FINE-GRAIN DATAFLOW MODEL AND ALGORITHMS FOR VISUALIZATION SYSTEMS

BY

DEYANG SONG

B.Sc., Fudan University, 1985
M.S., University of Lowell, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

Dataflow computing model is a simple yet powerful mechanism for constructing distributed visualization applications that transform numerical data into images. However, current dataflow visualization systems have serious memory efficiency problems when processing large data sets. In this thesis, we have identified a class of visualization algorithms that require sublinear memory space and are suitable for implementation in these systems. These algorithms are so called *fine-grain* algorithms, as opposed to the *coarse-grain* approach adopted by most of the previous systems. The fine-grain algorithms and systems are studied within the framework of fine-grain dataflow computing in general, and the *Syntax-directed Dataflow Transformation Method* (SDTM) in particular, in this thesis. The SDTM model combines the dataflow programming paradigm with the syntax-directed translation techniques. The new model extends the classical dataflow model in which tokens and nodal functions are atomic. It allows structural definition of the tokens and explicitly states the sequence of nodal actions in relation to the token structures. Unlike previous systems that tend to introduce multiple copies of large data sets during execution, our system is more memory efficient. This efficiency is achieved by keeping a dynamically adjusted minimal window on the input data stream, through the use of attribute grammar to specify attribute dependency and data transformation. Based on the fine-grain algorithms and the SDTM model, we have built a fine-grain visualization system that exhibits faster speed, less memory usage, and higher CPU utilization than a typical coarse-grain system.

To my wife, Lori Lai, and my parents

ACKNOWLEDGEMENTS

First I'd like to thank my advisor, Professor Eric Golin, for his interest in me and his guidance throughout this project, and my co-advisor, Professor Michael Norman, for his continuous support for me to work in his group at the National Center for Supercomputing Applications (NCSA). Other members of the committee, Professor Daniel Reed, Professor William Kubitz, Professor Ralph Johnson, Professor Simon Kaplan, Dr. Polly Baker, have given me valuable comments. Professor Robert Haber and Professor Donald Hearn have exposed me to the field of scientific visualization in the early days of my graduate study here at the UIUC campus.

Special thanks to Jeff Hostetler for giving me access to the IBM Data Explorer. Thanks also to Crystal Shaw, Kelvin Sung, Tom Zhou, Yong Ma, Henry Neeman, and Lun Xiao who have given me helps at different stages of my thesis project. Michael Norman, James Stone, David Clark, and Greg Bryan have provided me with rich scientific data sets to experiment with.

Thanks to Dr. Robert Brown at SGI, Lloyd Treinish at IBM T.J. Watson Research Center, and Dr. Steven Zenith at KAI (Kuck and Associates, Inc.), who have provided valuable input to my work. Dr. H.-G. Pagendarm from DLR, Göttingen, Germany, provided me with a case involving a large AVS network.

The financial and computing facility support provided by NCSA is gratefully acknowledged.

Portions of this thesis have been published as [90] and [89].

TABLE OF CONTENTS

Chapter

1 Introduction 1
- 1.1 Thesis Overview 2
- 1.2 Visualization Software Architectures 4
- 1.3 Dataflow Computing Models 7
- 1.4 Motivations 9

2 Related Work 14
- 2.1 Visualization Systems 14
  - 2.1.1 GRAPE 14
  - 2.1.2 FRAMES 15
  - 2.1.3 AVS 15
  - 2.1.4 apE 16
  - 2.1.5 IBM Data Explorer 16
  - 2.1.6 IRIS Explorer 18
  - 2.1.7 Khoros 18
  - 2.1.8 RIVERS 19
  - 2.1.9 DataVis 20
  - 2.1.10 Other Systems 20
  - 2.1.11 Summary 20
- 2.2 Attribute Grammars 24
- 2.3 Other Models 29

3 The Syntax-Directed Data Transformation Method (SDTM) 31
- 3.1 Coarse-Grain and Fine-Grain Dataflow Approaches: a Comparison 31
- 3.2 SDTM: A Fine-Grain Dataflow Model 34
  - 3.2.1 The Stream Data Language 35
  - 3.2.2 SDTM Modules 36
  - 3.2.3 SDTM Pipelines 42
- 3.3 Analytical Performance Analysis 46

4 Fine-Grain Visualization Algorithms 54
- 4.1 The Normalization Algorithm 54
- 4.2 The Slicing Algorithm 58
- 4.3 The Iso-Surface Algorithm 62
- 4.4 The Volume Rendering Algorithm 63

4.5 Discussion . . . 67

5 The Implementation of an SDTM Visualization System . . . 69
5.1 System Overview . . . 69
5.2 The Translator Generation Tool . . . 70
5.2.1 Parser Generator . . . 72
5.2.2 Lexical Analyzer Generator . . . 73
5.3 The Attribute Analyzer . . . 74
5.3.1 Attribute Dependency Analysis . . . 74
5.3.2 Attribute Storage Reduction . . . 75
5.4 Fine-Grain Pipelines . . . 79
5.4.1 The Data Slicer . . . 79
5.4.2 The Iso-Surface Tool . . . 81
5.4.3 The Volume Rendering Tool . . . 83

6 Performance Measurements . . . 85
6.1 Methodology . . . 85
6.2 Results and Discussion . . . 90
6.2.1 The Data Slicer . . . 90
6.2.2 The Iso-Surface Tool . . . 103
6.2.3 The Volume Rendering Tool . . . 111
6.3 Conclusions . . . 118

7 Conclusion . . . 119
7.1 Summary . . . 119
7.2 Contributions . . . 120
7.3 Future Work . . . 122

Appendix . . . 125

A The Flux Format . . . 125

B An Attribute Grammar Specification of the Module `norm` . . . 127

Bibliography . . . 144

Vita . . . 154

LIST OF TABLES

2.1 Feature comparisons of dataflow visualization systems. 22

3.1 Syntax-directed definition for module `norm`. 40

6.1 Performance comparison of coarse-grain and fine-grain implementations of an iso-surface construction pipeline in different hardware environments. 105
6.2 Performance comparison of coarse-grain and fine-grain implementations of a volume rendering pipeline in different hardware environments. 112
6.3 Performance comparison of coarse-grain and fine-grain volume rendering pipelines on a single processor using a large data set. 113

LIST OF FIGURES

1.1 The conceptual visualization model represented in a pipeline. 5
1.2 Sucessive snapshots of a dataflow program computing $(a+b)(b+c)$. 8

2.1 The apE pipeline construction and execution tool `wrench` 17
2.2 Classification of attribute grammars 27

3.1 A pipeline with a multiple-input module 41
3.2 The back-propagation of lookahead tokens 43
3.3 A linear LC pipeline. 44
3.4 A linear visualization pipeline 47
3.5 Time difference between a coarse-grain and a fine-grain pipeline 50
3.6 A comparison of analytical and empirical speedups 52

4.1 A 3D gridded scalar data set in *Flux* format 55
4.2 A 3D gridded scalar data set suitable for stream processing 56
4.3 Attribute dependencies in the variable group “foo” induced by the module `norm` . 57
4.4 Examples of slicing a 3D gridded data set 58
4.5 A fine-grain algorithm for iso-surface extraction. 63
4.6 An example of backward integration for ray-casting 65
4.7 The forward integration algorithm 66
4.8 The backward integration algorithm 66

5.1 The software architecture of apE modules. 71
5.2 The software architecture of new modules. 71
5.3 The `Data Slicer` pipeline. 80
5.4 Configuration of the `Data Slicer`. 80
5.5 Screen dump from the `Data Slicer` visualizing a CFD simulation result. 81
5.6 The iso-surface extraction pipeline. 82
5.7 An iso-surface from a 3D jet simulation data set. 82
5.8 The triangle list group. 83
5.9 The volume rendering pipeline. 83
5.10 Volume rendering of a 3D jet simulation data set. 84

6.1 Memory usage of the apE Data Slicer pipeline 94
6.2 Memory usage of the SDTM Data Slicer pipeline 95
6.3 Memory usage of the apE and SDTM Data Slicer pipelines 96
6.4 A closeup view of the memory usage of the Data Slicer pipelines. 97

6.5 Peak memory usage vs granularity for the Data Slicer 98
6.6 Speedup vs granularity for the Data Slicer. 99
6.7 CPU utilization vs granularity for the Data Slicer 99
6.8 Speedups of different slicing planes 100
6.9 CPU utilization of different slicing planes 101
6.10 Speedup vs number of processors 102
6.11 Memory usage of the Iso-Surface tool 106
6.12 Peak memory usage vs granularity for the Iso-Surface tool 107
6.13 Speedup vs granularity for the Iso-Surface tool. 108
6.14 CPU utilization vs granularity for the Iso-Surface tool. 108
6.15 Correct and incorrect number of lookahead tokens. 109
6.16 Timelines of two pipelines having different number of lookahead tokens. 110
6.17 Memory usage of the Volume Rendering tool 114
6.18 Peak memory usage vs granularity for the Volume Rendering tool. 115
6.19 Speedup vs granularity for the Volume Rendering tool. 115
6.20 CPU utilization vs granularity for the Volume Rendering tool. 116
6.21 Speedup of the volume rendering tool with different data formats. 116
6.22 CPU utilization of the volume rendering tool with different data formats. 117
6.23 Comparison of an empirical speedup curve with the analytical curve. 117

Chapter 1

Introduction

Most current visualization systems favor the dataflow paradigm where data flows into one end of a pipeline (or network) of modules and comes out from the other end as an image. An animation can easily be made this way if multiple data sets undergo the same pipeline. One of the nice features of a pipeline is that the throughput is independent of the number of stages. Only the slowest stage, the bottleneck, determines the throughput. Another feature is that the modules can run in parallel, if enough CPU's are available. In these systems each data set is treated as one frame which is transmitted and processed atomically, much the same way as tokens are treated in a dataflow computer. These software systems are often called *Large-Grain* (or *Coarse-Grain*) *Dataflow* systems [6, 83, 101].

Based on this dataflow model, each module in a coarse-grain dataflow system keeps an input and an output buffer that hold the entire data set being processed. There are two major problems associated with this way of data processing. Since scientific data sets are inevitably very large, it takes a long time to fill the visualization pipeline[1], especially when the pipeline is running in a distributed environment. Furthermore, since all of the intermediate data (partial results) have to reside in memory, the total memory requirement for a dataflow visualization system is extremely large.

The duplication and buffering of large data sets are the major drawbacks of dataflow systems, especially for visualization systems [10, 101]. These are serious problems that hinder the ability of visualization systems to process scientific data sets that are becoming increasingly large.

[1] In this thesis, we refer to dataflow or pipeline systems as software systems.

This thesis studies how the performance of visualization systems can be improved by using a fine-grain dataflow computing model. In many cases, memory requirements of fine-grain algorithms are sublinear in terms of the input data size. Some algorithms have the property that their memory requirements are independent of the size of the data set being processed, so they have the potential of processing data sets of unlimited size. The new approach also leads to execution speedups and improved CPU utilization.

Dataflow as a model of computation has been applied previously both at the architectural level [15, 102] and at the programming language level [34, 35]. In these systems each token is small and indivisible. The issue of granularity either does not exist or can be ignored. In visualization systems, however, the granularity of data has a significant impact on system performance. In this study we will try to answer the following question:

- Are there visualization algorithms whose memory requirements are sublinear in terms of the size of the input data set? These "streaming" algorithms are called fine-grain algorithms.
- If these algorithms exist, what is the formal and natural way to specify scientific data and its granularity, and the computations in relation to data granularity?
- How do various fine-grain techniques improve the execution speed, CPU utilization, and memory usage of a software pipeline?

We will identify a class of visualization algorithms that require sublinear memory space and are suitable for implementation in dataflow systems. We will also propose a fine-grain dataflow computing model in which the data granularity depends on both the input data and the module function, and the function granularity comes as the result of data granularity. Our granulating method is described in the framework of attribute grammars, a syntax-directed translation technique.

1.1 Thesis Overview

The thesis consists of two major parts: a fine-grain dataflow computing model based on attribute grammars – the *Syntax-directed Data Transformation Method* (SDTM) – and several fine-grain visualization algorithms that use the data locality property. The SDTM model provides a framework to develop and implement fine-grain algorithms, but it must be emphasized that fine-grain pipelining and attribute grammars are essentially independent issues and that SDTM is only one way to implement fine-grain pipelining.

A fine-grain pipeline consists of fine-grain modules, each of which implements an algorithm that exploits data locality by inputting, processing, and outputting data in small chunks. The distinction between a fine-grain module and a coarse-grain module is that a fine-grain module starts processing input data and generating output data without seeing the end of the input data stream. The granularity of the data can be adjusted by each individual module. The fine-grain algorithms can be implemented using the SDTM approach, or they can be implemented using other approaches (see Section 2.3). In any case, there must be a way to break down the data into smaller pieces.

In the SDTM model, the dataflow (data passed between modules) is considered as a Context-Free Language (CFL) and is specified using an attribute grammar (grammar rules with attributes). There are grammar rules for scientific data, geometric data, color tables, and so on. Each symbol in the grammar represents a piece of data, which can be further broken down if the symbol is a nonterminal symbol. In this way, a large data set can be decomposed into small chunks. Each module in the SDTM model is essentially a language translator. The single compute function found in the coarse-grain module is also decomposed into "smaller" semantic functions which are attached to symbols in the grammar rules. A semantic function is called when a meaningful piece of data is received. Each semantic function has access to data within a certain locality of the input data stream. Thus the module effectively keeps a dynamically adjusted window over the input data stream. Another name for this way of computing is the "streaming approach". The essence of this approach is data and function decomposition. If the input data to a module cannot be decomposed because of the requirement of the function (e.g., streamline integration), then only one semantic function is needed and it is attached to the top-most grammar rule. Thus the SDTM model provides a unified model for both fine-grain and coarse-grain computing.

In the remainder of this section, we will discuss the software architecture of dataflow visualization systems and compare it with that of systems using other models. We then briefly discuss the concept of dataflow computing. At the end of this chapter we'll give the motivations for this thesis work.

Chapter 2 provides an overview of the related work in building dataflow visualization systems, an overview of attribute grammars, and the reasons for using attribute grammars.

Chapter 3 presents the SDTM model for dataflow computing, followed by analytical studies of the performance of the new model. Some unique properties of the SDTM-based fine-grain pipelines are revealed and studied.

Chapter 4 contains four fine-grain visualization algorithms that are commonly used. These algorithms have sublinear memory requirements, and are implemented in the SDTM pipelines described in Chapter 5.

Chapter 5 describes the implementation of a module generator based on attribute grammar theory and three visualization pipelines consisting of fine-grain modules generated by the module generator.

Chapter 6 reports the performance of the fine-grain pipelines. We will show how data granularity, data representation, hardware environment, etc., affect the performance of these pipelines.

Chapter 7 summarizes the contributions of this work to visualization software systems and to the area of dataflow computing. It also points out future research directions.

1.2 Visualization Software Architectures

Scientific visualization transforms numerical data into images. It is a tool used by scientists to comprehend large volumes of data and to see the unseen. Scientific visualization systems have evolved from post-processing systems, to interactive, extensible, and distributed environments. The capabilities of these systems have been extended from generating 2D static images to doing 3D animations. With the fast advances in graphics hardware and computer architectures, the need for portable, high performance visualization software systems is ever increasing.

A typical scientific visualization application would involve the following steps [18, 29]:

1. *Data generation/collection*: This is the process of getting raw data, typically referring to computational simulations or empirical data measurements. In the following discussions, we will use the word *simulation* to refer to this stage.

2. *Data manipulation*: Raw data may be reorganized and normalized. In the case of empirical data, noise may be reduced.

3. *Data mapping*: Numerical data is mapped to attributes of graphical objects such as shape, color, and texture.

4. *Rendering*: Graphical objects are rendered in a scene, possibly with addition of external lights. The final image is typically viewed on a workstation.

5. *Interpretation*: The rendered graphical objects represent a subset of the original data. Inquiries and analysis of the subsets are allowed in this stage. By doing so, a scientist can gain insight into the original data set.

6. *Feedback*: The user can go back to previous phases and make changes to the parameters so that new results can be generated and visualized again.

This conceptual model of visualization can be depicted by the pipeline in Figure 1.1.

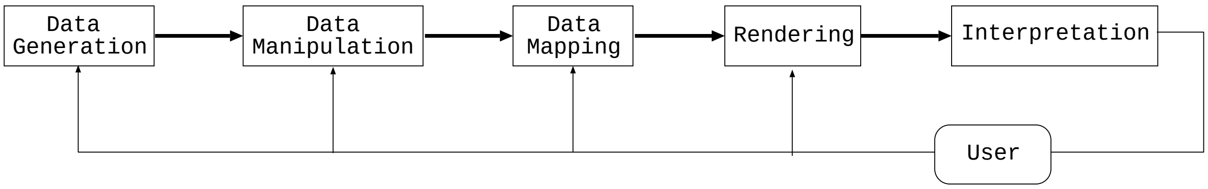


Figure 1.1: The conceptual visualization model represented in a pipeline.

Visualization software systems have undergone several generations [64]. The first generation are the *post-processing* systems, which include steps 2–4 above. Data files serve as the medium connecting the simulation and visualization. These systems are usually stand-alone programs with a centralized control panel. The second generation systems can also do *tracking*. With tracking, the data generation step is incorporated into the visualization system so that the images can be viewed as the simulation proceeds. These systems are typically distributed. The systems discussed in the following section fall into this category. Notice that the intermediate data files are unnecessary for this type of system. The next generation software will be able to do *data interpretation, analysis, and steering* of the simulation. This will include the last two steps in the visualization process and will form a tight loop between the simulation and visualization.

To meet the demands of various disciplines, a general-purpose visualization system must at least have the following properties:

1. The ability to extend a language to handle new application requirements without changing the basic environment. Here a language refers to a set of components from which applications are built. The dataflow visualization systems described below are examples of such languages. They are often called *Application Builders.*

2. The capability of distributing different functional layers to different machines in a distributed network that may include supercomputers, high performance graphical workstations, and desktop computers.

In other words, such a system must be interactive, extensible, and be able to run on different hardware platforms, in different software environments, and for different applications [65, 97, 101].

There are basically two different approaches [10] to achieving these goals: the "reductionist" approach (modular, loosely coupled systems) [19, 29, 31, 33, 44, 87, 68, 80, 81, 100] and the "holistic" approach (integrated, tightly coupled systems) [10, 105, 70]. The reductionist approach breaks down the process of specifying and generating images into a number of separate parts. Different parts are connected into a pipeline or a network that allows the data to flow from one end to the other, undergoing transformations at each node.

The holistic approach closely couples these parts together using a kernel structure that allows intense interaction between them. Each of these two approaches has advantages and disadvantages.

- The reductionist approach encourages modular design, clear interface between modules, and distributed processing. The software systems are flexible and easy to extend. One of the important goals of a dataflow software toolkit is to allow the users to create their own tools. One can add new modules or replace existing modules without touching other parts of a system. This method of "customization" is more powerful compared to other customization methods such as command line options or X Windows resources, in the sense that one has unlimited control over customization of a software system with the former approach. The disadvantage of this approach is the inefficiency of passing large amounts of data between programs[2], and having multiple copies of data in memory at the same time. Details of the problem will be explained in Section 1.4.

- The holistic approach, on the contrary, encourages sharing of data. It exploits, for example, temporal coherence existing in the data set. Its software system is usually a monolithic package, but often runs faster. The disadvantage is that, because of its centralized control mechanism and data management, the holistic approach usually can not take advantage of a distributed computing environment. Furthermore, the software system usually has little or no extensibility.

Because of its advantages, the "reductionist", or dataflow approach, is gaining more popularity [10].

It is interesting to observe that because of the difference in system models, the two types of systems have different emphasis in the software design. For a tightly integrated system, the emphasis is on the functionality of the system, the file format, etc. For a dataflow system, design emphasis is shifted toward the underlying data structures for inter-modular communications. Less emphasis is put on functionality since that can be extended by the user community.

1.3 Dataflow Computing Models

A dataflow program can be represented by a directed graph in which each *node* performs a function and the *links* carry data values from nodes to nodes. Each data value is called a *token*. The execution of a dataflow program can be regarded as a succession of snapshots of the dataflow graph between which tokens are removed and created by the "firing" of the nodes according to the following rules:

1. A node is enabled only if there is a token on each of its input links;
2. An enabled node may be fired to generate an output token;
3. An output token can be placed on an output link only if it is empty.

An example of successive snapshots of a dataflow graph is shown in Figure 1.2.

[2] What we refer to here is a general problem considered independently of the underlying communication mechanisms used.

