AGAVE: A Visualization Tool for Parallel Programming

Chandra Krintz and Steven M. Fitzgerald
Department of Computer Science
School of Engineering and Computer Science
California State University, Northridge
Northridge, CA 91330-8281
{secsckri,sfitzger}@secs.csun.edu

Abstract

In this paper, we describe a visualization tool, AGAVE, an acyclic graph assembler, viewer, and executor. This tool will facilitate program development, classroom instruction, and performance tuning for parallel programming. AGAVE will allow programmers to assemble, to view, and to execute data flow graphs within an integrated environment. Using AGAVE, a programmer will be able to construct a parallel program and to watch the execution in the form of an animated data flow graph. This activity will provide valuable insight, allowing for debugging and performance testing as synchronization and bottlenecks become obvious.

1 Introduction

With the emergence of high performance, massively parallel computers, new languages, tools, and programming models are needed to effectively utilize the technology. It is becoming more difficult to obtain increased performance from the traditional von Neumann model. The dataflow model has been proposed as an alternative model because it provides high programmability, better modularity, easier program verification, and superior confinement of software errors [AA82].

Data flow contains neither of the two basic characteristics of the von Neumann computer: a single program counter (PC) nor a global updateable memory. First, the inherent restriction in the use of a PC, which provides the machine with a sequence of instructions for execution, is that it implies a central point of control. This is a fundamental limitation of the von Neumann model in parallel environments because it forces concurrency to be exploited within only a small window [AA82]. Second, globally accessible memory can introduce side effects. Although, this memory module provides a powerful means for sharing data structures and for communicating among various parts of the program, it makes extraction of parallelism considerably more difficult [AG94].

Data flow provides a different definition for concurrent execution. Advantages of data flow languages include their ability to make optimal use of implicit parallelism. The only sequencing constraints are the data dependencies contained within the algorithm. An operation many be performed whenever all the required operands are available.

Many applicative languages, such as SISAL [MSA+85], readily convert to a graph-based intermediate form, IFx. Such intermediate languages aid in program generation as they readily expose available parallelism. Graphical representations also provide a useful mechanism for program visualization, enabling debugging and performance enhancement. Additionally, this representation is extremely useful for classroom instruction. Such tools allow programmers to obtain a comprehensive understanding of complicated internal structures and data dependence in parallel programs.

We are developing a multipurpose parallel programming environment to aid teaching, software development, and research. We have named the project AGAVE: an Acyclic Graph Assembler, Viewer, and Executor. This single programming environment will provide three main functions: IFx graph assembly, IFx graph viewing, and IFx graph execution. This environment is comprised of three separate modules that communicate via sockets.

2 Background

In this section, we present an overview of two tools, TWINE [MC92, Mil92] and GDT [MM91], and the graph-based intermediate language, IFx, upon which our project is based.

2.1 TWINE

TWINE is primarily a sequential execution engine for the IF1 language. Programs are serialized for sequential execution. A single valid execution order is predetermined, based on a depth-first-search (DFS) algorithm. TWINE then converts the serialized graphs to C code with a tool called YIFT (Yet Another Intermediate Form Translator). Source code is also retained for debugging purposes. The TWINE execution engine and its runtime system control the execution of an IF1 graph. During execution, system events produce internal signals. These signals provide a mechanism by which internal packages, can seize control of the environment and perform auxiliary functions.

2.2 GDT

The display module of AGAVE is built on top of a graph display tool, which we refer to as GDT. This tool was developed to display acyclic graphs such as IF1 [MM91]. The motivation behind GDT is to provide a means of visualizing data flow graphs to eliminate the tedious job of analyzing the textual form of intermediate languages. The graph layout is hierarchical and appropriate subgraphs can be easily viewed. The tool is an implementation of an algorithm that determines efficient layout of nodes and edges, and minimizes edge crossing.

2.3 IFx

IFx [SG85] is a family of graph based languages derived from the data flow model. These languages were designed as an intermediate form for applicative languages, such as SISAL. Individual operations are represented by simple nodes, e.g., the Plus node represents the addition operation. Compound nodes are used to describe control-flow constructs, such as conditionals and iterators. These compound nodes consist of subgraphs that describe the individual functionality of a control-flow construct. This interaction between subgraphs is implicit, therefore, the control-flow construct can be implemented differently for different architectures.

Consider the mathematical expression, $((x+a)*(x+b)*(b+c*a))$. The data flow graph for this expression is depicted in Figure 1. The inherent parallelism of the expression is made apparent by the graphical representation. Since the nodes on the first level are independent of each other, they can execute concurrently. Notice that additional information from the source code is presented on graph edges. For example, the %na=x indicates that the name of the variable (x) associated with an edge.

