compare the variations of the data across a period of time. Other visualisation systems use time as an implicit parameter. The display space only shows the performance data without a time axis, while the changing of data is shown through animation.

Many parallel program visualisation systems provide system–oriented displays using various types of statistical information. Conventional graphical notations are usually used at this stage. These include bar charts, dials, meters, scales, histograms, etc. They give quantitative measurements of a system's performance, such as cache misses, memory and interconnection–network traffic, communications, etc.

References

[1] J.T. Stasko and C. Patterson, Understanding and Characterizing Software Visualization Systems. Proc. 1992 IEEE Workshop on Visual Languages, Seattle, USA, 15–18 September, 1992, 3–10.

[2] B.P. Miller, What to Draw? When to Draw? An Essay on Parallel Program Visualisation. Journal of Parallel and Distributed Computing, 18(2), June 1993, 265–269.

[3] B.A. Myers, Taxonomies of Visual programming and Program Visualisation. Journal of Visual Languages and Computing, 1(1), 1990, 97–123.

[4] D. Zernik, *et al.* Using Visualisation Tools to Understand Concurrency. IEEE Software, May 1992, 87–92.