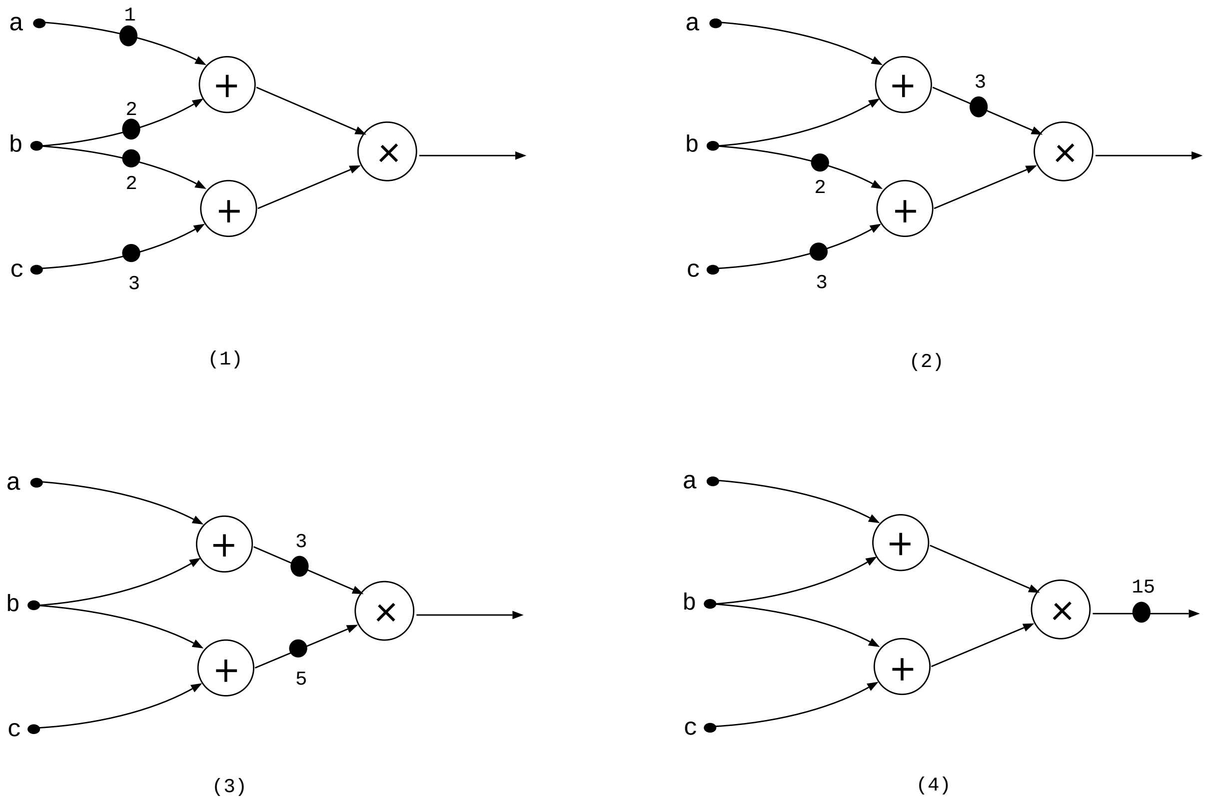


Figure 1.2: Sucessive snapshots of a dataflow program computing $(a+b)(b+c)$.

Dataflow is used as an ideal abstraction to capture the characteristics of the entire process of scientific visualization (see Figure 1.1). The concept of dataflow is not new, however, and has been broadly applied. Researchers have worked on dataflow computer architectures to explore fine-grained concurrency in computer programs [15, 26, 34, 102]. Large-grain dataflow software systems have been shown to help problem solving at a high level in areas other than scientific visualization [6, 81, 92]. In these systems each node of a dataflow diagram is a functional entity that can be implemented either as a function in a traditional language or as an individual program. The scheduling can be done by a special scheduler or by the underlying operating system.

Dataflow software systems also abide to the basic principles of data flow model of computation [25]:

1. *Asynchrony*: All operations are executed when and only when the required operands are available;

2. *Functionality*: All operations are functions; that is, there are no side effects.

The first principle implies distributed control of a dataflow system. It makes the detection of concurrency fairly easy and makes a dataflow software system extremely suitable for implementation in distributed environments. The second principle implies that there is no global data. Components of a dataflow system don't have a shared memory space, and they communicate with each other using messages (tokens).

The concept of dataflow can be applied at different levels of a system architecture. There are at least four distinct levels: the architecture level, the programming language level, the task level, and the algorithm level. The current dataflow visualization systems are implemented at the algorithm level and some of them include the task level also.

In traditional dataflow models, tokens are simply assumed to be indivisible and have a flat structure, and node firing is assumed to be atomic. In some extended models some of these assumptions are relaxed. For example, in Landry [56], the firing condition is relaxed and the tokens are typed but are of fixed length for a given type. In Uchida [99], the concept of variable length tokens are proposed to support composite token processing. However, the hierarchical structure of the tokens and the precise description of how they are related to the execution of the nodes has not been studied before.

1.4 Motivations

Time and space are two important criteria in evaluating algorithms. However, memory efficiency has not been addressed in many visualization system builders. This negligence makes the current visualization systems unsuitable for processing very large data sets.

It has been suggested that, in computational sciences, computer memory is often the limiting factor for larger simulations [71]. For many scientists, the problems they are solving now are orders of magnitude less than desired. A simulation of galaxy cluster formation with a resolution of 512^3 zones currently being worked on by Norman [11] requires several gigabytes of memory. Doubling the resolution in each dimension leads to an 8 times increase in both memory and computational demand. This is why scientists doing simulations on supercomputers are pushing for more and more memory.

One approach to dealing with memory limitation is to solve smaller problems. Another approach is using virtual memory. A better solution is to find sublinear algorithms that don't use as much (virtual) memory but still solve large problems. Several factors point to the need for memory-efficient algorithms.

One of such factors is the widening gap between the performance of the main memory and CPU's. Statistics show that the computer main memory increases four times every three years (about 59% a year), mass storage is half that, and processor speeds increase at least 70% a year [69]. Similar statistics suggesting the same trend are also reported elsewhere [78, pages 426–427]. Amdahl's rule of thumb states that memory capacity should grow linearly with CPU speed to keep a balanced system [3]. This CPU-memory gap suggests that the power of a computer is likely to be limited by the memory capacity, not by the CPU speed.

Although large secondary storage (hard disks) can support large virtual address space, the widening gap between fast main memory and slow disks [107] has made it important to exploit techniques for improving locality of memory access in algorithms in order to reduce the virtual memory page faults and page swaps. Table 6.3 presents an example showing how this factor affects the performance when large data sets are processed on a workstation with limited memory size.

The third factor lies in the difference in memory size between supercomputers and workstations. In real-time visualizations where simulation and visualization form a tight loop, the simulation module normally runs on a supercomputer with large memory and disk space and the visualization modules usually run on graphics workstations with much less memory. This forces the simulation data to be down-sampled before being fed into a visualization pipeline, which reduces the accuracy of the visualization.[3]

Dataflow computing, with many appealing merits, is not without its shortcomings. Passing large amount of (intermediate) data between modules has been criticized as one of the inherent problems of the dataflow systems, both in architecture design and software systems [10, 25, 34]. This problem becomes even more serious in scientific visualization systems because the

[3] Today's largest supercomputers have about 2 to 8 gigabytes of memory while the largest graphics workstations have about 256 megabytes of memory and an ordinary graphics workstation has about 16 megabytes of memory. So a supercomputer memory is about 8 to 512 times larger than that on a graphics workstation, which roughly means that the large simulation data sets have to be down sampled to at least one eighth of the original size before transmitted from a supercomputer to a graphics workstation for visualization.

visualization process usually involves large volumes of data residing both in main memory and on disks. Buffering and duplicating large data sets at each individual module makes the situation worse for visualization systems that are already hungry for memory. Numerous reports [28, 104, 74, 93] have complained that dataflow visualization systems use excessive memory during execution and often cause the computer systems to run out of swap space. Pagendarm [74] described an AVS network of 56 modules used to visualize a shock wave in a hypersonic flow field. The grid size is $63 \times 162 \times 31$ and each point holds 8 floating point values. The file is about 12 MB large, but a SPARCstation 10 with 96 MBytes of main memory and 250 MBytes of swap space cannot satisfy the memory need of the AVS network.

Dyer [101] indicated that the problem lies in the software architecture of these visualization systems. He pointed out that treating large datasets atomically is the main reason for causing the low performance. Other people also realized the problem with coarse-grain dataflow visualization systems [83], but not much work has been done to tackle it.

Some related work has been reported in managing large volumes of scientific data. Managing large data sets is a central issue of visualization systems [97, 94], and it is also a challenging one [5]. It has been pointed out that our ability to manage and comprehend data is being rapidly outstripped by our ability to generate them [97]. Treinish [95] summarized the importance and the progress in effective external data management. Equally important is the internal representation and movement of the data. Efficient data movement is of vital concern for dataflow systems [4, 80]. A simple solution using binary data format has been used [24, 33]. This method does not solve the inefficiency problem, but only improves the transmission speed. Using shared memory can alleviate the problem in some cases, but may fail in other cases (for instance, in distributed systems).

There are two ways to solve the problem of data duplication and buffering in dataflow systems. One is to change the underlying dataflow execution model to a more efficient one while retaining the dataflow interface to the users. This requires a compiler that can translate a dataflow visual program into executable programs. This approach is used in the GRAPE system. In GRAPE, each node in the dataflow diagram is a function. A DAG compiler compiles the diagram into a single executable program. What this approach does is essentially to let all the nodal functions of a dataflow program share a single virtual address space, thus eliminating the movement of data across the process boundary. A similar approach taken by IBM Data

Explorer is to compile every function into a single executable program, thus further eliminating the link time required in the GRAPE system. A penalty for this is the large executable program residing in memory, possibly containing many unnecessary functions. Considering the size of the data sets, this small penalty is probably negligible. But the approach used by GRAPE and the Data Explorer only solves the data duplication problem, and leaves the problem of buffering large data sets unsolved.

In the Data Explorer, parallel processing is used to speed up data processing. Parallel processing can be done by explicitly or implicitly partitioning large data sets into smaller ones. The input data set is partitioned into smaller, non-overlapping subsets. The executive creates multiple threads of tasks to process each subset. This approach is called the "medium-grain" parallel processing approach [96]. The problem with partitioning large data sets into small ones is that although each task is now handling a smaller data set, the entire data set is still cached in memory collectively by each task. However, the data partitioning method is attractive when it is applicable to the problem solution and when multiprocessor, shared-memory computer hardware support is available.

Another way to solve the problem is to change the assumption that data sets are transmitted and processed atomically. What it means is that data elements should be accessed in a sequence of smaller chunks instead of once in a large chunk. Data processing is started as soon as enough data elements have been received. This is the streaming approach, or fine-grain approach, taken in this thesis. The most obvious advantage of the streaming approach is that it can handle large data sets that can not be loaded entirely into the memory of the machine. Unfortunately, none of the dataflow systems currently in use provide a mechanism to do stream processing.

It should be noted that the streaming approach and the data partitioning approach are complementary, rather than competing, strategies. The streaming approach does not imply one-process-per-module style of implementation. We will give two examples of combining parallel processing and streaming approaches in the following discussion.

The data partitioning method works well with rectilinear or other types of regular mesh data sets. But it is hard to partition data in a non-regular mesh such as a curvilinear mesh or an unstructured mesh. It is both hard to partition the mesh and hard to map the partitions onto the underlying multiprocessor topology. In Williams [108] a *Direct Projection Volume Rendering* algorithm was developed for rendering irregularly meshed scalar dataset. The algorithm directly projects each cell of the mesh onto the screen in back-to-front or front-to-back order. Each cell has an assigned color value and opacity value. The algorithm is as follows:

1. create the adjacency graph for the cells of a given convex mesh.
2. for any specified viewpoint, a visibility ordering can be computed simply by assigning a direction to each edge in the adjacency graph based on the in-front-of relation.
3. the splatting algorithm traverses the directed acyclic graph in a topological order and computes ray integrations for the mesh cells being visited.

Although the input data mesh cannot be partitioned into spatially local partitions, computations within each step of the algorithm can be parallelized. This is an example of embedding parallelism within the streaming approach. The streaming is realized when multiple input datasets are being rendered.

The above example is one way of combining parallel processing with streaming. Another way of combining the two is by embedding streaming approach inside parallel tasks. In the IBM Data Visualizer, a large dataset can be partitioned into n smaller sets. One task is created for each of the partitions. Within each task, the input data objects are still processed in one big chunk. With the streaming approach developed in this thesis, each task can process input data in a stream fashion.

In this thesis, we will show that in most cases coarse-grain modules can be converted into fine-grain modules. In our system, a fine-grain module degenerates to a coarse-grain one in the worst case.

Chapter 2

Related Work

2.1 Visualization Systems

A large number of interactive, integrated visualization software systems have been developed in the last five years. Most of them use the dataflow paradigm at both the user interface level and the module implementation level, although various degrees of difference exist between different packages. In this section we will survey dataflow visualization systems. Systems that use dataflow visual programming paradigms but not in the scientific visualization domain are not covered. Hils [35] provides a good survey of such systems.

Unless otherwise noted in the following discussions, "visualization systems" refer to systems that use dataflow computational model at various levels.

2.1.1 GRAPE

GRAPE (General Rendering Applications Programming Environment) is a programming environment aimed at facilitating experiments with the structure and basic algorithms of a display system [68]. The environment consists of two levels: micro and macro levels. The micro level consists of nodes written in conventional languages and saved in libraries. At the macro level, nodes are connected into directed acyclic graphs (DAG) which are compiled by a dag compiler to form an application. Typed common data elements are defined to allow nodes to communicate. Unlike most other systems surveyed below, the nodes in GRAPE are not executables. They must be compiled each time a new DAG is created. Their connections to each other are static in that they are established at compile time by the dag compiler. Each node in GRAPE has a *gofor* function to deal with its input and *server* function to deal with its output. This is very similar to the functions performed by the *control wrapper* and the *data wrapper* in the IRIS Explorer (Section 2.1.6).

2.1.2 FRAMES

FRAMES (Flexible Rendering, Animation and Modeling Experimentation System) is a collection of software tools in the UNIX environment [80]. Each module of FRAMES is an executable program. They can be built into a pipeline and they communicate through UNIX pipes. FRAMES allows the distribution of processes to multiple machines on a local network. Messages passed between modules form the command language. Each module acts as a command language interpreter that selectively focuses on those parts of the data stream that it needs to do its processing. Like UNIX, FRAMES only allows linear pipelines. Each module has only single input and output.

2.1.3 AVS

AVS (Application Visualization System) is a general-purpose 3D visualization system [100, 12], running on a large number of different platforms. It contains a collection of executable modules and libraries upon which new modules can be built. Modules are connected into an application in the form of a DAG, or network in AVS's terminology. Each module contains a computational function that gets called every time this module is "triggered". A visual-oriented graph editor helps the user to build the applications. Modules can have multiple input and output communication ports. Data are communicated between modules through these ports. In AVS, the number of ports a module can have is fixed and the ports are typed so communicating ports must have compatible types.

AVS uses demand-driven execution model. It has a flow executive that acts as a run-time scheduler. The flow executive supervises data movement between modules, keeps track of which parameters have changed, and invokes modules in the correct order. This centralized control strategy is a disadvantage in a distributed environment. In AVS V3.0, only one module is active at a time.

AVS has a scripting capability. An AVS application (network of modules) can be saved as a script and be invoked without the graphical user interface. It also has a distributed capability. Modules can be distributed to run on different machines on a network.

2.1.4 apE

apE (animation production Environment) is also a general-purpose visualization system [19], similar to AVS in many respects. It has a visual graph editor that allows the user to build a pipeline by dragging and clicking (see Figure 2.1 for an example of apE's pipeline "editor" `wrench`). The pipeline must be a DAG.[1] Each module can be connected to other modules and it is triggered by the availability of data.

apE uses the data-driven execution model. Once a pipeline is started, each module in the pipeline runs autonomously. Frame markers encapsulating each set of data act as triggers for the data-driven modules.

apE has a high-level data description language called *Flux*. All the data types in apE, such as geometry, image, and multi-dimensional data, are defined in Flux. In fact, all the data that flows between modules, including control information is written in Flux. New data types can be defined using Flux, so apE can be easily extended to new applications.

As with most other systems surveyed here, data flows in apE are transmitted and processed in large chunks. For example, the whole frame of data, which may be a large 3D array, has to be received by a module before it can be processed. AVS uses the same approach. Although the program structure of receiving, processing, and generating data is clear and simple when data is handled this way, it will inevitably result in large volumes of intermediate data in main memory.

2.1.5 IBM Data Explorer

The IBM Data Explorer [43, 44] also uses the dataflow execution model, as well as parallel processing capability provided by its POWER Visualization Server. Users can construct "visual programs" using the graphical user interface called the *visual program editor*. Each module in the visual program represents a callable function. A function must be a pure function with no

[1] There is one exception when using the `loop` module to construct a loop.

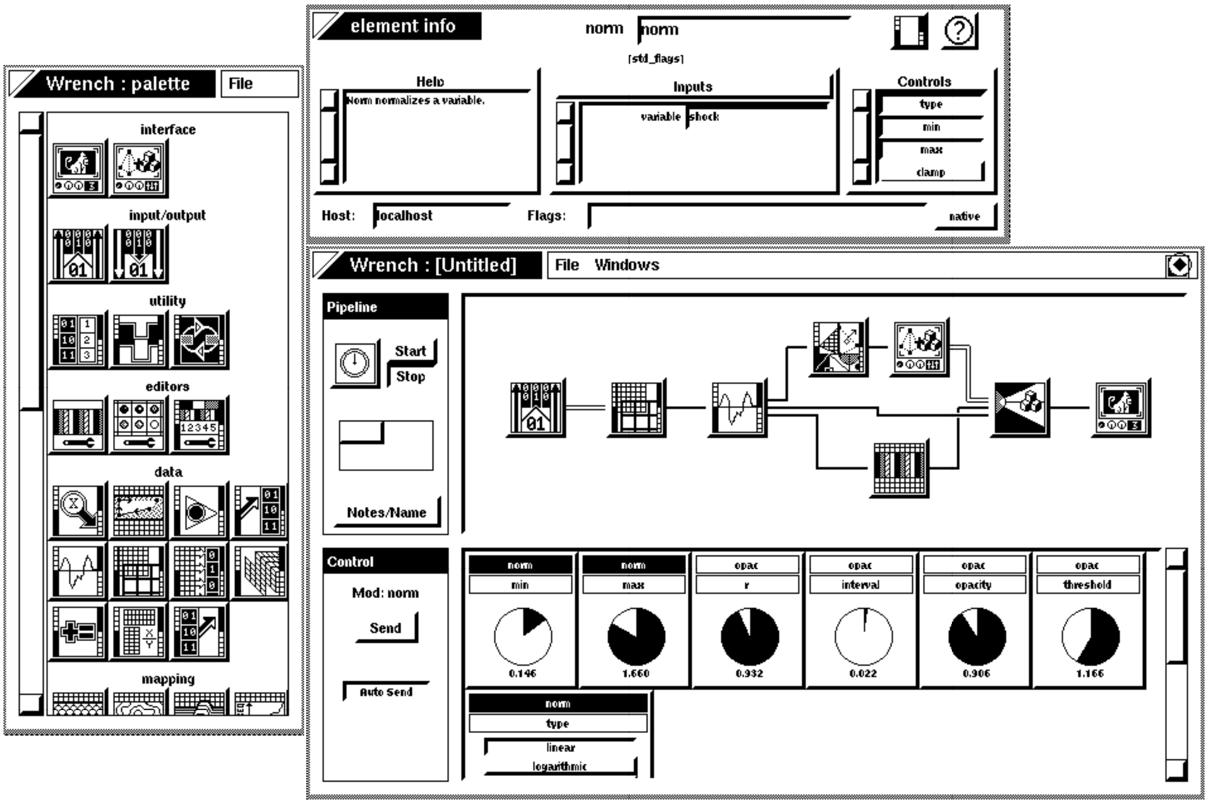


Figure 2.1: The apE pipeline construction and execution tool `wrench`. There are three separate windows in this figure. The *pipeline window* in the bottom contains an editing area for constructing the pipeline and a control area for interactive manipulation of parameters at run time. The *palette window* on the left presents an array of the modules available for pipeline construction. A pipeline is constructed by dragging each individual module into the editing area of the pipeline window and then connecting them. The *info window* on the top presents information about a particular module and allows the user to change it.

side effects, and it has a standard calling interface resembling the input and output arcs of a node in a dataflow graph. The visual programs are translated into the scripting language which is interpreted by an *executive*. The scripting language is a fairly complex one allowing assignments, function calls, macro definitions and limited arithmetic computations. The data model explicitly defines primitive naming mechanisms and scoping rules. Unnamed data blocks can be referenced by byte offsets. The complex data model is carefully designed to allow maximum amount of sharing between data sets. Byte-offset reference is error prone, inflexible, and only suitable for shared memory communication model. The macro mechanism is a distinctive feature of the IBM Data Explorer, which allows structural visual program construction.

The executive of the Data Explorer is a program that contains all the functionality provided by the Data Explorer. If a user adds a new module, he would have to link the new module function with the library and gets a new version of the executive. The software architecture limits the distributed capability of the Data Explorer. The major speedup is gained by parallel processing provided by the expensive POWER Visualization Server. Parallel processing in Data Explorer is aided by partitioning a dataset into spatially local partitions and creating multiple tasks that share the same virtual space.

2.1.6 IRIS Explorer

The IRIS Explorer from SGI [86, 87] is the latest visualization package following AVS and apE's one-process-per-module philosophy. It provides many different user interface tools to minimize programming in constructing an application. The *map editor* allows the user to compose a "map" consisting of modules connected into a network. There is no central controller in the map editor that schedules the execution of the modules. Topological information is obtained by each individual module before they are started. The DataScribe is a data conversion utility that allows the user to define templates for imported and exported data. A template is essentially a graphical representation of the syntax of the data and the DataScribe generates the parser on the fly. But the DataScribe can not describe optional parts of a template, or linked lists. This method may be used for simple data format but may be hard for describing complex ones. Another visual editor provided by IRIS Explorer is the *module builder*, which assists users to modify existing maps and build new ones. It adds a *control wrapper* and an optional *data wrapper* to the computational function in each model. The control wrapper is responsible for receiving data and initiating the computational function. The data wrapper performs data conversion from input data to actual parameters passed to the computational function. The module builder is essentially a tool for people who don't know how to program in C and how to use `make` files. Another feature is Shape, an array manipulation language with a C-like syntax, interpreted by an interpreter. It is intended to be used as a means of fast prototyping.

2.1.7 Khoros

Khoros (previously named Chorus) is a software development environment that consists of the user interface specification, a code generator, and a visual language called Cantata, among many other tools that are built on top of these [81]. Khoros is primarily used for image processing, signal processing and numerical analysis. Cantata is a dataflow visual language. It provides not only variables and expressions, but also operators such as if-then-else and merger. The scheduler is responsible for scheduling the "glyphs" for execution. Each glyph may represent an executable program or an operator provided by the visual language. Current implementation of the scheduler does a sequential scheduling: at any moment only one block of data can be generated and sent down an arc. When parallelism is introduced, the semantic definition of the conditional and merger operators needs rigorous treatment to prevent the race condition. Another distinctive feature of Cantata is the visual hierarchy it provides. Visual procedures can be constructed similar to the way subroutines are built in conventional languages.