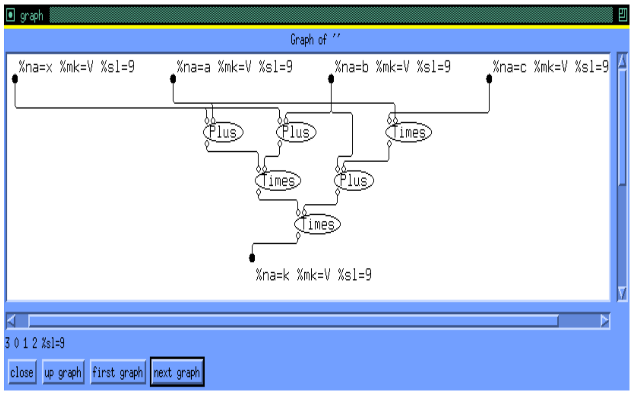

Figure 1: Graphical representation of ((x+a)*(x+b)*(b+c*a)) in IF1

3 The AGAVE Environment

We are developing a prototype that will be used in a learning environment to aid in the development of parallel programming strategies. Its purpose is to provide both a user-friendly display tool for IFx languages and a mechanism for visualization of parallel program execution. The three modules that comprise AGAVE are the Executor, the Viewer, and the Assembler.

The Executor module is primarily the TWINE execution engine. We have extended the package mechanism within the tool to communicate with other tools via sockets. Information about the execution of individual nodes can be communicated outside the program through sockets. In addition, we are extending sequential execution engine of TWINE to be a parallel execution engine. This will enable visualization of parallel execution.

The Viewer module of the environment is a direct adaptation of the graph display tool. We have incorporated the package methodology from TWINE into this module so that it may receive external information through sockets. Additionally, we have included the development of a user-friendly interface, interaction with the Assembler module, and interactive links between IF1 and user source code.

Since both the Executor and the Viewer use packages, interaction between the tools is simplified. As different internal events occur, signals are passed between the tools via the package mechanism. A graphical overview of the environment is depicted in Figure 2. For example, the EXECUTOR signals the VIEWER when either a node is about to be executed or has completed its execution. The VIEWER will then highlight or de-highlight the corresponding node. As these operations are performed for multiple nodes, the execution of the graph can be visualized.

The ASSEMBLER is the third module in the environment, which will interact with both the EXECUTOR and the VIEWER. The ASSEMBLER is a drawing tool that can be used to modify the graph layout manually, to generate new graphs, and to convert user-generated data flow graphs to the corresponding IF1 code. The purpose of the module is to aid in the understanding of data flow languages and to provide increased user control over graph layout.

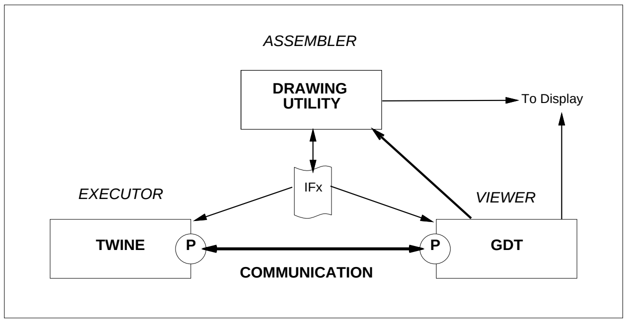

Figure 2: The AGAVE environment. The **P** indicates the package component of the EXECUTOR and VIEWER.

4 Conclusion

AGAVE is an environment developed to aid in the management of parallel programming considerations, such as race conditions and performance tradeoffs between parallelization and communication. It incorporates three very important aspects of program development: design, implementation, and debugging. It is a tool that will encourage involvement by students, faculty, and researchers in data flow by providing a visual representation of parallel programs. It allows visualization of implicit parallelism in programs and provides a means for debugging and effectively utilizing parallel architectures. AGAVE builds on research from GDT and TWINE projects and can easily be enhanced and extended into new projects.

References

[AA82] Tilak Agerwala and Arvind. Data flow systems. *IEEE Computer*, 15(2):10–13, February 1982.

[AG94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* The Benjamin-Cummings Publishing Company, Inc., second edition, 1994.

[MC92] Patrick J. Miller and Walter Cedeño. *A User's Guide to TWINE*, September 1992.

[Mil92] Patrick J. Miller. TWINE: A portable, extensible SISAL execution kernel. In *Proceedings of the Second SISAL Users' Conference*, pages 243–256, San Diego, California, December 1992. Lawrence Livermore National Laboratory, CONF-9210270.

[MM91] Srdjan Mitrovic and Stephan Murer. A tool to display hierarchical acyclic dataflow graphs. In *International Conference on Parallel Computing Technologies*, pages 304–315, September 1991.

[MSA$^+$85] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2.* Lawrence Livermore National Laboratory, M-146 edition, March 1985.

[SG85] Stephen Skedzielewski and John Glauert. *IF1 — An Intermediate Form for Applicative Languages.* Lawrence Livermore National Laboratory, Livermore, CA, M-170 edition, July 1985.