2.1.8 RIVERS

The RIVERS (Research in Interactive Visualization EnviRonmentS) project at the National Center for Supercomputing Applications (NCSA) was aimed at providing an interactive visualization and simulation steering software [31]. RIVERS software provides the following features for a distributed visualization software environment: flexibility of configuration (distributing computational tasks among different hosts), capability for high-performance data transfer, and support for the efficient pipelining of operations on the data. Its software architecture uses modular design and distributed execution model. The communication abstraction port uses PVD (Per Vertex Data) for communications between modules. The data flow is abstracted as a stream of PVD structures, each containing an XYZ location, a 3D vector, and an RGB color value. The RIVERS project attempted to solve the dataflow inefficiency problem at a low level [32]. Communicating modules read from or write to a moving window that is advanced through the data set. The window size is determined by the communication software implementing the port abstraction and thus is not known a priori to the modules. Each module must account for cases in which the window size is too small or too large. The limitation of RIVERS method is that it is specified at a low level and it only handles one data type, i.e. PVD structures. The PVD structure limits the RIVERS software to handle geometric data only. It is not appropriate to be used for other data types, such as multi-dimensional arrays, grids, images, etc. RIVERS software is so far the only system that tackles the inefficiency problem of dataflow visualization systems by supporting fine-grain dataflow computation.

2.1.9 DataVis

DataVis was developed to be a visual programming language for scientific visualization applications [37]. Unlike other visualization systems that provide a special-purpose, visual-oriented language at the user-interface level but use a textual language at the module implementation level, DataVis provides a consistent and unified visual programming paradigm at both levels. It uses boxes and links to represent functions and data paths respectively. Like other systems, it provides a library of predefined programs relating to data processing and visualization. Besides it also provides language constructs such as iterations, conditionals, procedural abstractions, and higher-order functions. These features distinguish DataVis from other visualization systems and make it a powerful general-purpose visual language.

2.1.10 Other Systems

There are other systems like ConMan [33] and FAST [29] that are similar in nature to the ones mentioned above. Visual programs in ConMan use files for inter-module data communications. Each component in the visual program is a separate Unix process.

FAST is also a modular system but does not have a graphical editor to build up a dataflow diagram. Instead it provides a global set of registers, each of which contains a multi-dimensional data set. Individual modules can be invoked to operate on these registers. FAST heavily uses shared memory for data communications between modules.

2.1.11 Summary

Our analysis of previous systems has focused on the execution model, data language, communication mechanisms, and other features that facilitate application and system development. Table 2.1 compares the following features provided by these systems:

1. Execution Model

    - Is there a central controller?
    - Scheduling: Is it data driven, demand driven, or mixed?
    - Does the system support distributed/parallel processing?

2. Data Model

- What types of data are supported?
- How is the data moved between modules (data granularity)?
- Are the data structures hierarchical or flat?

3. Communication Mechanisms

    - What are the communication mechanisms: shared memory, sockets, pipes or files?
    - Does a user have the right to choose which mechanism to use?

4. Grouping

    - Does it allow grouping?
    - Does it allow hierarchical grouping?

5. Scripting

    - Is there a scripting language? In other words, can a pipeline run in a batch mode?

6. Extensibility

    - Can new modules be added?
    - Can new data types be added?

Many of the systems surveyed here called themselves visual languages, but, except for DataVis, they do not have the power of a general-purpose visual programming language, in the sense that new modules can not be built using only constructs provided by the language. New modules are built using conventional textual languages. We call these systems *special-purpose visual languages*. In Hils [36] these systems are called *module connection systems*. We can use the application domain or language features to distinguish a general-purpose from a special-purpose visual language. A more rigorous criterion is to see if a language is Turing equivalent – a general purpose visual programming language should have the same computing power as a Turing Machine. Many of the visualization systems are now starting to offer language constructs that are provided by general-purpose visual languages. For example, Khoros provides a conditional construct (if-then-else) and an iteration construct (while-loop) [109], and IBM Data Explorer provides procedural abstractions [44]. Since each system uses a different

	GRAPE	FRAMES	AVS	apE	DX[a]	IX[b]	Khoros	RIVERS	DataVis
Execution Model									
node executable	no	yes	yes	yes	no	yes	yes	yes	yes
executive	no	no	yes	no	yes	no	yes	no	yes
scheduling[c]	DM	DT	DM	DT	MX	DT	MX	DT	DT
distributed	no	yes	yes	yes	no	yes	no	yes	no
Data Model									
data types[d]	G,I	A	A	A	A	A	I	G	A
granularity[e]	C	C	C	C	M	C	C	F	C
hierarchical	?	yes	no	yes	yes	yes	no	no	no
Communications									
mechanism[f]	F,P	P	S,M	F,P,S	S	S,M	F,M	F,P,S,M	V
user-selectable	no	no	no	no	no	no	yes	yes	no
Grouping									
allowed	yes	no	no	no	yes	yes	yes	no	yes
hierarchical	yes	no	no	no	yes	yes	yes	no	yes
Scripting	no	yes	yes	yes	yes	yes	yes	yes	no
Extensibility									
new modules	yes	yes	yes	yes	yes	yes	yes	yes	yes
new data types	?	yes	yes	yes	no	yes	yes	no	yes

[a]IBM Data Explorer.
[b]IRIS Explorer.
[c]DT: data-driven; DM: demand-driven; MX: mixed (both data-driven and demand-driven).
[d]G: geometry; I: image; A: any.
[e]F: fine-grain; M: medium-grain; C: coarse-grain.
[f]F: files; P: Unix pipes; S: Unix sockets; M: shared memory; V: shared virtual space.

Table 2.1: Feature comparisons of dataflow visualization systems.

name for the dataflow diagram representing a visualization pipeline[2], we'll use the these terms interchangeably in the following discussions.

AVS, apE, and IBM Data Explorer provide scripting capabilities that are extremely useful when a user is debugging a new module or is experimenting with data preprocessing when no graphical display or interface is needed. The apE also provides different ways to monitor the execution of a dataflow network, one of which is to use a monitor module that can be connected to any module to examine the output data flow from that module.

Grouping mechanisms (macros, or procedural abstraction) are provided by Khoros, IRIS Explorer and IBM Data Explorer. They help users make higher level abstractions of the visual

[2]For example, IRIS Explorer uses 'map', IBM Data Explorer and Khoros use 'visual program', AVS uses 'network', and apE uses 'pipeline'

programs they already built. Grouping is a very important feature that will make the visual programming paradigm work better. Grouping is called procedural abstraction in DataVis.

Both GRAPE and IBM Data Explorer represent a visual program as a single executable program. GRAPE uses a compiler that translates a visual representation of the dataflow network into a program. Different GRAPE networks generate different executable programs. IBM Data Explorer goes a step further to eliminate the compilation step by compiling *all* the "modules" as functions into *one* program. An IBM Data Explorer network may contain only a few functions that have been compiled into that big executable program. This approach taken by GRAPE and IBM Data Explorer may work well on a single machine but does not extend itself naturally to a more general distributed environment. These system are also hard to extend – it requires recompiling every time some new modules are added.

At the other extreme is the one-process-per-module approach in which each module is implemented as a Unix process. The systems based on one-process-per-module approach are natural extensions of the Unix utilities and pipelines, only that they are more powerful (multiple input and output, graphical user interfaces, etc), deal with more complex data types, and inevitably grow very fast in size when more and more modules become available. The disk space taken up by these modules sometimes become prohibitively large on some systems [101].

The previous systems demonstrated that the dataflow model and visual programming paradigm is an effective way of building user friendly, extensible, generic software systems. In all of the systems mentioned above, each module in a visual program represents a function that transforms input data objects into output data objects. The implementation of a module can either be a subroutine within a program or an executable program. But they all share the property that every module performs one data processing function, written by the module writer, on the input data objects. The three steps of data input, data processing, and data output of one module never overlap in time. This is a straightforward application of the pure dataflow computing model of the computer architecture to the software system.

The straightforward implementation of the pure dataflow model presents a simple programming interface to application programmers (module writers). Most of the above-mentioned systems share this property. But it is often the case that convenient programming models do not necessarily lend themselves to efficient implementations. The current visualization systems suffer from the problem of inefficiency in terms of memory usage and CPU utilization, which can be solved by using a fine-grain model.

2.2 Attribute Grammars

An *attribute grammar* (AG) is a context-free grammar (CFG) augmented by attributes, functions defining values of attributes, and conditions over attributes [14, 21, 53]. An attribute grammar can be used to specify a translation of a language using attributes associated with syntactic components. This thesis treats scientific visualization as a translation from one data language to another and presents an AG-based approach to building modules.

Attribute grammars are chosen to be both the specification language and the implementation tool for fine-grain modules for the following reasons:

- AGs provide a natural way to specify fine-grained communications and computations, and there are automatic module generation tools to convert the specifications into programs. A module is essentially a language translator, so module generation tools can help module builders write modules the same way compiler generators help compiler writers write compilers.

- There are well-defined algorithms in the attribute grammar theory to determine whether an attribute grammar is an L-AG so that the attributes can be evaluated in one pass [2, Chapter 5]. This is a formal way to verify that a module is indeed "fine-grain". Without it, we have to do it manually.

- AGs have been successfully applied to a number of important application areas in computer science.

One application of attribute grammars is language based editors (LBE) [82, 48]. In LBEs a program is represented as an attributed tree with consistent attribute values. When the program is modified, the editor will update attributes that are affected by the editing and keep them consistent. A related research area is incremental graph evaluation [39, 41] where incremental computations are based on graphs instead of trees as in LBEs. One basic assumption in LBEs and incremental computing is that there is always an attributed graph/tree to work on. The

SDTM model proposed here avoids building the entire parse tree during data parsing and translation.

Attribute grammars have also been used in automatic user interface generation [40] where user interfaces, semantic feedbacks are generated by techniques derived from incremental attribute evaluation; in natural language processing [58] where attributes are used to ensure the agreements between parts of a sentence that can not be represented by context free grammar rules; and in VLSI design [46] where a circular attribute grammar is used to represent hierarchical VLSI design. The most thorough studies on the attribute grammars are in the area of compiler construction [2, 8, 49, 52, 53, 103]. A large number of automatic compiler construction systems based on attribute grammars have been built [20, 30, 50].

It must be stressed that AG is only one way to implement fine-grain pipelining. Other models that can be used to implement fine-grain modules are discussed in Section 2.3. It needs to be seen whether the AG-based SDTM approach will be practical from the software engineering point of view. But it is our belief that SDTM provides the best solution.

The central idea of AG is attribute definitions and attribute evaluations. Each grammatical symbol of a language is associated with a set of attributes. The values of each attribute occurrence is defined in terms of zero or more other attribute occurrences within the same production rule. Attribute evaluation is the process of determining the meaning of a parse tree decorated with attributes. Assuming that the attribute dependencies don't form cycles in any possible parse trees, i.e. the AG is well-formed, any evaluator that sorts the attribute instances in a topological order will be able to evaluate the attributed parse tree. Various attribute evaluators and classes of attribute grammars have been examined in many research papers.

Most of the previous studies did not treat *defining* attributes as an important and integral part of an attribute grammar. For our purpose the attribute specification is an important process that will affect the class of the grammar being defined and the choice of the attribute evaluator being used. Bochmann [8] gives an example of how to modify the syntax of a block-structured language that potentially requires an infinite number of passes: The identifier declarations of Algol can be placed anywhere inside a block and they are visible throughout the whole block; by changing the syntax such that all identifier declarations of a block are located at its beginning, we obtain a new language that needs only a single left-to-right pass. The modification does not change substantially the structure of a language and its expressive power, but is significantly important in our stream-like implementation of a dataflow system. There is also an example in our implementation that demonstrates the same idea. In general the efficiency of the evaluator is increased sometimes at the expense of the flexibility or even the expressiveness of the language being defined.

A fine-grain module in a visualization system should be able to do one-pass, left-to-right parsing for a large class of data types. In the case of scientific visualization software, it is important to have a space efficient parsing scheme as well as a time efficient one.

Attribute evaluators can be divided into two categories: dynamic and static evaluators. A *dynamic evaluator* constructs the attribute dependency graph and decides the evaluation order of the attribute occurrences based on their topological order. A *static evaluator* determines the evaluation order for the attributes based on the analysis of the production and semantic rules of the AG. The evaluation order of a dynamic evaluator is determined at run time while that of a static evaluator is determined at evaluator construction time. Generally speaking static evaluators are more efficient than dynamic evaluators. In some cases even the parse tree does not need to be completely built by a static evaluator in order to evaluate all the attributes.

Among the attribute evaluation systems that have been built in the past, some of them accept arbitrary well-formed attribute grammars while others are restricted to special classes of attribute grammars in order to be more efficient. The properties of attribute dependencies determine the classes and the dependency relations can be very complex in the general case. This is the common source of the difficulty of finding an evaluation algorithm which is both general and efficient. A class of attribute grammar chosen for a particular application ought to:

1. be decidable in polynomial time,

2. have an evaluation algorithm efficient in space and time, and

3. be as large as possible.

Various restrictions put on AGs form a hierarchy of classes. The characterization problem is to decide what class a particular AG belongs to. Below is a general class hierarchy of AGs from [49].[3] They are listed in descending order of expressive power. The complexity of characterization test is indicated in the parentheses. The classes defined below the dashed line

[3] More complete classifications of attribute grammars based on tree walk can be found in Deransart [17] and Engelfriet [22].

AG		general attribute grammars
WAG	(exponential)	well-defined attribute grammars
ANCAG	(polynomial)	absolutely noncircular attribute grammars
OAG	(polynomial)	ordered attribute grammars
------------------		
m-APAG	(polynomial)	attribute grammars evaluable in m alternating passes
n-PAG	(polynomial)	attribute grammars evaluable in n passes from left to right
1-PAG $\equiv$ L-AG		attribute grammars evaluable in 1 pass from left to right,
	(linear)	L-attributed grammars
S-AG	(linear)	S-attributed grammars

Figure 2.2: Classification of attribute grammars

are based on a tree walk strategy defined a priori, whereas those above the line are based on strategies computed from the attribute dependencies.

Of particular interest in the class hierarchy defined in Figure 2.2 is the L-AG [8, 60]. An *L-Attributed Grammar* is an attribute grammar in which each inherited attribute of $X_j, 1 \leq j \leq m$, on the right side of any production $p : X \rightarrow X_1 X_2 \cdots X_j \cdots X_m$, depends only on

1. the attributes of the symbols $X_1, X_2, \cdots, X_{j-1}$ to the left of X_j in p, and

2. the inherited attributes of X.

In other words, the semantic functions defining the inherited attributes of a given symbol in a production can use attributes of symbols to the left of the given symbol, but not attributes of symbols to the right. For L-AGs, attribute evaluation can be done in parallel with LL(k) parsing, thus no parse tree needs to be built explicitly. This property is important to our streaming approach in which data buffering is either totally eliminated or being kept to the minimum. Furthermore, the characterization test and attribute evaluation for L-AGs are very efficient.

L-attributed translations with an underlying LL(1) CFG can be performed using a recursive-descent parsing method called *predictive parsing*. In this method there is one procedure for recognizing each nonterminal in the grammar. The lookahead symbol determines which production to use for that nonterminal. The procedure has a parameter for each attribute of the corresponding nonterminal. Inherited attributes are input parameters and synthesized attributes are output parameters. Suppose X is a nonterminal and $a_1, a_2, \ldots, a_m$ are X's at-

Algorithm 2.1 (Predictive parsing with simultaneous attribute evaluation)

```
procedure X(a_1, a_2, ..., a_m)
begin
    { Let t be the current input token }
    if there is a production X → α such that t ∈ FIRST(α) then
    begin
        { Let α = X_1 X_2 ··· X_n }
        for i := 1 to n do
        begin
            if X_i is a nonterminal then
            begin
                evaluate inherited attributes of X_i;
                X_i(b_1, b_2, ..., b_{m_i});    { suppose X_i has attributes b_1, b_2, ..., b_{m_i} }
            end
            else { X_i is a terminal symbol }
                match(X_i);
        end
        evaluate synthesized attributes of X;
        return
    end
    else if t ∉ ⋃_{j=1}^{k} FIRST(α_j) and t ∈ FOLLOW(X) then
        { where X → α_1 | α_2 | ··· | α_k, and there is a production X → ε}
        begin
            evaluate synthesized attributes of X;
            return
        end
        else
            error;
end.
```

tributes. FIRST(α) is the set of first terminal symbols in any of the strings generated by α. The procedure that recognizes X is described by Algorithm 2.1. A call to the procedure S(...) corresponding to the start symbol S parses and translates an entire sentence. An iterative version of the parse-time attribute evaluation for L-AGs with underlying LL(1) CFG can be found in Akker [73].

The L-AG formalism is ideal for representing stream processing. However, L-AGs are too restrictive and have weak expressive power. (See the class hierarchy of AGs in Figure 2.2) Ordered attribute grammars (OAGs) provide more expressive power while still being efficient. The class of OAGs is sufficiently large for programming language definitions [49, 103]. OAGs have the following properties: (1) for each symbol of the grammar a partial (or linear) order over the associated attributes can be defined such that in any context of the symbol the attributes are evaluable in that order, and (2) a visit sequence can be computed for each production of the OAG at the attribute evaluator construction time. From these two properties one can analyze lifetimes of attribute instances and subtrees and design a memory efficient evaluator that will keep only part of the parse tree and attributes that are needed in further attribute evaluations. Both L-AGs and OAGs have been widely used to build compilers and other systems [9, 30, 50, 106]

Deransart [17] pointed out that restricting the class of accepted AGs improves the size of the evaluator but brings no benefit regarding their efficiency in time and space, unless we go down to very restrictive classes such as L-AGs. For this reason we can choose from one of the two versions of the attribute evaluator for each module in our system. The choice is made at the evaluator construction time: if the grammar passes the L-AG characterization test, the more efficient evaluation algorithm for L-AGs is used, otherwise the algorithm for OAGs is used. The characterization tests for AGs and OAGs can be found in Aho [2] and Kastens [49] respectively.

The extravagant consumption of storage space has been noted as the major drawback of AG-based systems [51, 82]. Attribute storage optimization is of primary concern to our research, because of the large amount of data that a visualization module has to deal with. The storage optimization contains techniques including variable globalization, variable deallocation, subtree deletion, and others [17, 23, 47].

2.3 Other Models

There are many theoretical frameworks for modeling complex and distributed systems [16, 38, 67, 79]. A Petri net, for example, is an abstract model for concurrent systems. It addresses discrete event dynamics systems, such as process control systems, communications protocols, and real-time systems. These systems are usually complex systems in terms of states and state transitions. The purpose of using Petri nets is to obtain analytical results on some properties of the modeled systems, such as boundedness and liveness. Petri nets and their derived models are comparable to the Chomsky language hierarchy. They define the syntax of languages but do not provide a semantic framework in which data transformations can be done. Almost all of the abstract models assume that data communications are atomic actions. This inadequacy of modeling data transmission and transformation has caused the memory inefficiency problem when visualization systems are built on the pure dataflow model.

Denotational semantics deals primarily with translations [84]. It requires definitions of the semantic domain, operations, etc. While these are necessary for formal specification and verification of the translations, they are an extra level of abstraction that is irrelevant to solving the problems we are addressing here. On another hand, research in attribute grammar has generated techniques that can help us build efficient parsers.

There are languages that provide stream processing capabilities [91, 1, 66]. A stream is a sequence of values (or objects), all of the same type, that are passed in succession, one-at-a-time between program modules. A stream could be infinitely long and their components can be accessed before they are fully constructed. Only a limited number of operations such as *concatenate, head, tail*, are provided by the languages. Streams provide a natural way of expressing the flow of data between modules. In theory, stream languages can also be used to implement fine-grain modules, but it is not clear whether they present a better choice than the syntax-directed method due to the limited number of stream operation functions they provide, and their performance and availability on different hardware platforms.

One of the goals of our research is to find ways to express computations mixed with data parsing and apply it to visualization systems. Syntax-directed approach seems to be the most appropriate way. Parsing and translation are two important aspects of this type of computation.

Chapter 3

The Syntax-Directed Data Transformation Method (SDTM)

In the SDTM model, fine-grain pipelines consist of fine-grain modules, each of which is regarded as a translator. A translator can be built either manually or automatically. Here we choose an automatic method based on attribute grammars. Attribute grammars provide a natural way to associate computations with pieces of data. Attribute grammar theory also provides a formal method to prove that certain algorithms are one-pass streaming ones. The analytical performance analysis demonstrates that the approach of syntax-directed definition and the stream-like implementation is more efficient and yields better performance than existing coarse-grain systems.

New problems concerning chunk sizes and lookahead tokens arise when multiple modules (translators) are connected into a pipeline. This chapter uses the properties of SDTM pipelines to develop a theorem to solve these problems.

3.1 Coarse-Grain and Fine-Grain Dataflow Approaches: a Comparison

The approach used in most previous systems (except the RIVERS software) to move data between modules can be classified as the coarse-grain approach, which treats an entire (one frame of) data set as an indivisible token when transmitting it. A frame typically consists of all the data needed for a module to perform its task. A module repeatedly runs the same task on each input frame. The program structure of a typical module can be described by the following pseudo code:

```
main()
{
    while (true)
    {
        wait for the trigger signal
        read in the whole frame
        call the computation routine compute()
        write out the whole frame
    }
}

compute()
{
    check the validity of the data
    do necessary castings
    data processing
}
```

The problems with the coarse-grain approach are:

- The processor utilization is low because other modules may be waiting while one module is processing a large chunk of data [29];

- Because each module is buffering its entire frame, the total memory requirement is very large [10]. Continuous swapping of running processes by the operating system has been observed on systems with even moderate memory size [104], and it is also measured and shown in Table 6.3 in Chapter 6.

- The startup time of the pipeline is long. This also delays error detection if an error occurs in the data set.

The coarse-grain approach reflects the traditional firing rule in a pure dataflow system which mandates that all required inputs be available on input arcs and that the output arc have space for the result [98]. Landry [56] modified the conditions for the firing rule and proposed the Firing Semantic Set (FSS) which leads to more general and flexible firing condition specifications. An

FSS is a specification of the set of input and output configurations that allow a node to be enabled.

In the SDTM model, firing conditions for each module are embedded in the semantic functions and are at least as powerful as the FSS in terms of expressiveness.

The idea of the SDTM model is

- to keep the minimum amount of data within each module so that the total amount of data that exists in the pipeline at any moment is also reduced,
- to start processing data as soon as they become available so as to increase the utilization of the processors, and
- to generate output as early as possible so that the startup time of the pipeline can be reduced.

The result is a dataflow system with continuous data flow and minimum time delay at each node. The schematic structure of a module using SDTM can be shown by the following pseudo code, while a detailed example of a module using SDTM is discussed in Section 3.2:

```
main()
{
    while (true)
    {
        read a data token from input
        if (no production rules can be reduced) then
            push parser state and token onto the stack
            go to another state
        else
            pop up states and tokens
            call semantic functions as tokens are reduced
            go to a new state
        endif
    }
}
```

The SDTM approach has the following advantages over the coarse-grain approach:

- The processing speed and throughput of the visualization pipeline is increased, not through the use of native binary data format, but through granulating data.

- By minimizing the idle time, the utilization of the processors is increased.
- The startup time of the pipeline is decreased.
- It provides a formal method for implementing stream-like dataflow computations in a software system. Unix data processing utilities are based on the concept of character streams. Utilities can be connected in a linear fashion to make powerful tools. Programs like `yacc` and `lex` are used to facilitate building these utilities. Similar needs arise in visualization systems, too. But the data types in a visualization system are far more complex and larger in size.
- The stream-like implementation maintains a continuous flow of data in the visualization pipeline, reducing the intermediate buffering to the minimum amount. This leads to the efficient use of storage space so that it is possible for machines with limited memory and disk space to be able to run visualization programs that would have required a supercomputer merely for its large memory.
- Errors can be handled systematically. They can be detected or sometimes recovered as early as possible.

The analytical analysis of the speedup and memory savings of the SDTM approach over the coarse-grain approach is presented in Section 3.3.

3.2 SDTM: A Fine-Grain Dataflow Model

An SDTM system consists of the following components:

1. a data language suitable for stream processing,
2. fine-grain modules that are translators generated from attribute grammars, and
3. pipelines composed of fine-grain modules that are connected in the form of directed acyclic graphs and operate according to two rules: the **Buffer Size Rule** and the **Lookahead Rule**.

3.2.1 The Stream Data Language

In the SDTM model, each module in a dataflow visualization system can be viewed as a small language translator, and the data streams can be specified by a context-free grammar. The syntax of the source language is a predefined context-free language and the semantics is determined by the functionality of the module. The attribute grammar defining the source language is then classified and an appropriate attribute evaluator is chosen for the translator. A module writer only needs to provide the semantic functions in a declarative way. All the rest of the work can be automated.

The *Flux* data specification language of apE is used as the basis for studying the SDTM approach [72]. Flux is a self-describing, easily extended, and highly portable data language. It allows hierarchical definitions of complex data types, much like a complex data representation in a structured programming language. Some extensions will be made to Flux to explore hierarchical nature of the language and to facilitate the development of fine-grain algorithms. A full description of the Flux language can be found in Appendix A.

An important element of Flux is the *group*. A group is a collection of any Flux entities surrounded by begin and end lines. Groups are used to encapsulate series of fields, arrays, and other groups. Examples include variable groups, image groups, grid groups, object groups, etc. A *variable group* represents a named block of data of a certain type. Following is an example of a Flux *variable group* that represents a multi-dimensional array:

```
begin variable "foo"
   notes[1] string ascii
       "This is a variable group"
   ^spatial_ref grid:"foo.grid"
   range 0 7
   data[2][2][2] float ascii
   0.1
   0.2
   0.3
   0.4
   0.5
   0.6
   0.7
   0.8
end variable "foo"
```

A context-free grammar can describe its structure:

variable_group → *begin header array end*

begin → "begin variable" **id**

end → "end variable" **id**

header → *field header* | ϵ

field → *notes* | *spatial_ref* | *range*

notes → "notes" *count* **type format** *n_list*

count → "[" **inum** "]"

n_list → **string** | **string** *n_list*

spatial_ref → "spatial_ref" *reference*

reference → *variable_group* | *grid_group*

range → "range" **inum inum**

array → "data" *dims* **type format** *data*

dims → "[" **inum** "]" | "[" **inum** "]" *dims*

data → **num** | **num** *data*

In the above example, **type, format, id, inum, num, string** are tokens, and any names in double quotes are literals. The data element **type**'s are *string, float, int, color*, etc. The data representation **format**'s are *ascii, native binary,* and *xdr*[1].

Since the *variable group* allows the *spatial reference* to be another variable group, i.e. production rules for the variable group allow the generation of nested structures, a context-free grammar is required to describe various group structures.

3.2.2 SDTM Modules

A module can have a single input port (e.g. in a linear pipeline) or multiple input ports (e.g. in a general dataflow graph). Since linear pipelines are the subject of this thesis, multiple input will not be handled by the SDTM modules. But they could be added later, and a high-level description of the scheme is presented in Section 3.2.2.2.

[1] XDR (eXternal Data Representation) is defined by Sun Microsystems to provide a way of describing arbitrary data structures in a machine-independent fashion. It converts between C data types and an external bit-string representation.

3.2.2.1 Single Input Port

An SDTM module is essentially a syntax-directed translator. In a dataflow environment, a module can be viewed as a black box that performs a certain function transforming the input data to the output data. The attribute grammar specifies the parsing and translation of the input data. If the AG is L-attributed, then a fine-grain module can be generated that implements parse-time attribute evaluation described by Algorithm 2.1. Modules can be generated using standard compiler generation tools such as `yacc` [45] or `Eli` [30].

Granularity can be varied using different grammar rules. Following the sample variable group "foo" used in the last section (Section 3.2.1), the granularity for a 3D data array could be a single data element, a 2D slice, or an arbitrary chunk. The rules for defining the input data with different granularities are, respectively, the following three:

$$\textit{data} \rightarrow \textbf{num} \ \{ \text{ semantic\_function}(\cdots) \ \} \mid \textbf{num} \ \textit{data} \ \{ \text{ semantic\_function}(\cdots) \ \}$$
$$\textit{data} \rightarrow \textbf{slice} \ \{ \text{ semantic\_function}(\cdots) \ \} \mid \textbf{slice} \ \textit{data} \ \{ \text{ semantic\_function}(\cdots) \ \}$$
$$\textit{data} \rightarrow \textbf{chunk} \ \{ \text{ semantic\_function}(\cdots) \ \} \mid \textbf{chunk} \ \textit{data} \ \{ \text{ semantic\_function}(\cdots) \ \}$$

As a complete example, the syntax-directed definition and translation scheme of a module `norm` is given below. The module `norm` takes a named scalar or vector variable as input, normalizes the data values to between 0 and 1, then outputs the normalized variable, leaving the spatial reference, if any, unaffected. The minimum and maximum data values are provided by the user at run time. The normalization function, which may be linear, exponential, logarithmic or user-supplied, is also bound at run time. As an example, the variable group shown below is the output from `norm` taking the sample variable data set in Section 3.2.1. The normalization function used here is a linear one.

```
begin variable "foo"
   notes[1] string ascii
       "This is a variable group"
   ^spatial_ref grid:"foo.grid"
   range 0 7
   data[2][2][2] float ascii
   0.0
   0.14
```

```
    0.29
    0.43
    0.57
    0.71
    0.86
    1.0
end variable "foo"
```

The syntax-directed definition for module `norm` is presented in Table 3.1. The translation scheme is given below:

var_group → *begin*	{ *var_group*.name = *begin*.lexeme }
header	{ *var_group*.attr = *header*.attr }
array end	{ if (*end*.lexeme == *begin*.lexeme) send_end(*end*.lexeme) else { send_end(*begin*.lexeme) warning() }}
begin → *"begin variable"* **id**	{ send_begin(**id**.lexeme) *begin*.lexeme = **id**.lexeme }
end → "end variable" **id**	{ *end*.lexeme = **id**.lexeme }
header → *field*	{ *header*.attr = *field*.attr }
header → *field* $header_1$	{ *header*.attr = union(*field*.attr, $header_1$.attr) }
field → *notes*	{ *field*.attr.notes = *notes*.lexeme }
field → *spatial_ref*	{ *field*.attr.ref = *spatial_ref*.ref }
field → *range*	{ *field*.attr.range = *range*.range }
notes → "notes" *count* **type**	{ *notes*.type = **type**.val }
format	{ *format*.type = **format**.val }
n_list	{ send_notes(*n_list*.count, **type**.val, **format**.val, *n_list*.lexeme) *notes*.count = *n_list*.count *notes*.lexeme = *n_list*.lexeme }
count → "[" **inum** "]"	{ *count*.val = **inum**.val }
n_list → **string**	{ *n_list*.lexeme = **string**.lexeme *n_list*.count = 1 }

n_list → **string** n_list_1	{ *n_list*.lexeme = append(**string**.lexeme, n_list_1.lexeme) *n_list*.count = n_list_1.count + 1 }
spatial_ref → "spatial_ref" *reference*	{ send_ref(*reference*.name) *spatial_ref*.ref = *reference*.name }
reference → *var_group*	{ *reference*.name = *var_group*.name }
reference → *grid_group*	{ *reference*.name = *grid_group*.name }
range → "range" $\mathbf{inum}_1$	{ *range*.range.min = $\mathbf{inum}_1$.val }
$\mathbf{inum}_2$	{ send_range($\mathbf{inum}_1$.val, $\mathbf{inum}_2$.val) *range*.range.max = $\mathbf{inum}_2$.val }
array → "data" *dims* **type**	{ *data*.type = **type**.val }
format	{ *data*.format = **format**.val send_array(*dims*.count, **type**.val, **format**.val) if (*data*.count != *dims*.count) error() }
data	
dims → "[" **inum** "]"	{ *dims*.count = **inum**.val }
dims → "[" **inum** "]" $dims_1$	{ *dims*.count = $dims_1$.count × **inum**.val }
data → **num**	{ send_data(normalize(**num**.val, *data*.type), *data*.format) *data*.count = 1 }
data → **num**	{ send_data(normalize(**num**.val, *data*.type), *data*.format) $data_1$.type = *data*.type $data_1$.format = *data*.format }
$data_1$	{ *data*.count = $data_1$.count + 1 }

The send_* functions in the attribution rules can be regarded as defining dummy synthesized attributes. Those dummy attributes depend on the attributes passed in as arguments to the functions. Through attribute dependency analysis, it could be verified that the above AG translation scheme satisfies the L-AG conditions stated in Section 2.2. Thus a one-pass translator can be built for the module `norm`.

Production	Semantic Rules
$var\_group \rightarrow begin\ header\ array\ end$	send_end($begin$.lexeme) var_group.name = $begin$.lexeme var_group.attr = $header$.attr
$begin \rightarrow$ "begin variable" **id**	send_begin(**id**.lexeme) $begin$.lexeme = **id**.lexeme
$end \rightarrow$ "end variable" **id**	end.lexeme = **id**.lexeme
$header \rightarrow \epsilon$	$header$.attr = $\emptyset$
$header \rightarrow field\ header_1$	$header$.attr = union($field$.attr, $header_1$.attr)
$field \rightarrow notes$	$field$.attr.notes = $notes$.lexeme
$field \rightarrow spatial\_ref$	$field$.attr.ref = $spatial\_ref$.ref
$field \rightarrow range$	$field$.attr.range = $range$.range
$notes \rightarrow$ "notes" $count$ **type** **format** n_list	send_notes(n_list.count, **type**.val, **format**.val, n_list.lexeme) $notes$.count = n_list.count $notes$.type = **type**.val $notes$.format = **format**.val $notes$.lexeme = n_list.lexeme
$count \rightarrow$ "[" **inum** "]"	$count$.val = **inum**.val
$n\_list \rightarrow$ **string**	n_list.lexeme = **string**.lexeme n_list.count = 1
$n\_list \rightarrow$ **string** n_list_1	n_list.lexeme = append(**string**.lexeme, n_list_1.lexeme) n_list.count = n_list_1.count + 1
$spatial\_ref \rightarrow$ "spatial_ref" $reference$	send_ref($reference$.name) $spatial\_ref$.ref = $reference$.name
$reference \rightarrow var\_group$	$reference$.name = var_group.name
$reference \rightarrow grid\_group$	$reference$.name = $grid\_group$.name
$range \rightarrow$ "range" $\mathbf{inum}_1$ $\mathbf{inum}_2$	send_range($\mathbf{inum}_1$.val, $\mathbf{inum}_2$.val) $range$.range.min = $\mathbf{inum}_1$.val $range$.range.max = $\mathbf{inum}_2$.val
$array \rightarrow$ "data" $dims$ **type** **format** $data$	send_array($dims$.count, **type**.val, **format**.val) $data$.type = **type**.val $data$.format = **format**.val if ($data$.count != $dims$.count) error()
$dims \rightarrow$ "[" **inum** "]"	$dims$.count = **inum**.val
$dims \rightarrow$ "[" **inum** "]" $dims_1$	$dims$.count = $dims_1$.count $\times$ **inum**.val
$data \rightarrow$ **num**	send_data(normalize(**num**.val, $data$.type), $data$.format) $data$.count = 1
$data \rightarrow$ **num** $data_1$	send_data(normalize(**num**.val, $data$.type), $data$.format) $data$.count = $data_1$.count + 1 $data_1$.type = $data$.type $data_1$.format = $data$.format

Table 3.1: Syntax-directed definition for module `norm`.

3.2.2.2 Multiple Input Ports

A more complicated situation arises when a module has multiple input ports. For example, in Figure 3.1 module `onion` has two input ports: one to input a 3D scalar data set, the other to input surface properties to be associated with the iso-surfaces it generates. The following syntax rule describes how to handle two input ports:

$$A \rightarrow B \,\&\, C$$

The & construct means that a B from one input port and a C from another input port form a symbol A.

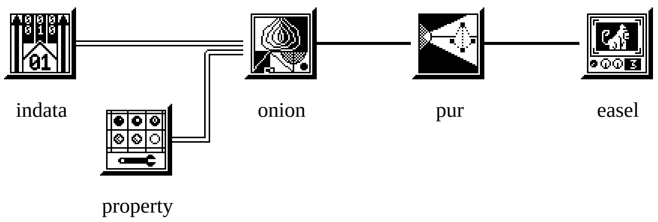


Figure 3.1: A pipeline with a multiple-input module

In GRAPE [68], Nadas added two functions for each node: a *gofor* to deal with its input and a *server* to deal with its output. Every time a node needs a data element, it calls its gofor indicating what is required, and the gofor calls the appropriate server of another node. The server buffers the data produced by the node until they are consumed by other gofors. This implies a demand-driven, synchronous communication scheme.

For a data-driven system such as apE that allows both synchronous and asynchronous communications, we can have multiple, coordinating parsers running at the same time. Two models are found to be suitable in this case. The first is the coroutine model ([13] or [63, pp. 314–317]). In this model, a module consists of several coroutines, each of which is a parser responsible for one input port. Whenever a coroutine needs some data from a certain input port, it resumes the coroutine responsible for that port, and remains suspended until some other coroutine resumes it again.

Another model is a group of light-weight processes, or threads, each responsible for one input port. All the parsers are running simultaneously. One process would block when waiting for data from another process. Processes communicate through shared variables by using proper synchronization mechanisms.

Conway [13] showed that the coroutines can be put together as a one-pass compiler. In contrast, the coarse-grain approach can be loosely considered as a two-pass process (the first pass reading in the whole data set, the second one consuming it). Knuth [54, pp. 190-196] pointed out that a process done by n coroutines can often be transformed into an n-pass process. The inverse is not true, because some algorithms are inherently multipass. A one-pass algorithm is generally faster than a multipass one because it doesn't need to manage intermediate data between passes. The disadvantage of the coroutine model is that there is only one coroutine active at one time. In a multiprocessor system it is desirable to have such high-level computational units to run concurrently on multiple processors, which is the advantage of using the light-weight process model. The light-weight process model provides more concurrency than the coroutine module — a module can consume multiple streams at the same time. The triggering rules for module activation could also be incorporated into the syntax-directed definition.

3.2.3 SDTM Pipelines

A single fine-grain module uses less memory than a coarse-grain module, but it does not run faster. The speedup is made possible only when multiple fine-grain modules are connected into a software pipeline. SDTM pipelines have their own properties due to the new features such as data granularity and syntax-directed translation introduced into the software pipelining. Maximum performance can be achieved if proper methods are used.

SDTM pipelines have the following two properties (see Figure 3.5 for an illustration):

1. Each module runs through cycles of I (input), P (processing) and O (output). Each triplet of IPO forms a cycle. Any phase in a cycle could be null.

2. Modules overlap their IPO cycles in time.

In each cycle, a module processes one chunk of data. The number of chunks a data set is divided into is used later to measure the granularity of the data stream.

The overlapping happens between the output of one module and the input of the next module down the stream. There are two rules that govern the communication between the neighboring modules. The first rule (Rule 3.1) restricts the choice of size of the input and output buffers. The second rule (Rule 3.2) indicates that some extra pieces of data must be considered in order to get the maximum performance. In Rule 3.2, a data token is a minimum piece of data returned by the lexical analyzer of a translator.

Rule 3.1 (Buffer size) *The user-selected output chunk size for one module must match the input chunk size of the downstream neighboring module.*

It is important to make the chunk size of the output stream from a module be equal to the chunk size of the input stream of the next module, so as to avoid busy waiting between communicating modules. This matter is made more complicated due to the fact that each module requires a certain number of lookahead tokens to make parsing decisions.

Figure 3.2 illustrates how the number of lookahead tokens propagates backwards and accumulates in a pipeline. The three modules in this figure are connected in the way illustrated in Figure 3.4, with module A as the first module in the pipeline and module C as the last. Each module chooses to process data in chunks of 3 tokens each. Suppose the parsers implemented in the modules are all $LL(1)$ parsers, since module C requires 1 lookahead token, module B must put 4 tokens in the first chunk of its output. Now module B must get 4 tokens from module A for its first input. Since module B also requires a lookahead token, module A must generate 5 tokens for the first chunk of its output. In the second cycle of *IPO* – and it is the same for the rest of the cycles – for module C, only 3 tokens are required from module B, because the token left over from cycle 1 (not processed and output), together with the first two tokens received from module B in the second cycle, will form the second data chunk, and the third token from module B will serve as the lookahead token for the parser in module C. The same thing happens in modules B and A, since each of them has exactly one token left over from the previous cycle.

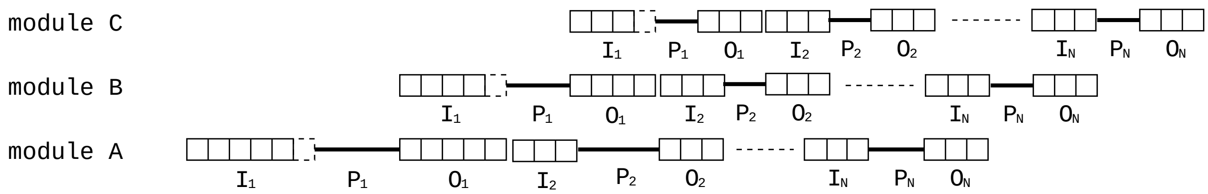


Figure 3.2: The back-propagation of lookahead tokens. The chunk size for each module is 3 tokens. The tokens represented by the dashed squares are the lookahead tokens required by the parser in the modules. The extra number of tokens put in the first output chunk from each module is cumulatively requested by downstream modules.

The following theorem formally states the sufficient conditions for lookahead number propagations and how to compute them.

Rule 3.2 (Lookahead) *The first output chunk from a module must contain some extra data tokens. The precise number of extra tokens is determined by Theorem 3.1.*

Theorem 3.1 (Back-propagation of the number of lookahead tokens) *Let $f(x)$ be the number of tokens that module i will generate for output given x tokens in the input stream. A module with such an f is called a* Lookahead-Conductive (LC) *module. A pipeline consisting entirely of LC modules is an LC pipeline. If a linear LC pipeline consists of modules $1, 2, \cdots, K$, with module 1 being the last module and module K being the first one, and if module i requires L_i lookahead tokens and f_i^{-1} exists, then the total number of extra tokens that must be generated by module i is*

$$O_i = \underbrace{f_{i-1}^{-1}(f_{i-2}^{-1}(\cdots(f_2^{-1}}_{i-2}(L_1) + L_2)\cdots) + L_{i-2}) + L_{i-1} \tag{3.1}$$

for $i = 2, 3, \cdots, K$.

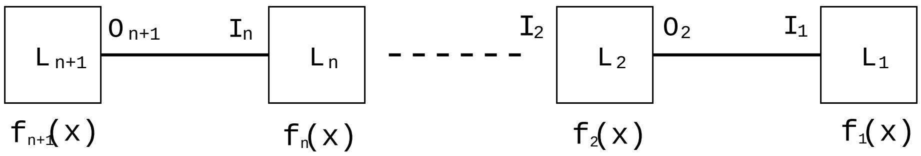


Figure 3.3: A linear LC pipeline.

Proof

Figure 3.3 shows a linear LC pipeline. Let

L_i be the number of lookahead tokens required by module i;

I_i be the number of extra tokens needed on the input port of module i;

O_i be the number of extra tokens generated by module i;

P_i be the number of tokens needed to generate O_i.

We find the following relations to be true:

$$O_i = I_{i-1}, \qquad 2 \le i \le K \tag{3.2}$$

$$P_i = f_i^{-1}(O_i), \qquad 1 \le i \le K \tag{3.3}$$

$$I_i = L_i + P_i, \qquad 1 \le i \le K \tag{3.4}$$

$$\begin{aligned} O_1 &= P_1 \\ &= 0 \end{aligned} \tag{3.5}$$

Equation 3.2 is based on the condition that module i generates the exact number of extra tokens needed by module $i-1$. Equation 3.3 is true by the definition of f. Equation 3.4 is true by definition, too, because I_i is the sum of two parts: the number of lookahead tokens (L_i) in the input which will be consumed by the module (the parser in the module, for example), and the number of extra tokens (P_i) that will be translated into output tokens. Equation 3.5 is true because module 1 does not have any downstream modules and so it does not need to generate extra tokens.

When $i = 2$,

$$\begin{aligned} O_2 &= I_1 && \text{(from (3.2))} \\ &= L_1 + P_1 && \text{(from (3.4))} \\ &= L_1 && \text{(from (3.5))} \end{aligned} \tag{3.6}$$

and for $i > 2$,

$$\begin{aligned} O_i &= I_{i-1} && \text{(from (3.2))} \\ &= P_{i-1} + L_{i-1} && \text{(from (3.4))} \\ &= f_{i-1}^{-1}(O_{i-1}) + L_{i-1} && \text{(from (3.3))} \\ &= \cdots && \text{(substitute } O_{i-1}, \cdots, O_3) \\ &= \underbrace{f_{i-1}^{-1}(f_{i-2}^{-1}(\cdots(f_2^{-1}}_{i-2}(O_2) + L_2)\cdots) + L_{i-2}) + L_{i-1} \\ &= \underbrace{f_{i-1}^{-1}(f_{i-2}^{-1}(\cdots(f_2^{-1}}_{i-2}(L_1) + L_2)\cdots) + L_{i-2}) + L_{i-1} && \text{(from (3.6))} \end{aligned}$$

Thus (3.1) is true for $i \geq 2$. ∎

Note that in general the number of lookahead tokens $L_i = PL_i + AL_i, 1 \leq i \leq K$, where PL_i is the number of lookahead tokens required by the parser and AL_i is the number of extra tokens needed by the algorithm implemented in module i. In other words, both the parser and the visualization algorithm contribute to the number of lookahead tokens. In Chapter 4, for example, we will show that the Marching Cubes algorithm needs two "lookahead" (2D) slices to generate the output polygons for the "current" slice it's working on.

If the number of output elements of a module does not functionally correlate to the the number of input elements, then the number of lookahead tokens from the downstream modules does not propagate through this module, and this module becomes a break point in the back-propagation of lookahead tokens. This is again exemplified by the Marching Cubes algorithm in which the number of triangles generated from the algorithm does not depend on the number of grid points in the data set. Of the 9 modules discussed in Section 5.4, only two modules, the data slicing module `slice` and the iso-surface module `onion`, don't have the LC property.

More discussion about the lookahead tokens will be given in the real visualization pipelines discussed in Chapter 5, and their effect on the performance of the pipelines will be discussed in the Chapter 6.

3.3 Analytical Performance Analysis

A general dataflow graph is a *Directed-Acyclic Graph* (DAG). But it can be transformed into a linear pipeline after a topological sorting. So in this section we only study performance of linear pipelines.

According to the definitions by Kogge [55], a *pipeline* is a form of computer organization in which successive steps of a process are executed in turn by a linear sequence of specialized modules capable of operating concurrently, so that another process can be begun before the previous one is finished. A *linear pipeline* is a pipeline in which each stage is connected to only a single succeeding stage.

The analysis below shows the ideal speedup that can be achieved by fine-grain pipelines. The speedup of a k-stage linear pipeline processor over an equivalent nonpipeline processor is well-known [42]. Here we are analyzing the speedup of one pipeline implementation scheme over another one.

An illustration of a linear visualization pipeline and its input data sets are given in Figure 3.4. Suppose that each module needs to process the same number of data elements and that each data element in a data set takes the same amount of time to process by a particular module. Let

K be the number of modules in a linear pipeline;

M be the number of frames of data to be fed into the pipeline;

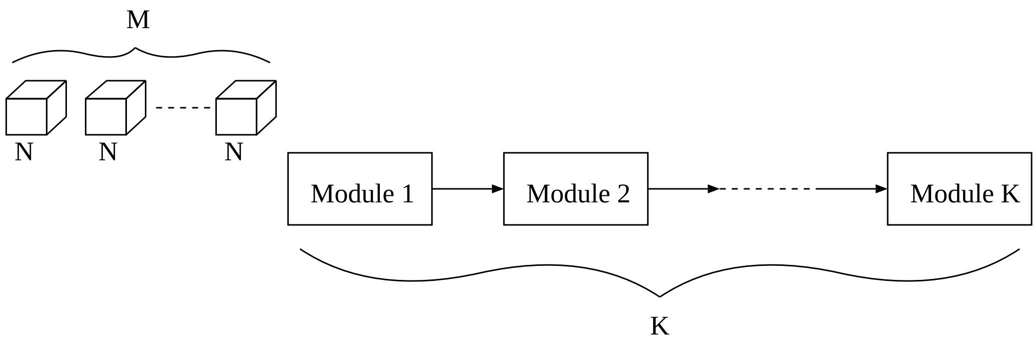


Figure 3.4: A linear visualization pipeline.

N be the number of data elements in one frame;

p_i be the time taken to process one data element by module i;

t be the longest stage delay of the pipeline using coarse-grain approach;

t' be the longest stage delay of the pipeline using stream approach;

$p_{max} = max\{p_i \mid 1 \le i \le K\}$;

$P = \sum_{i=1}^{K} p_i$.

Here time t is the time taken to process N data elements by the slowest module, and time t' is the time taken to process 1 data element by the slowest module. We have

$$\begin{aligned} t &= max\{p_i * N \mid 1 \le i \le K\} \\ &= N * max\{p_i \mid 1 \le i \le K\} \\ &= Np_{max}, \\ t' &= p_{max}. \end{aligned}$$

With the coarse-grain approach, where one frame is sent at a time, the time taken to complete processing M frames (ignoring the communication delay) is the time taken to generate the first output frame plus the time taken to generate the rest $M-1$ frames, i.e.,

$$\begin{aligned} T &= \sum_{i=1}^{K} Np_i + (M-1)t \\ &= NP + (M-1)Np_{max} \end{aligned}$$

With the fine-grain approach, where one element is sent at a time, the time taken to complete processing M frames is the time taken to generate the first output data element plus the time taken to generate the rest $MN-1$ elements, i.e.,

$$
\begin{aligned}
T' &= \sum_{i=1}^{K} p_i + (MN-1)t' \\
&= P + (MN-1)p_{max}
\end{aligned}
$$

The speedup is defined as the total execution time of a coarse-grain pipeline over the execution time of the corresponding fine-grain pipeline. So

$$
\begin{aligned}
speedup &= \frac{T}{T'} \\
&= \frac{NP + (M-1)Np_{max}}{P + (MN-1)p_{max}} \\
&= \frac{P/p_{max} + M - 1}{(P/p_{max} - 1)/N + M}
\end{aligned} \tag{3.7}
$$

Since $1 \leq P/p_{max} \leq K$ always holds, and $N \gg M$ and $N \gg K$ are generally true for scientific data sets, we can derive an approximation for the speedup from Equation 3.7:

$$
speedup \approx \frac{P/p_{max} + M - 1}{M} \tag{3.8}
$$

We can draw two conclusions from the above formula. When $M = 1$, we have

$$
speedup \approx \frac{P}{p_{max}} \tag{3.9}
$$

And when $M \gg K$, we have

$$
speedup \approx 1 \tag{3.10}
$$

We define the first case, where $M = 1$, to be the *experimental phase*. In the experimental phase, only one dataset is run through the pipeline. The maximum achievable speedup is K, the number of stages, when $p_1 = p_2 = \cdots = p_K$. The speedup could also approach 1 in the case of uneven module processing times, when $p_{max} \approx P$. We define the second phase, where $M \gg K$, to be the *production phase*, during which multiple datasets go through the pipeline to make an animation. In this phase, the speedup approaches 1.

The above analysis considers the case where communication overhead can be ignored. Shared-memory multiprocessors are examples of this type. When communications take up a significant portion of the execution time, we need a different model. The following analysis presents a performance model involving the communication times.

We now consider possible speedups in the experimental phase. Let

K be the number of modules in a linear pipeline;

N be the number of chunks the input dataset is divided into;

I_i, P_i, and O_i be the input, processing, and output time of the ith coarse-grain module;

I_i^j, P_i^j, and O_i^j be the input, processing, and output time of the jth *IPO* cycle of the ith fine-grain module;

T be the total execution time for the coarse-grain pipeline;

T' be the total execution time for the fine-grain pipeline;

$t_i = I_i + P_i + O_i$ be the total execution time of the ith coarse-grain module;

$t_{max} = max\{t_i \mid 1 \leq i \leq K\}$;

We assume that $I_i^j = I_i/N$, $P_i^j = P_i/N$, and $O_i^j = O_i/N$ for $1 \leq i \leq K$ and $1 \leq j \leq N$, i.e., in each fine-grain module, the input time is the same in every *IPO* cycle, and the same is true for the processing time and the output time.

In the coarse-grain pipeline each module's output overlaps with the next module's input. Thus $I_i = O_{i-1}$, for $2 \leq i \leq K$ (see Figure 3.5). The total execution time of the coarse-grain pipeline is the time for one large chunk of data to go through all the stages of the pipeline. In the fine-grain pipeline each module has N cycles of *IPO* and the following relationship holds: $I_i^j = O_{i-1}^j, 2 \leq i \leq K, 1 \leq j \leq N$. The total execution time of the fine-grain pipeline is the time to generate the first output chunk from the Kth module, plus the time for the next $N-1$ chunks to come out of the Kth module. The stage delay in this case is t_{max}/N. So we have

$$\begin{aligned} T &= I_1 + \sum_{i=1}^{K}(P_i + O_i) \\ T' &= \frac{I_1}{N} + \sum_{i=1}^{K}(\frac{P_i}{N} + \frac{O_i}{N}) + (N-1)\frac{t_{max}}{N} \end{aligned}$$

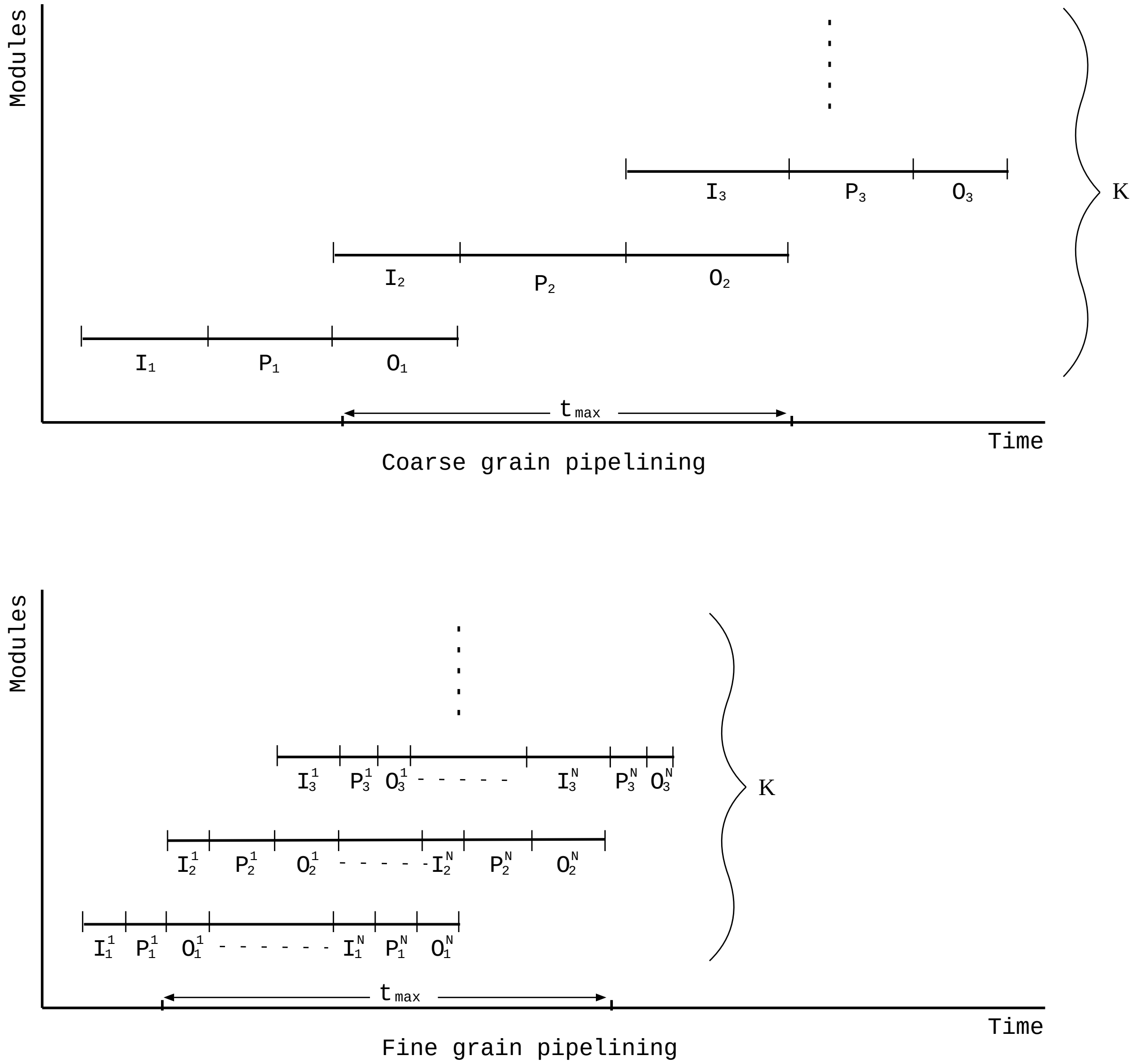


Figure 3.5: Time difference between a coarse-grain and a fine-grain pipeline. Suppose the second module is the bottleneck of the pipeline.

$$= \frac{1}{N}[I_1 + \sum_{i=1}^{K}(P_i + O_i) + (N-1)t_{max}]$$

and

$$speedup = \frac{T}{T'} \tag{3.11}$$

$$= \frac{I_1 + \sum_{i=1}^{K}(P_i + O_i)}{I_1 + \sum_{i=1}^{K}(P_i + O_i) + (N-1)t_{max}} N \tag{3.12}$$

To understand this formula, let us consider some special cases.

Case 1: Suppose $t_1 = t_2 = \cdots = t_K = t$, and $I_i = P_i = O_i = \frac{1}{3}t, 1 \leq i \leq K$, then Equation 3.12 can be simplied to

$$speedup = \frac{2K+1}{3N + 2K - 2} N \tag{3.13}$$

When $N \gg K$, the above equation can be further simplified to

$$speedup \approx \frac{1}{3}(2K+1) \tag{3.14}$$

Case 2: Suppose that the IO time is zero, i.e., $I_i = O_i = 0, 1 \leq i \leq K$, then Equation 3.12 can be simplified to

$$speedup = \frac{\sum_{i=1}^{K} P_i}{\sum_{i=1}^{K} P_i + (N-1)p_{max}} N \tag{3.15}$$

where $p_{max} = \{P_i \mid 1 \leq i \leq K\}$. This equation can also be derived from Equation 3.7 with $M = 1$.

Figure 3.6 compares the analytical speedup and empirical speedup measured from the execution of a contrived linear pipeline consisting of 6 modules. The input data set to this pipeline is a $360 \times 120 \times 120$ shock-wave data in binary format. Each module performs a simple normalization function except the first module which reads the data from a file. The data granularity of this pipeline is a parameter that can be varied. For the formula in Equation 3.12, the speedup is a function of the granularity N. We measured

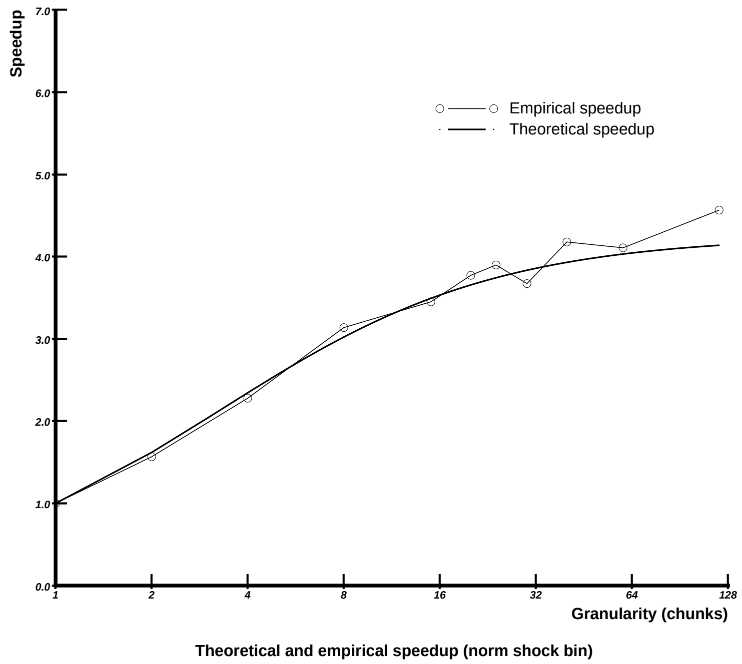


Figure 3.6: A comparison of analytical and empirical speedups. The pipeline consists of 6 modules. The input is a $360 \times 120 \times 120$ data set in binary format.

1. I_i, P_i, O_i of each module for the coarse-grain pipeline, from which the analytical speedup was derived, and

2. the total execution time for all the pipelines, from which the empirical speedup was derived.

This figure indicates that the formula given in Equation 3.12 is a rather accurate prediction for the performance of fine-grain pipelines based on the measured performance data of the coarse-grain pipeline. In this particular test case, the host machine has a large main memory (256 Mbytes), so there is not forced page swaps by the operating system. In Chapter 6 we'll present a case where the main memory size is much smaller than the total memory requirement of the pipeline and page swaps do affect the speedup. The speedup formula in Equation 3.12 does not consider factors such as virtual memory page faults and cache hit ratios, thus it will fail to predict the performance improvement in these cases.

For coarse-grain pipelines, the upper bound for CPU utilization is 2 in the experimental phase. The upper bound for CPU utilization in fine-grain pipelines is K (the number of stages).

Reduced memory usage is illustrated by two examples. Suppose we have a 3D scalar field of $n \times n \times n$ elements. With the coarse-grain dataflow approach, every module requires memory of size $O(n^3)$. Using the SDTM, we can show that the normalization module `norm` only needs memory of size $O(1)$, and the `isosurface` module that implements the Marching Cubes algorithm requires memory of size $O(n^2)$. Chapter 4 describes in detail 4 fine-grain visualization algorithms and their memory requirements.

Chapter 4

Fine-Grain Visualization Algorithms

This chapter describes fine-grain algorithms implemented using the SDTM method for four visualization tasks. They range from a simple algorithm, normalization, to a rather complex algorithm, volume rendering. It is assumed that these algorithms work on 3D gridded (uniform or non-uniform) scalar data sets whose size is $n \times n \times n$, although two of them, the normalization algorithm and the slicing algorithm, can be generalized to work on data sets of other dimensions. We will analyze the amount of memory required to store the data for each algorithm. The time complexity of these algorithms are the same as their corresponding coarse-grain algorithms, so we are not trading time with space here.

4.1 The Normalization Algorithm

The normalization algorithm takes a named scalar variable as input, normalizes the data values to between 0 and 1, then outputs the normalized variable, leaving the spatial reference, if any, unaffected. The minimum and maximum data values are stored together with the data values. The normalization function may be a linear, exponential, logarithmic or user-supplied function. The minimum and maximum values used by these functions can be taken from the min-max values stored with the data or can be supplied by the user at run-time. In the former case, assuming the data is stored in Flux format, the algorithm has to buffer the whole data set in memory because of the way the min-max values are stored. This can be illustrated by Figure 4.1, in which the min-max values are indicated by the **range** field containing two pointers into the data array.

```
begin variable "foo"
   notes[1] string ascii
       "This is a variable group"
   ^spatial_ref grid:"foo.grid"
   range 0 7
   data[2][2][2] float ascii
   0.1
   0.2
   0.3
   0.4
   0.5
   0.6
   0.7
   0.8
end variable "foo"
```

Figure 4.1: A 3D gridded scalar data set in *Flux* format

The fine-grain normalization algorithm requires a slightly different data language design. Instead of pointers the actual min-max values are stored in the field **range**. The sample 3D data set represented by the modified data language is shown in Figure 4.2. The only difference with the Flux format is the definition for the field **range**. The algorithm is described in Algorithm 4.1. Table 3.1 shows the SDTM implementation of the algorithm. The minimum memory requirement of the algorithm is $O(1)$. Figure 4.3 shows the attribute-annotated parse tree (if it is retained) generated by the SDTM module `norm` when parsing the input data set "foo" shown in Figure 4.2.

Algorithm 4.1 (Normalization)
begin

1. set the output buffer size;
2. read data header,
   send data header;
3. read minimum and maximum values,
   send the minimum and maximum values;

```
begin variable "foo"
   notes[1] string ascii
       "This is a variable group"
   ^spatial_ref grid:"foo.grid"
   range float 0.1 0.8
   data[2][2][2] float ascii
   0.1
   0.2
   0.3
   0.4
   0.5
   0.6
   0.7
   0.8
end variable "foo"
```

Figure 4.2: A 3D gridded scalar data set suitable for stream processing

4. **for** each input data element **do**
   begin
      normalize the value;
      save the normalized value in the output buffer;
      if output buffer is full **then**
         send all data in the output buffer to the downstream module;
   end

5. Send "end of data".

end.

In Figure 4.3, solid lines represent the parse tree and dotted lines with arrows represent attribute dependencies. The names in boxes represent attribute instances. If no dotted lines point to siblings to the left, then a one-pass attribute evaluation algorithm can be implemented.

Although the fine-grain normalization algorithm is a simple and obvious algorithm, there are three points that can be made and that are important in the SDTM-based systems. The first point is the *granularity control*, as illustrated by step 1 of the algorithm. Unlike the coarse-grain implementation of the normalization algorithm which allocates input and output buffers (in this algorithm the input and output buffers can be the same) large enough to hold the whole 3D data set (refer to Figure 5.1), the SDTM algorithm allocates an output buffer (it can be as

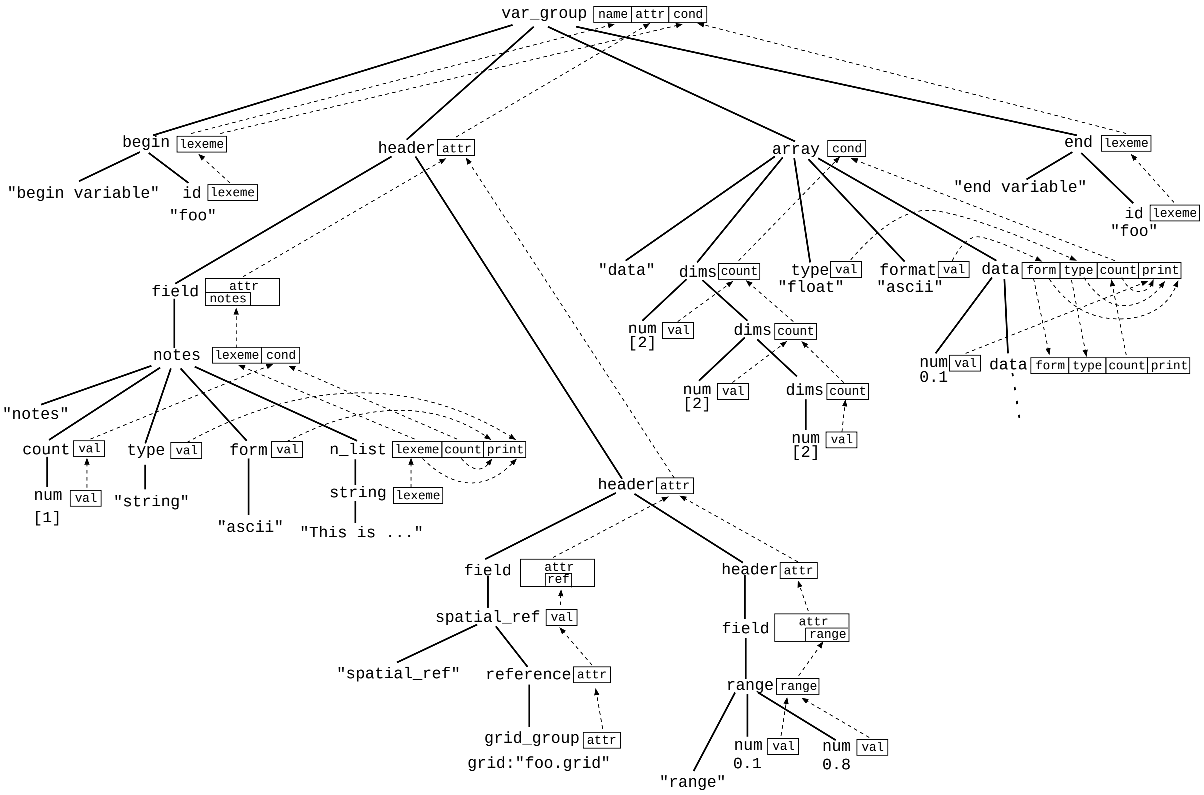


Figure 4.3: Attribute dependencies in the variable group "foo" induced by the module `norm`. The leaf nodes are the input data tokens to the normalization algorithm.

small as one data element) whose size is independent of the input data set. There is no explicit input data buffer since the input data is parsed by a lexical analyzer which has its internal buffer large enough to hold a token. The data granularity, here measured as the number of chunks that a whole data set is broken into, is a factor in determining the execution time of the whole visualization pipeline (see Equations 3.7 and 3.12).

The second point is that the communication between modules in a pipeline happens when the upstream module sends data down the pipeline. Each time a chunk of data is sent to the downstream module, a synchronization happens. Synchronizations add overhead to the execution of the pipeline, and too many synchronizations can actually decrease the performance dramatically (see Figure 6.6 for an example). Again the data granularity determines the synchronization frequency.

The third point is the relationship between a fine-grain algorithm and a one-pass attribute evaluation algorithm. They are actually two different views of the same solution. The L-AG theory provides an algorithm to verify that a module is indeed "fine-grain".

4.2 The Slicing Algorithm

The slicing algorithm takes a named 3D variable as input and outputs an orthogonal 2D slice from the 3D data. An X slice is a slice perpendicular to the X axis. Y and Z slices are similarly defined.

The coarse-grain slicing algorithm for X slices works as follows:

Algorithm 4.2 (Taking an X slice – coarse-grain)
Input: A one-dimensional stream of data representing a 3D array in row-major order.
    slice – the position of the slicing plane along the X axis in the index coordinate.
Output: An orthogonal 2D slice from the 3D array.
Variables:
    x, y, z – number of grid points on X, Y, and Z axis respectively
    $data[z][y][x]$ – an array storing the input data
    $outbuf[z][y]$ – an array storing the output data
begin
read data from input stream into an array data[z][y][x];
for j:=0 **to** y-1 **do**
    for k:=0 **to** z-1 **do**
        outbuf[k][j] = data[k][j][slice];
end.

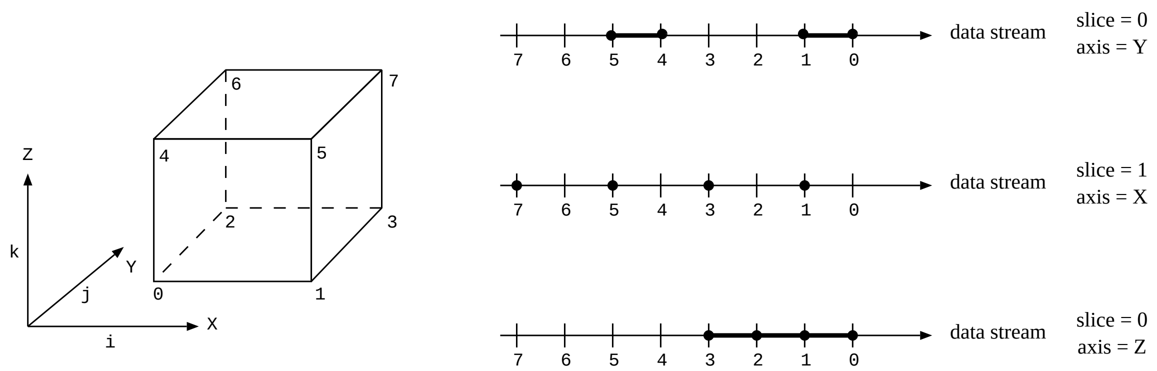


Figure 4.4: Examples of slicing a 3D gridded data set. Data values at highlighted grid points are selected for output. Data elements that are in one output chunk are connected by thick lines.

The fine-grain algorithm generates the output slice while it is receiving the input. The algorithm assumes that the input data is received in chunks, and that the size of each chunk is not known *a priori*. Thus the algorithm must handle the most general case. An example of taking a slice from a simple 3D data set can illustrate the idea of taking slices on the fly. In Figure 4.4, a $2 \times 2 \times 2$ 3D data set is shown on the left. On the right is how the data elements on the requested slices are selected. For an X slice, the selected elements are 2 (which is the number of grid points on X axis) points apart in the input data stream. For a Y slice, the selected elements are contained in several continuous segments, each containing 2 numbers, which is the number of grid points on X axis. For a Z slice, the selected elements are all in one continuous block of 4 (which is the number of grid points in an XY plane) elements. The complete algorithms for taking X, Y, and Z slices are given below.

Algorithm 4.3 (Taking an X slice)
Input: A one-dimensional stream of data representing a 3D array in row-major order.
    slice – the position of the slicing plane along the X axis in the index coordinate.
Output: An orthogonal 2D slice from the 3D array.
Variables:
    $last$ – the X index of the last element received
    $dist$ – difference of the X index between last element and the slicing plane
    x – number of grid points on X axis
    $data$ – a chunk of data from the input data stream
    $count$ – number of data elements in the chunk pointed to by 'data'
    $outbuf$ – buffer that stores output data
Initialization:
    $last = 0$;
    $x =$ number of grid points on X axis;

while not end of the data stream **do**
begin
    get a chunk of data from the input stream ($data$ and $count$ are set here);
    if $slice \geq 0$ **then**
        $dist = slice - last$;
    else
        $dist = slice + x - last$;
    if $count \geq dist$ **then**
    begin
        $i = dist$;
        $j = 0$;
        while $i < count$ **do**
        begin
            $outbuf[j] = data[i]$;

$i = i + x$;
$j = j + 1$;
end
output data in $outbuf$;
end
$last = (last + count)$ MOD x;
end.

In Algorithm 4.3, the granularity control is implemented through defining how large each input chunk is.

Algorithm 4.4 (Taking a Y slice)
Input: A one-dimensional stream of data representing a 3D array in row-major order.
slice – the position of the slicing plane along the Y axis in the index coordinate.
Output: An orthogonal 2D slice from the 3D array.
Variables:
$before$ – the number of elements expected before the next segment is seen
in – the number of elements in one segment to be selected since the last chunk is processed
$selected$ – the number of elements selected for output
x – number of grid points on X axis
y – number of grid points on Y axis
xz – number of grid points on an XY plane
$data$ – a chunk of data from the input data stream
$count$ – number of data elements in the chunk pointed to by 'data'
$outbuf$ – buffer that stores output data
Initialization:
$before = slice \times x$;
$in = x$;
x = number of grid points on X axis;
y = number of grid points on Y axis;
$xy = x \times y$;

while not end of the data stream **do**
begin
get a chunk of data from the input stream ($data$ and $count$ are set here);
if $count \leq before$ **then**
begin
$before = before - count$;
continue {go to the next iteration of the loop}
end
else
begin
$i = before$;
$j = 0$;

while $i < count$ **do**
begin
$selected = Min(count - i, in)$;
for $k := 0$ **to** $selcted - 1$ **do**
$outbuf[j + k] = data[i + k]$;
$i = i + xy - (x - in)$;
$j = j + selected$;
end
output data in $outbuf$;
end
end.

Algorithm 4.5 (Taking a Z slice)
Input: A one-dimensional stream of data representing a 3D array arranged in row-major order.
slice – the position of the slicing plane along the Z axis in the index coordinate.
Output: An orthogonal 2D slice from the 3D array.
Variables:
$before$ – the number of elements expected before the slicing plane is seen
in – the number of elements in the slicing plane that will be selected since the last chunk is processed
x – number of grid points on X axis
y – number of grid points on Y axis
xz – number of grid points on an XY plane
$data$ – a chunk of data from the input data stream
$count$ – number od data elements in the chunk pointed to by 'data'
$outbuf$ – buffer that stores output data
Initialization:
$x =$ number of grid points on X axis;
$y =$ number of grid points on Y axis;
$xy = x \times y$;
$before = slice \times xy$;
$in = xy$;

while not end of the data stream **do**
begin
get a chunk of data from the input stream ($data$ and $count$ are set here);
if $count \leq before$ **then**
begin
$before = before - count$;
continue {go to the next iteration of the loop}
end
else
begin
if $count \geq before + in$ **do**
begin

```
            for k := 0 to in − 1 do
                outbuf[k] = data[before + k];
            in = 0;
            before = 0;
        end
        else
        begin
            for k := 0 to count − before − 1 do
                outbuf[k] = data[before + k];
            in = in − (count − before);
            before = 0;
        end
        output data in outbuf;
    end
end.
```

Since the memory requirement by each of the three slicing algorithms is bounded by the chunk size, and the chunk size could be as small as one data element, the minimum memory requirement for the slicing algorithm is $O(1)$.

4.3 The Iso-Surface Algorithm

The fine-grain iso-surface algorithm is a restructuring of the Marching Cubes algorithm [61]. The algorithm "marches" through each cell in the grid, and generates polygons belonging to a particular iso-surface as it goes.

The implementation of the fine-grain algorithm is not particularly hard. In fact, the idea of working slice by slice has been mentioned by Lorensen [61]. Here we put it in the context of the SDTM model. For the iso-surface extraction module, each slice of the three-dimensional input data set is a meaningful chunk of data. Associated with each slice is a compute function that constructs surface polygons from that slice. When a slice of data is received, that compute function is called. If transparent iso-surfaces are generated, we must find a viewpoint-dependent order to generate the polygons from slices of data. But if no transparency is concerned, we can simply work slice by slice as the data comes in. Only four slices need to be buffered at all times (see Figure 4.5): slices 2 and 3 are used to generate vertices of the iso-surface polygons, and slices 1 and 4 are used to derive gradients which are normalized for surface shading calculations.

Slice 2 is considered the "current" slice, and slices 3 and 4 are "lookahead" slices needed in order to generate polygons. Here one slice is a token in the sense of Theorem 3.1.

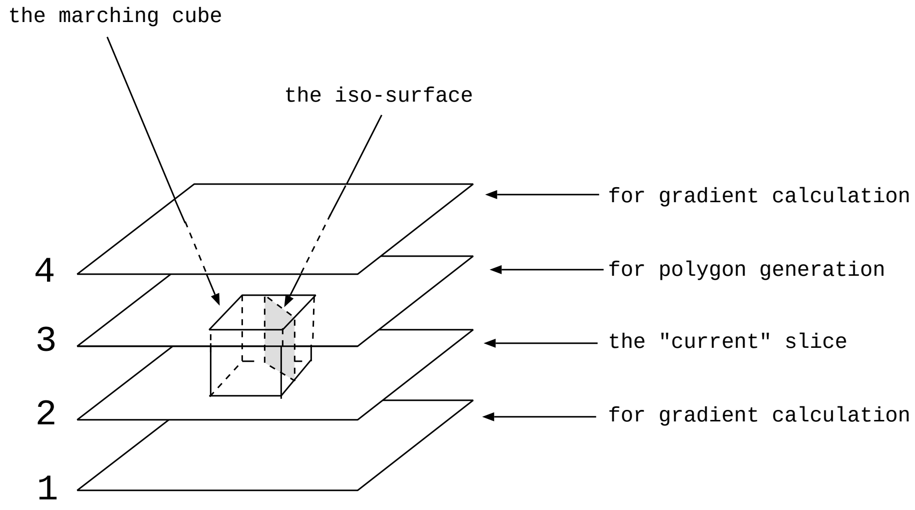


Figure 4.5: A fine-grain algorithm for iso-surface extraction.

The difference between a fine-grain algorithm and a coarse-grain algorithm in implementing the Marching Cubes algorithm is that the fine-grain algorithm allocates four 2D arrays as the input buffer to store four slices of data and the coarse-grain algorithm allocates one 3D array to store the whole data set. Unlike the normalization algorithm, this algorithm requires that the minimum chunk size be a 2D data slice, not a single data element. Thus the minimum memory requirement for this algorithm is $O(n^2)$.

4.4 The Volume Rendering Algorithm

The volume rendering algorithm implements a class of ray-casting algorithms exemplified by the one proposed by Levoy [59]. It is a two-pass algorithm. In the first pass, voxels are shaded and classified, which generates color and opacity for each voxel. In the second pass, a ray is cast from the eye point through each pixel in the image plane into the volume of data. Along each ray, colors and opacities are resampled at evenly spaced locations, and they are integrated in back-to-front order according to the following compositing formula at each sample point

$$C_{out} = (1 - \alpha)C_{in} + \alpha C \tag{4.1}$$

where C and α are the color and opacity of the current sample point, C_{in} and C_{out} are the result of the ray integration up to the previous and the current sample point respectively, in back-to-front order. Initially, $C_{in} = C_{bg}$ is the background color.

The first pass can be easily implemented using a fine-grain algorithm, which is similar to the one described for the Marching Cubes algorithm. The difficult part is to design a fine-grain algorithm for the second pass, which does ray-casting and ray-integration.

Suppose there are K sample points along the ray through a particular pixel and point 1 is closest to the eye point. The calculated color for that pixel can be derived as the following:

$$\begin{aligned}
C &= \alpha_1 C_1 + (1-\alpha_1) C_{in,1} \\
&= \alpha_1 C_1 + (1-\alpha_1)\alpha_2 C_2 + (1-\alpha_1)(1-\alpha_2) C_{in,2} \\
&\cdots \\
&= \alpha_1 C_1 + (1-\alpha_1)\alpha_2 C_2 + (1-\alpha_1)(1-\alpha_2)\alpha_3 C_3 \\
&\quad + \cdots + (1-\alpha_1)\cdots(1-\alpha_K) C_{bg}
\end{aligned} \tag{4.2}$$

Equation 4.2 can be evaluated from left to right. It can also be reformulated as the following and be evaluated from right to left:

$$\begin{aligned}
C &= \alpha_1 C_1 + (1-\alpha_1) C_{in,1} \\
&= \alpha_1 C_1 + (1-\alpha_1)[\alpha_2 C_2 + (1-\alpha_2) C_{in,2}] \\
&\cdots \\
&= \alpha_1 C_1 + (1-\alpha_1)\{\alpha_2 C_2 + (1-\alpha_2) \\
&\quad \underbrace{[\cdots[}_{K-2} \alpha_K C_K + (1-\alpha_K) C_{bg}] \cdots]\}
\end{aligned} \tag{4.3}$$

We call the formula in Equation 4.2 a *forward* integration formula and the one in Equation 4.3 a *backward* integration formula. It turns out that, although ray integration is a viewpoint-dependent computation, we can use one of these two equivalent formulas to implement the ray-casting stage of the volume rendering function in a streaming fashion, i.e. doing the ray casting as the data comes in.

The second pass (ray-casting and ray-integration) is done in two steps. In the first step, an image buffer is created and initialized. The image buffer contains color and opacity values for each pixel. It stores partial values of the integration until the last slice has been received, at which point the buffer contains the final volume-rendered image. In this step, the viewpoint is known and the geometry of the grid has been received, but the actual voxel data has not come yet. If the direction of a ray is the same as the direction along which the voxel slices progress (the voxel slices always progress from slice 1 to slice L, the total number of slices), the forward integration formula in Equation 4.2 will be used for ray integration. Otherwise, the backward integration formula in Equation 4.3 will be used. Figure 4.6 shows an example when backward integration is chosen because voxel slice 1 will be received first and the ray intersection with slice 1 is the last sample point in the integration equation (point K in Equation 4.3). For optimization reasons, a forward perspective projection of the bounding box of the 3D grid is performed to obtain the 2D bounding box of the image in the image plane. Pixels outside the 2D bounding box are filled with background color. For pixels inside the bounding box, either black color (having RGB values of zero) or the background color is used to initialize their values in the image buffer depending on whether forward or backward integration formula will be applied (see Figures 4.7 and 4.8).

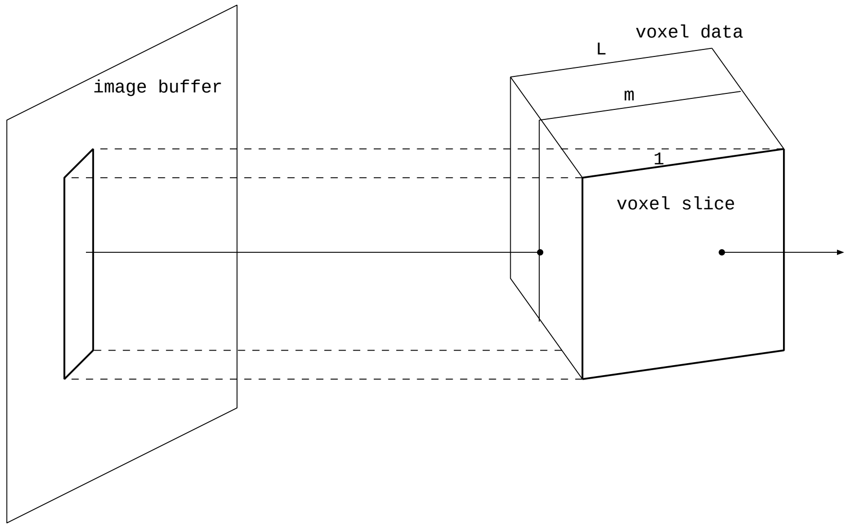


Figure 4.6: An example of backward integration for ray-casting. The ray in the graph intersects with slices from 1 to m.

In the second step, the color and opacity values in the image buffer are updated when successive slices are received. When a slice is received, a perspective projection is done to calculate the affected area in the image buffer. The color and opacity values associated with each pixel in the projected region are updated according to the formulas described in Figures 4.7 and 4.8, depending on the direction of integration. These formulas are derived from Equations 4.2 and 4.3. It is clear that backward integration requires one less storage cell for each pixel in the image buffer, but the selection of the integration direction is not decided arbitrarily, as has been mentioned before. However, the above analysis does point to a more efficient way to implement this class of algorithms in coarse-grain modules. In the inner-most loop (integrating sample points along the rays), the forward integration algorithm requires 3 multiplications and 2 additions, while the backward integration algorithm requires 2 multiplications and 2 additions.

Initialization: $C = 0$
$\alpha = 1$

for each sample point $i := 1$ to K **do**
begin
$C = C + C_i * \alpha * \alpha_i$
$\alpha = \alpha * (1 - \alpha_i)$
end

Figure 4.7: The forward integration algorithm. Variable C represents color and α represents opacity.

Initialization: $C = background$

for each sample point $i := K$ to 1 **do**
begin
$C = C * (1 - \alpha_i) + C_i * \alpha_i$
end

Figure 4.8: The backward integration algorithm. Variable C represents color.

In the first pass, a previous slice and a slice ahead are needed for shading calculation. Thus the minimum memory requirement for the first pass is $O(n^2)$. In the second pass, two slices have to be stored at any time for the points along the ray to be resampled. The minimum memory requirement for the second pass is $O(n^2 + wh)$, where w and h are the width and height of the image. If we assume fixed image resolution but unlimited data size, then the minimum memory requirement is $O(n^2)$.

4.5 Discussion

To write a module that works in streaming fashion, the module writer must carefully design the algorithm and the system designer must provide a data language that facilitates fine-grain algorithm design. For example, a two-pass algorithm such as Levoy's volume rendering algorithm can be implemented using two fine-grain modules, with each module implementing one pass. This eliminates the buffering of the output from the first pass. If the algorithm is implemented by a single module, which is the case in the apE module `opac`, the output from the first pass must be buffered. Breaking up a module into two or more modules is also a way of solving bottleneck problems, as it turns out that the volume rendering module is often the bottleneck in visualization pipelines.

As another example, the original Flux definition of the variable group does not allow a fine-grain implementation of the normalization algorithm. With a slight change in the field **range**, a fine-grain algorithm can be written easily. A similar example of using a triangle list group (similar to the PVD format used in the RIVERS software) instead of the Flux geometry group can lead to fine-grain rendering algorithms. Details of the triangle list group is given in Section 5.4.2.

Restructuring coarse-grain algorithms or designing new fine-grain algorithms is often a non-trivial task. It is relatively easy to implement the Marching Cubes algorithm in a fine-grain way. For Levoy's volume rendering algorithm it is not immediately clear that a fine-grain algorithm can be found, until after a ray integration formula is found to be evaluable in both directions. Writing a fine-grain module is usually harder than writing a coarse-grain module. The implementation of the volume rendering algorithm using SDTM contains 2971 lines of C code, while the original implementation of the coarse-grain algorithm in apE contains only 1784 lines of code. In the implementation of the fine-grain module `slice`, the increase is from 732 lines to 1292 lines.

Not all visualization functions can be implemented in a streaming fashion. Some functions for which fine-grain algorithms may be hard or impossible to find are streamline integration and ray-tracing. But there are a large number of functions that have corresponding fine-grain algorithms to implement them. So far, we have identified fine-grain algorithms that can implement functions like normalization, color-mapping, texture mapping, iso-surface extraction, contouring, hidden-line hidden-surface removal, and volume rendering. Here we assume that the data is generated on a rectilinear grid.

Granularity in a fine-grain module is an important leverage to improve performance of pipelines on multiprocessor and distributed environments. Each algorithm discussed in this chapter is able to control the granularity of input and output buffer sizes.

The memory requirements for the fine-grain normalization algorithm and the fine-grain slicing algorithm are $O(1)$. The memory requirements for the fine-grain iso-surface algorithm and the fine-grain volume rendering algorithm are $O(n^2)$. The memory requirements for the corresponding coarse-grain algorithms are all $O(n^3)$.

Because of the complexity and difficulties in building fine-grain modules, we have developed an automatic module generation system to help module writers specify granularity of data and computations. This is the topic to be discussed in the next chapter.

Chapter 5

The Implementation of an SDTM Visualization System

This section discusses the implementation of a visualization system based on the SDTM approach. The goals of this system are:

- To demonstrate that using fine-grain dataflow approach improves the memory efficiency, processing speed and CPU utilization of visualization applications.
- To study how well visualization algorithms can adapt themselves to this stream processing paradigm.
- To explore the use of a formal method to specify and generate dataflow visualization systems automatically.

5.1 System Overview

The SDTM system includes the following parts:

- A toolkit of predefined *modules* that can either be run individually or be connected into pipelines to form *tools*.
- An automatic module generator that takes grammar rules and semantic functions, and automatically generates the translators (the modules). Different storage-saving techniques that are suitable for our application will be applied to the generated translators.

- Three tools that are typically used in many visualization systems: The `Data Slicer` takes a 2D slice from a 3D volume data and maps it into a color image, the `Iso Surface` tool extracts iso-surfaces from a 3D volume data set, and the `Volume Rendering` tool renders the entire volume of data instead of only one surface or one slice.

In the coarse-grain approach, there is only one user-supplied *compute function* that is called once on each frame of data. With SDTM, that function is broken down into smaller semantic functions that are called during the parsing. The difference between the new software architecture and the original apE architecture can be found in Figures 5.1 and 5.2.

The syntax rules for each group of data will not change across modules. The differences in functionalities of the modules are reflected in semantic functions. An example of the attribute definition and translation scheme for the module `norm` has been shown in Section 3.2. Recursive descent parsers are built for each module.

5.2 The Translator Generation Tool

The SDTM module generator is based on the Purdue Compiler-Construction Tool Set (PCCTS) [77, 75]. Similar to Unix compiler generation utilities YACC and LEX, PCCTS contains a parser generator, `antlr` (ANother Tool for Language Recognition), and a lexical analyzer generator, `dlg` (DFA-based Lexical analyzer Generator). The `antlr` accepts as input a complete language description including rules for both lexical and syntactic analysis.

PCCTS constructs $LL(k)$ parsers with limited power for attribute specifications and analysis, and no provisions for binary data recognition and translation. To suit the needs of the SDTM, several enhancements have been made to the PCCTS so that module generation can be done completely in this environment. We call this new translator generation tool the *Module Generator*. Following is a list of major extensions to both the `antlr` and the `dlg`:

- An attribute analyzer to do attribute dependency analysis and to handle storage optimizations for attribute instances;
- Mechanisms for binary data recognition;
- Both terminal and nonterminal symbols can have multiple, user-defined attributes of arbitrary type;

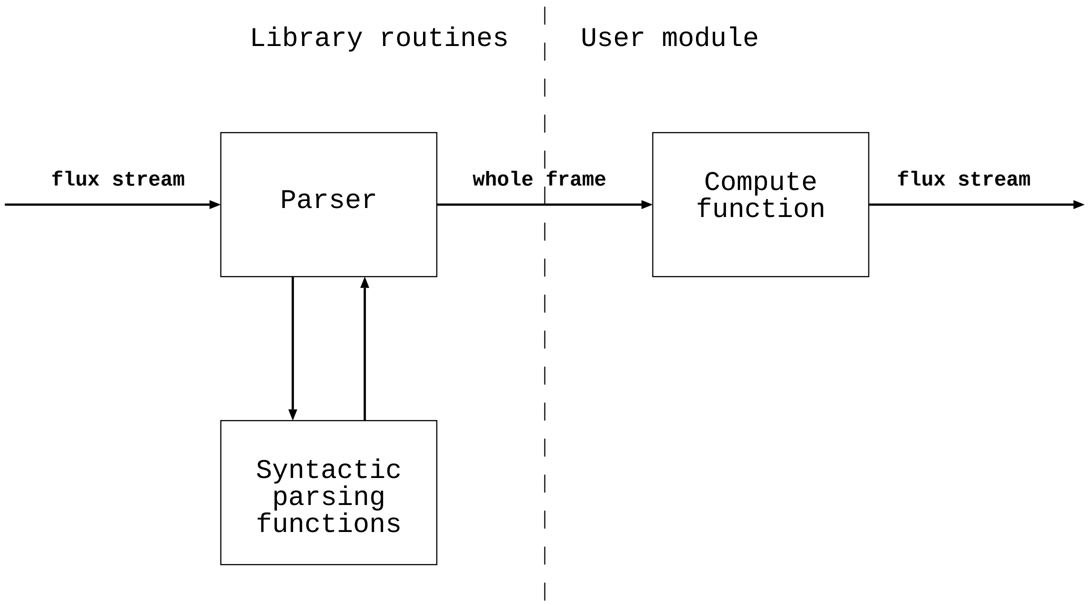


Figure 5.1: The software architecture of apE modules.

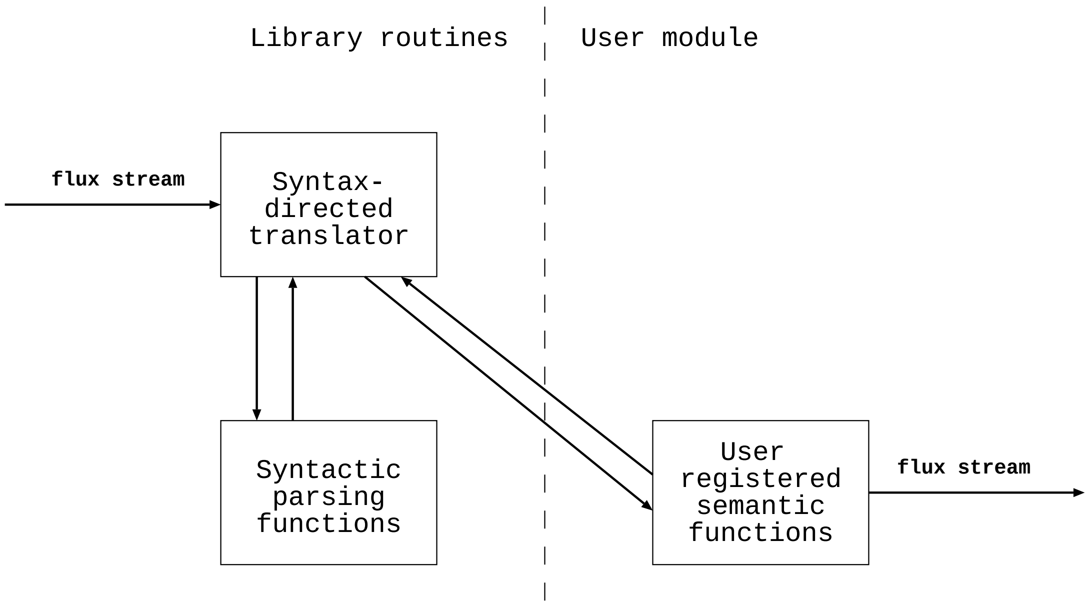


Figure 5.2: The software architecture of new modules.

- Attribute evaluation re-ordering based on the attribute analysis.

5.2.1 Parser Generator

The parser generator has the following capabilities:

- It generates $LL(k)$ parsers.
- It allows user-defined attributes (synthesized and inherited) to be attached to any non-terminal symbols and EBNF (Extended Backus-Naur Form) expressions.
- It allows user-defined semantic functions to be associated with each rule.
- It decides the order of attribute evaluation. Thus all semantic functions can be placed at the end of a rule.
- It contains an attribute analyzer.

As an example, the following translator translates the American format of a date (*mm/dd/yy*) to the European format (*dd/mm/yy*) [75]. Associated with each nonterminal symbol is an attribute which defaults to an integer in this example. `@` represents the end of file. `\t` and `\n` represent a tab and a newline respectively.

```
#header <<#include "int.h">>

<<
main()
{
    ANTLR(date(), stdin);
}
>>

#token "[\t\ \n]"     << zzskip(); >>      /* Ignore White */
#token Digits2 "[0-9][0-9]"                /* Define mm, dd or yy */

date:  Digits2 "/" Digits2 "/" Digits2 "@"
       <<printf("%02d/%02d/%02d\n", $3, $1, $5);>>
       ;
```

The output from the parser generator is an $LL(k)$ parser in the C language, and a high-level description of the lexical tokens, which is used by the lexical analyzer generator to generate the scanner. For example, the token specifications generated from the above grammar for date translation is the following:

```
[\t\ \n]
        <<
                LA(1) = 2;
                zzskip();
        >>
[0-9][0-9]
        <<
                LA(1) = Digits2;

        >>
```

To overcome the inefficiency of comparing k lookahead tokens for every parsing decision, the `antlr` actually generates an "optimized" $LL(k)$ parser called $LL_m(n)|m \leq n \leq k$ [76]. The theory of $LL_m(n)$ parser is based on the following observations.

- The majority of parsing decisions for a typical $LL(k)$ parser can be made without k lookahead tokens. At each decision point, the parser generator decides how many lookahead tokens, n, are needed. This is what n means in the name.
- The m means that in the cases when it is necessary to look n tokens into the future, it is rarely necessary to compare the next n tokens of input against all possible n-tuples for that production. For some alternatives, m-tuple comparisons may be adequate.

Note that m and n are varied at each point of parsing decision.

5.2.2 Lexical Analyzer Generator

The lexical analyzer generator now provides the following capabilities:

- It allows user-defined attributes (synthesized only) to be associated with each token, and user-defined functions computing values of the attributes to be called after a token is recognized.
- Parser-directed recognition of binary data as tokens. To handle binary data efficiently, a "chunk" mechanism was developed. Three types of binary chunks have been implemented:

1. float chunk: a float chunk consists of homogeneous floating point numbers.
2. voxel chunk: a voxel chunk is a 2D slice of opacity values plus RGB color values for each voxel.
3. tlist chunk: a tlist chunk consists a list of triangles in PVD (Per-Vertex Data) format.

The ASCII tokens are recognized using the DFA (Deterministic Finite-state Automaton) technique, which is both slow and space-consuming (it requires large tables).

5.3 The Attribute Analyzer

The attribute analyzer contains two parts: attribute dependency analysis and attribute storage optimization.

5.3.1 Attribute Dependency Analysis

The attribute dependency analyzer checks the conditions for L-AG (see Section 2.2) and provides the dependency information for the parser generator to generate the right order for attribute evaluations. It is presented in Algorithm 5.1.

Algorithm 5.1 (L-AG test)

begin

{ Assuming that the semantic functions are in Bochmann Normal Form }

    for each rule p of the grammar **do**

        for each nonterminal occurrence X_j on the right side of p **do**

            for each defining function f for inherited attributes of X_j **do**

                for each argument $Y.a$ of f **do**

                begin

                    if node_position(a) $= 0$ and a is a synthesized attribute **then**

                        return FAIL;

                    if node_position(Y) $\geq j$ **then**

                        return FAIL;

                end

    return PASS;

end.

5.3.2 Attribute Storage Reduction

The storage optimization issue in attribute evaluators can be characterized as the following:

1. Whether instances of an attribute should be implemented as:
   (a) components of parse tree nodes, or
   (b) elements of a stack, or
   (c) values of a global variable.
2. Which of the attributes that are implemented as stacks or variables can be combined together in order to:
   (a) eliminate copy rules, or
   (b) reduce the number of stacks and variables.

The *lifetime* of an attribute instance is defined as the time between its computation and its last application. The necessary condition for using stacks to store attribute instances states that if, in *any* semantic tree (parse tree augmented with attribute instances), the lifetimes of any two instances of $X.a$ are either *disjoint* or *properly nested*, then $X.a$ can be implemented as a global stack.

The top-down parsing and one-pass, left-to-right attribute evaluation means that the lifetimes of attributes of any parent node in a parse tree and those of its descendents are either properly nested or disjoint. So it is possible to use one stack for each attribute occurrence to store all its instances. For example, in the following rule:

$X \rightarrow Y X_1 Z$

$Y.b = X.a;$

$X_1.a = X.a;$

$Z.c = X.a;$

the lifetime of a of the X_1 on the RHS is properly nested in that of the X on the LHS. Here X and X_1 are the same symbol. The subscription is used only to distinguish the two occurrences.

For multi-pass AG's, the situation is more complicated and the lifetimes of the same attribute's instances are not guaranteed to be properly nested or disjoint.

In the case of PCCTS implementation, since attributes are allocated as local variables of the subroutine corresponding to the LHS of a rule, their lifetimes are all the same: the lifetime of that subroutine. Their lifetimes can be regarded as nested. And they all share a single stack – the run-time stack for the program – instead of one stack for each attribute. The attributes of terminal symbols are implemented using an attribute stack. Because of the L-AG property of the grammars, no explicit parse tree nodes need to be constructed, thus no attribute instances are stored with the parse tree.

If an attribute of a symbol is a large data array, using stacks to store its instances is still not efficient enough. The best solution is to use a global variable. However, the necessary condition for globalization of attribute instances is more strict: If, in *any* semantic tree, the lifetime of *any* instance of an attribute $X.a$ cannot overlap the lifetime of any other instance of $X.a$, then $X.a$ can be implemented as a global variable.

The globalization of attribute instances can lead to the elimination of copy rules. By eliminating copy rules, we not only save storage space for attribute instances but also save computing time. At this point, we only consider the elimination of copy rules that copy values between instances of the same attribute of the same symbol (such as attribute a in both X and X_1 in the above example). For example, in the rule: $data \rightarrow \mathbf{num}\ data_1$, attributes of $data$ can be implemented as global variables. Values copied between different attributes are not considered, since in our case we allow an attribute to be a structure.

We adapted the techniques used in GAG for globalization of certain attributes [23]. The original techniques are for multi-pass attribute grammars. We simplified them to work for the L-AG case. The sufficient conditions for globalization of inherited attributes and synthesized attributes are different.

Sufficient Conditions for Globalization of Inherited Attributes: Assuming production p has X as its LHS, an inherited attribute $X.a$ can be implemented as a global variable unless, between the attribute evaluation of $X.a$ in some other rule q and the last reference to it in rule p, either

1. a RHS occurrence of $X.a$ is defined, or

2. some RHS node Y is visited and $Y \stackrel{*}{\Longrightarrow} X$. ∎

The algorithm for globalization of inherited attributes is presented in Algorithm 5.2. $Y \stackrel{*}{\Longrightarrow} X$ (which reads: X is derivable from Y) if and only if (Y, X) is in the reflexive and transitive closure of the relation $\{(A, B) \mid \exists p : A \Longrightarrow \alpha B \beta\}$. As an example, the following AG definition for the inherited attribute $X.a$ violates the sufficient condition 1, because a is defined for X_1 on the RHS of rule p before the last use of $X.a$ in defining $Y.b$ in p.

$q : Z \rightarrow X$

  $X.a = Z.g;$

$p : X \rightarrow X_1 Y$

  $Y.b = X.a;$

  $X_1.a = X.a + 1;$

Algorithm 5.2 (Find globalizable inherited attributes)

begin
  Mark all inherited attributes as globalizable;
  for each production $p : X \rightarrow X_1 X_2 \cdots X_m$ **do**
  begin
    for each inherited attribute $X.a$ **do**
      for each X_i on the RHS **do**
        if defined($X_i.a$) > 0 and defined($X_i.a$) $<$ last_reference($X.a$) **then**
        begin
          mark $X.a$ not globalizable;
          break {go to the next iteration of the inner-most loop}
        end
    for each nonterminal symbol X_j on the RHS **do**
      if $X_j \stackrel{*}{\Longrightarrow} X$ **then**
        for each inherited $X.a$ such that last_reference($X.a$) $> j$ **do**
          mark $X.a$ not globalizable;
  end
end.

Sufficient Conditions for Globalization of Synthesized Attributes: For a synthesized attribute $X.a$, assuming production p has a RHS occurrence of X, say X_j, it can be implemented as a global variable unless, between the attribute evaluation of $X.a$ following the visit to X_j and the last reference to $X_j.a$ in rule p, some RHS node Y is visited and either $Y = X$ or $Y \stackrel{*}{\Longrightarrow} X$. ∎

The algorithm for globalization of synthesized attributes is presented in Algorithm 5.3. The example below shows that $X.a$ cannot be globalized because Y is visited during the lifetime of $X.a$ in p and that $Y \stackrel{*}{\Longrightarrow} X$.

p: $A \rightarrow XYZ$

$\quad\quad Z.g = X.a;$

$\quad X \rightarrow \alpha \mid \gamma$

$\quad Y \rightarrow X\beta$

Algorithm 5.3 (Find globalizable synthesized attributes)

begin

Mark all synthesized attributes globalizable;

for each production $p : X \rightarrow X_1 X_2 \cdots X_m$ **do**

  for each nonterminal occurrence $X_j = X$ **do**

    for each synthesized attribute a of X_j **do**

      for each nonterminal Y from j to last_used(a) $- 1$ **do**

        if $Y = X$ or $Y \stackrel{*}{\Longrightarrow} X$ **then**

          mark a not globalizable;

end.

A new form of attribute storage reduction called *attribute substitution* can be used in the case of EBNF expressions (e.g., expressions like $(A|B)+$). An EBNF expression has many differences with a normal CFG (Context-Free Grammar) rule (see Barrett [7, Chapter 4]). But in terms of definition and reference of attributes of EBNF expressions, they have the same format as normal CFG rules. An EBNF expression, considered as a nameless rule, can have its own attributes just as CFG rules do. The following rule defining nonterminal A illustrates the idea:

$A\,[a]{>}[b] \rightarrow (B\ C\ D \mid E\ F\ G){+}\,[s]{>}[v]$

$$s = a; \quad (1)$$

$$b = v; \quad (2)$$

In the above production, a is an inherited attribute and b is a synthesized attribute of the nonterminal A, s is an inherited and v is a synthesized attribute of the EBNF expression $(BCD|EFG)$. $(\cdots)+$ means that the EBNF expression in the parentheses can occur one or more times. In PCCTS, attributes must be declared in the rules because EBNF expressions are allowed in the rules and they don't have names to refer to.

When an EBNF expression is subsumed by a CFG rule, and two attributes, one from the nonterminal in the CFG rule and one from the EBNF expression, can share one global variable (the techniques for identifying globalizable attributes described in this section can be used here), then the attribute of the EBNF expression can be replaced by the attribute of the nonterminal in the CFG rule. In the above sample grammar rule, for example, both semantic functions (1) and (2) can be eliminated and attributes s and v can be substituted by a and b respectively. The reason for this is because the attributes of the CFG rule are visible inside an EBNF expression in the PCCTS implementation. This relationship doesn't exist among normal CFG rules. Thus attribute substitutions are only available in rules involving BNF expressions. The difference between attribute substitution and globalization is that no global variables are created in attribute substitution.

5.4 Fine-Grain Pipelines

This section describes three visualization pipelines composed of fine-grain modules. These modules implement the algorithms given in Chapter 4. Each module was generated using the module generator.

5.4.1 The Data Slicer

`Data Slicer` extracts an orthogonal 2D slice from a 3D volume data and draws a color contour map of that slice. It has the basic functionality provided by the NCSA DataSlice tool. The pipeline consists of the following modules:

`slice` extracts an $n-1$ dimensional variable from an n-dimensional variable;

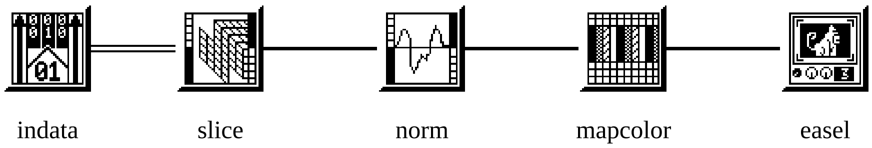


Figure 5.3: The `Data Slicer` pipeline.

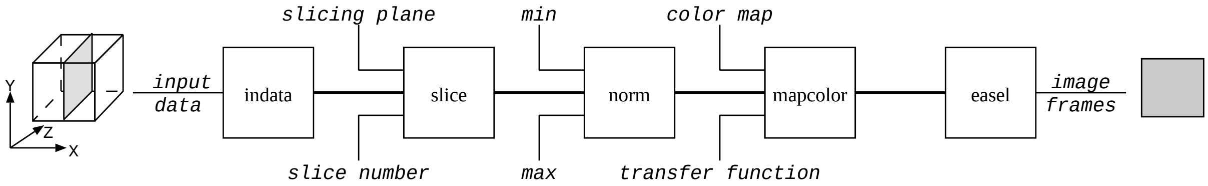


Figure 5.4: Configuration of the `Data Slicer`.

`norm` normalizes a variable using either a linear, logarithmic, exponential, or user-defined function;

`mapcolor` maps a variable to a raster image using a colormap and an optional transfer function;

`indata` loads flux data from files;

`easel` views/animates images.

The pipeline is shown in Figure 5.3. A screen dump of the `Data Slicer` in execution is shown in Figure 5.5.

The `Data Slicer` allows the user to interactively select, view and animate parts of a large data set generated by the simulation. The configuration of the `Data Slicer` is illustrated in Figure 5.4. The thin (dangling) lines (except the rightmost one) represent possible inputs from other modules or files. Thick lines represent connections between modules of the `Data Slicer`. Defaults are provided if no input connections are made. An animation can be made if a series of input frames are sent to a certain input port. Some parameters can be set interactively as well, such as the slicing plane and the slice number of the module `slice`.

All the modules are generated by the Module Generator using the parse-time attribute evaluation algorithm given in Section 2.2. The syntax rules used by modules `slice`, `norm`, and `mapcolor` are exactly the same. They only differ by the semantic functions associated with

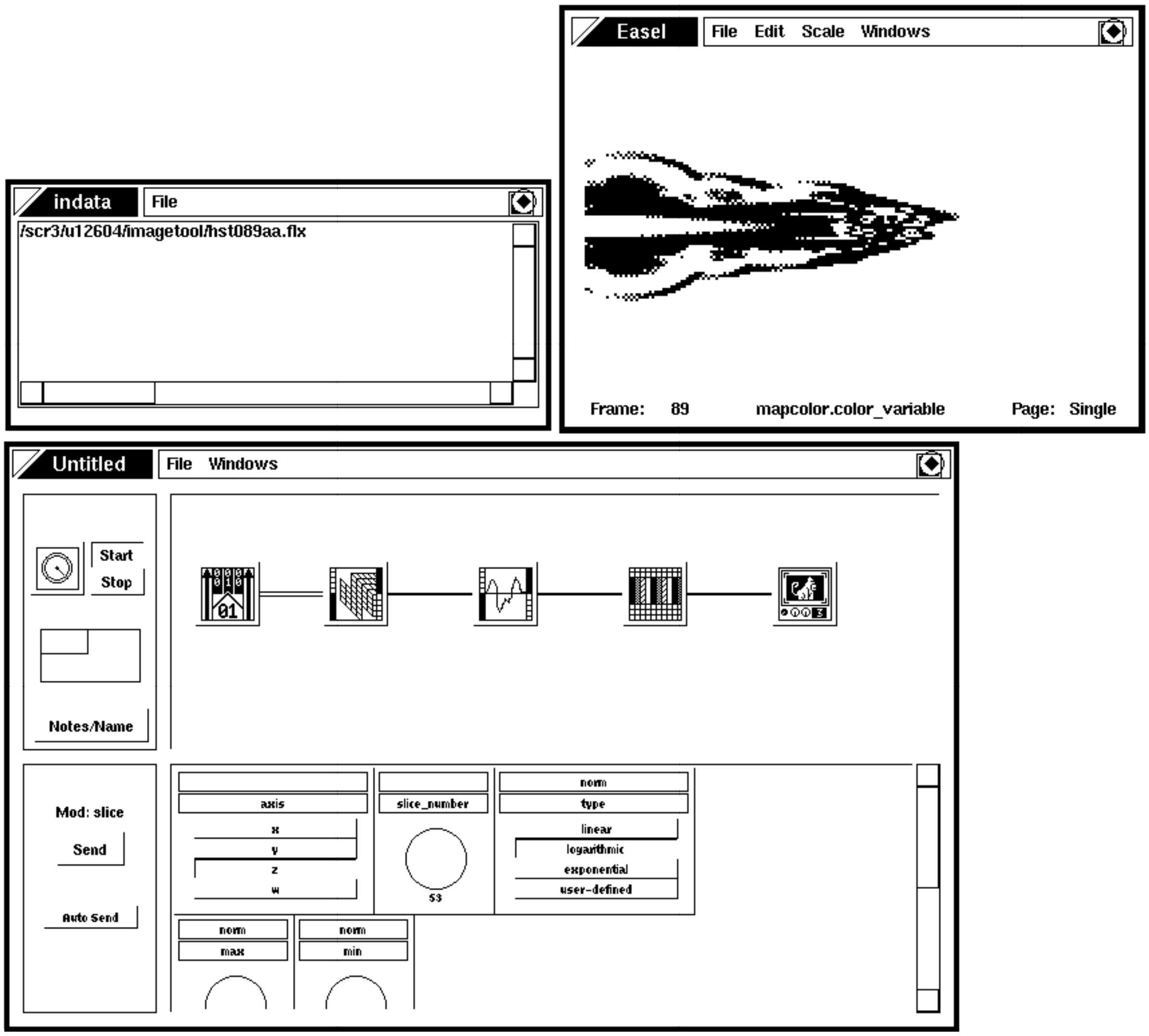


Figure 5.5: Screen dump from the `Data Slicer` visualizing a CFD simulation result.

each rule. As an example, the translation scheme is outlined in Section 3.2 for module `norm`. The source listing of the AG specification of `norm` written in PCCTS is given in Appendix B. Note that the input buffer size of one module must be the same as the output buffer size of the upstream module in order to obtain maximum speed. The granularity of each module can be any size from one data element to a whole data set. This pipeline is an LC pipeline.

5.4.2 The Iso-Surface Tool

The pipeline that extracts iso-surfaces (Figure 5.6) consists of the following modules:

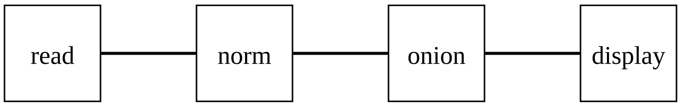


Figure 5.6: The iso-surface extraction pipeline.

`read` loads data from files;

`norm` normalizes 3D data using either a linear, logarithmic, exponential, or user-defined function;

`onion` extracts an iso-surface from a 3D scalar field;

`display` displays triangulated polygons.

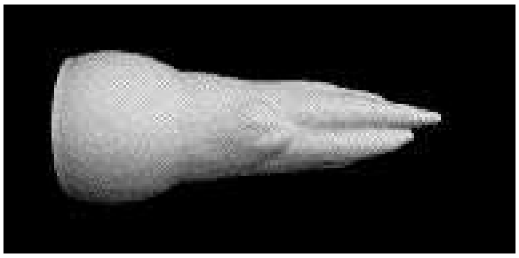


Figure 5.7: An iso-surface from a 3D jet simulation data set.

Each module is an autonomous Unix process communicating through pipes. The data granularity of each module is a parameter that can be controlled by a command line option. The difference between a fine-grain pipeline and a coarse-grain pipeline is the granularity of data communication and processing. For example, in the iso-surface pipeline, the minimum grain size is a 2D slice. The iso-surface extraction module `onion` takes one slice from the upstream module, constructs the surface polygons, and sends them to the downstream module. This three-step process repeats many times. Modules `read` and `norm` also work slice by slice in this case. Each module in the coarse-grain pipeline processes an entire 3D data set at one time. An output image dump from the pipeline is shown in Figure 5.7.

Because the `display` module (see Figure 5.6) processes a stream of triangles generated by the `onion` module, the total number of triangles is not known ahead of time. A new variable group called `tlist` has been created to represent a stream of triangles (see Figure 5.8). A `tlist` group contains an optional field `notes` that can be an arbitrary string of characters, and a data field containing the location, normal vector, and color components for each triangle vertex. The parser in `display` can recognize the end of the list based on the grammar rules specifying the `tlist` group.

```
begin tlist "shock wave"
   notes[1] string ascii
       "iso surface at value 0.25"
   data float ascii
   # Per-Vertex data containing: x, y, z, Nx, Ny, Nz, r, g, b
       0.5784 0.05263 0.05263 0.9907 0.1248 0.05334 0.90 0.90 0.30
       0.5789 0.05263 0.03767 0.9913 0.123 0.04361 0.90 0.90 0.30
       ....
end tlist "shock wave"
```

Figure 5.8: The triangle list group.

The z-buffer algorithm is implemented in `display` for hidden-surface removal. The z-buffer algorithm is readily applicable to fine-grain processing, and is implemented in hardware on most graphics workstations. The reason for implementing the z-buffer algorithm in hardware is probably the same for using fine-grain algorithms for visualization, i.e., the algorithm can do stream processing, and the algorithm itself does not put limit on the size of the data it can process.

5.4.3 The Volume Rendering Tool

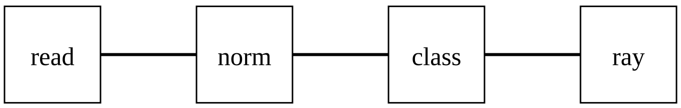


Figure 5.9: The volume rendering pipeline.

The pipeline that does volume rendering (Figure 5.9) consists of the following modules:

- `read` loads data from files;
- `norm` normalizes 3D data using either a linear, logarithmic, exponential, or user-defined function;
- `class` maps each voxel into an opacity value and an RGB color;

`ray` performs ray-casting and ray integration through a voxel data set, and generates an image.

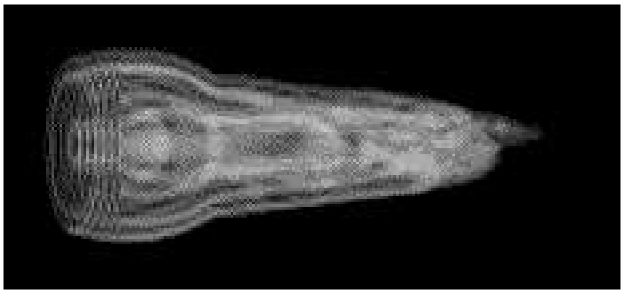


Figure 5.10: Volume rendering of a 3D jet simulation data set.

Again, the modules in the fine-grain pipeline process data slice by slice. Module `class` implements the first pass of the volume rendering algorithm, and module `ray` implements the second pass. The two passes are explained in details in Section 4.4. The modules in the coarse-grain pipeline buffer and process the entire data set only once. An output image from the module `ray` is shown in Figure 5.10.

Chapter 6

Performance Measurements

6.1 Methodology

The advantage of using fine-grain visualization pipelines is demonstrated by showing their performance improvement over coarse-grain pipelines. Performance measurements can also be used to identify problems with existing systems. There are, however, many measurements that can be taken at different levels. At the high level, measurements include throughput, response time, memory usage, and so on. At the low level, measurements can be taken as page swaps, cache hit ratios, CPU utilization, etc. Since the main purpose of designing fine-grain algorithms and using fine-grain pipelines is to reduce the memory usage, we will take a close look at the memory issues. As the result of using fine-grain modules, pipelines running on multiprocessor machines or distributed systems achieve higher CPU utilization and shorter execution time than their counterparts. Although speedup on multiprocessor machines can also be achieved by parallelizing each module, memory usage reduction can not be realized at the same time with this approach. The following measurements were taken for all the test pipelines:

1. *memory usage:* Two forms of memory usage were measured:
   - total *virtual memory size*, and
   - total *resident memory size*.

The virtual memory of a module includes code, data, and stack. The resident set of a module is the set of pages of the module[1] that are physically resident in memory. The total (virtual and resident) memory usage of a pipeline is the sum of the total amount of memory used by each module in the pipeline. The memory usage is a function of time, so measurements must be made continuously during the execution of the pipeline, and the time interval between any two measures must be short enough to catch all large memory size changes. The total virtual memory size shows how much memory is requested by the pipeline, and the total resident memory size shows how much memory is actually given to the pipeline by the OS for its use. These two measurements show how much memory is needed to run the pipeline efficiently without resorting to the swap device which could significantly slow down the execution. The *peak* (virtual and resident) *memory* usage can be derived from the measurements and it is another important quantity used in performance comparisons. From the peak memory usage, we can derive the *memory usage reduction (MUR)* which is defined as:

$$MUR_v = 1 - \frac{\text{peak virtual memory usage of the fine-grain pipeline}}{\text{peak virtual memory usage of the coarse-grain pipeline}}$$

and

$$MUR_r = 1 - \frac{\text{peak resident memory usage of the fine-grain pipeline}}{\text{peak resident memory usage of the coarse-grain pipeline}}$$

MUR gives a quantitative measurement of how much memory usage reduction is achieved.

2. *CPU utilization:* The CPU utilization is defined as the ratio of *user* plus *system times* to *real (elapsed) time*. The user time is the amount of time spent executing in user mode, and the system time is the amount of time spent in the system executing on behalf of the process. The real time is the wall-clock time. All of the times are added up across all the modules in a pipeline. On a machine with 4 processors, for example, CPU utilization of 400% means that all the processors are fully busy all the time. Software pipelining is a way of exploring parallelism, and the CPU utilization shows how effective this technique works.

[1] In the SDTM implementation, as well as in apE, a module is a process. If a module is not implemented as a process, other measuring methods must be found.

3. *execution time (completion time):* The execution times of the pipelines were recorded, and they were used to derive another quantity called speedup. The *speedup* is defined as the ratio of the execution time of the coarse-grain pipeline over that of the fine-grain pipeline. In most cases, the two pipelines are actually the same one with different granularity values. The speedup shows how well a fine-grain pipeline performs compared with a coarse-grain pipeline, and it can be measured by an end user. Speedup is related to CPU utilization, but they are not quite the same. Figures 6.6 and 6.7 show the difference.

Occasionally some other measurements were taken only to make specific points. These specific measurements include the timelines, process run queue length, page swaps, etc.

The three visualization pipelines described in Chapter 5 were used in the tests. For each pipeline, there are many parameters whose values can be varied to show their effect on the performance. There are pipeline-specific parameters such as the slicing plane for the `Data Slicer` tool and the iso-value for the `Iso-Surface` tool, and there are parameters common to all the pipelines. Pipeline-specific parameters will be explained in the following sections. Common parameters are explained below. The following parameters are orthogonal to each other and will be varied independently during the performance measurements.

1. *data set:* Performance data was collected for two simulation data files with different sizes. The first data file contains the density values of a shock wave. It is a volume data of $360 \times 120 \times 120$, about 5 million grid points. The second data file contains the density values of a jet simulation, and has dimensions $160 \times 80 \times 80$, about 1 million grid points. Large data sets normally require longer processing time, but some patterns of performance improvement using fine-grain pipelines are independent of the input data set. If the maximum speedup can be maintained while the input data size increases, then the pipeline is said to have the *data-scalability* property.

2. *granularity:* Granularity is the number of chunks that one data set is divided into. A chunk is a segment of data that a module receives, processes, and outputs atomically. So an SDTM pipeline with granularity of 1 corresponds to a coarse-grain pipeline. Granularity is the single most important factor in determining the performance of a fine-grain pipeline. It will be shown later that increasing the granularity increases the CPU utilization up to a limit. Too fine a granularity can increase the execution time and decrease

the performance because of the overhead incurred during the execution. The control of granularity is sometimes limited by the algorithms implemented in the modules, and finding the optimum granularity is done through trial and error. The granularity is a new dimension in the parameter space which does not exist in other dataflow systems. A typical performance curve related to granularity would be drawn using granularity as the abscissa and a measured or derived quantity such as CPU utilization, memory usage, or speedup as the ordinate.

3. *data format:* Data values can be in either ASCII format or binary format. Pipelines running ASCII data streams can spend as much as 90% of their time doing data conversion, whereas pipelines running binary data streams can spend more than 90% of the time doing data processing. The *IPO* patterns of the modules can be drastically different. However, we shall see that the performance of fine-grain pipelines improves in both situations.

4. *hardware configuration:* The pipelines are tested in three different configurations: a single-processor system (an SGI Personal Iris), a shared-memory multiprocessor system (an SGI VGX 4D/440), and a distributed multi-machine system (4 Personal Iris's connected to an Ethernet). The SGI VGX has 256 MBytes of main memory, and each Personal Iris has 16 MBytes of main memory. Unless otherwise noted, the tests were done on the SGI 4D/440.

5. *interprocess communications:* Interprocess communications can take the form of shared memory, pipes, files, or sockets. Only standard I/O was used for inter-modular communications in this implementation. All the pipelines are run as Unix pipelines using C-Shell scripts.

Memory usage is a function of time and it is sampled every 10 milliseconds. Measurements were made for each module and then added up. On Irix 4.0 (SGI's version of Unix operating system) there is a system call syssgi() that can return the total (virtual) process size and the resident set size of a process at the time the function is called. Each module being measured would spawn a subprocess that is only responsible for collecting memory statistics of its parent process every 10 milliseconds (which is the highest time resolution a user process can get). Memory size returned by syssgi() is accurate to 1 page (4 Kbytes in Irix 4.0).

We argue that the sampling rate is high enough so that no significant memory size changes will be missed based on the observation that:

- For scientific visualization software, a largest portion of the memory is used for storing data, and this piece of the memory is not likely to be allocated and released immediately without any data processing. For our test data and pipelines, large peaks of memory usage last at least for a few seconds.
- The run-time stack growth does not show up most of the time because the page size in Irix 4.0 is 4K and the memory size returned by the system call syssgi() is accurate to 1 page.

The two points will be supported by two memory usage plots drawn in Figures 6.1 and 6.2 in the next section (Section 6.2.1).

Two commands can be used to measure the CPU utilization and the execution time of a pipeline: C-Shell's built-in command `time` and the IRIX command `timex`. The `time` command returns the elapsed time, user time, system time, and CPU utilization of any UNIX command following it, which in our case is a C-Shell script. The `timex` command returns information on elapsed time, user time, system time, and a lot of other system activities such as paging information, cash hits/misses, disk I/O, etc. The C-Shell command `time` was used to measure the execution time and CPU utilization, and the IRIX command `timex` was used to measure page swaps and run queue length in a few cases. Time measurements were done separately from memory measurements to avoid time distortions caused by memory measurements, which can range anywhere from 2% to 22% of the total execution time.

Available through the software instrumentation is the events that denote *IPO* cycles of each module. A 'begin' and an 'end' event is generated for each phase of the *IPO* cycles. Those events can be used to draw timelines which can reveal how much time each module spends on computing and communications, and how well modules synchronize. Timelines can help improve pipeline performance by identifying synchronization problems. An example is given in Figure 6.16. Performance distortion caused by the software instrumentation is less than 1% in our system.

All the tests were conducted in dedicated mode (when no other users were logged in). Each number plotted in the following graphs is the average of four runs. In most cases, the standard deviation of time measurements is between 1% to 2% of the average value, and the memory measurements such as peak usage normally have no variance.

Performance measurements for three visualization pipelines for real applications are reported in the next section. Memory usage and the effect of granularity on performance is the constant theme throughout all the tests. Issues like the synchronization problem caused by lookahead tokens (addressed by Theorem 3.1) and the performance of pipelines running extremely large data sets are discussed as experiments with specific pipelines, since we believe that similar conclusions can be drawn for other pipelines.

6.2 Results and Discussion

6.2.1 The Data Slicer

With the Data Slicer tool, we studied the following issues:

- How the SDTM pipeline is compared with the original apE pipeline in terms of memory usage during the execution. This is done by measuring and presenting memory measurements and calculating the *MUR*.

- How the granularity affects their performance comparisons. Memory usage, CPU utilization, and speedup are calculated and plotted as functions of granularity.

- Data-scalability of the Data Slicer tool. Multiple datasets were used in the tests and their performance curves are drawn together to show similar performance curves.

- The *IPO* behavior of the pipeline taking slices of different orientations. This is to show how pipeline-specific parameters affect the performance.

- How the number of available processors affect the performance. This is to show how well the performance of both fine-grain and coarse-grain pipeline scales up with the number of processors. Both have limitations, but the limitations are different.

There is an apE pipeline that implements exactly the same function as the SDTM pipeline and consists of the same modules. The major difference is that the SDTM module `slice` implements a different slicing algorithm (described in Section 4.2), and that all the SDTM modules have the ability to vary the granularity. The entire Data Slicer pipeline is illustrated in Figure 5.3. We chose to take part of the pipeline from `indata` to `mapcolor` for performance measurements. It is done by running a C-shell script containing the pipeline: `cat <data file> | slice | norm | mapcolor`.

Data values in the two test data sets are stored as double-precision floating points in binary format. In each graph, we compare the performance of both the original apE pipeline (a coarse-grain approach) and our new implementation using SDTM (a fine-grain approach). Here the speedup is defined as the execution time of the apE pipeline over the execution time of the SDTM pipeline. This definition is slightly different from the definition used in the next two subsections in which the speedup is defined as the execution time of the SDTM pipeline with granularity of 1 over the execution time of the same SDTM pipeline with various granularities. We also varied the slicing plane and the slice number in the tests.

The virtual and resident memory usage for the apE pipeline and the SDTM pipeline is plotted in Figures 6.1 and 6.2 separately. Each figure shows two runs randomly selected from a total of five runs of the same pipeline. The two figures show that except for some small, random shift in time, the memory usage from different runs has the same characteristics. More importantly, the peak memory usage is exactly the same for both runs.

Figure 6.3 draws the memory usage of the apE and SDTM pipelines in one graph for comparison. The memory usage reduction $MUR_v = 96\%$, and $MUR_r = 98\%$. The coarse-grain apE pipeline has a wide peak that represents the memory usage of the `slice` module. When `slice` finishes at about 22 seconds, the 2D output slice goes through `norm` and `mapcolor`, and the memory requirement is much less than for 3D data. By contrast, the fine-grain modules all have about the same lifetimes, so the peak memory usage of the fine-grain pipeline stays the same during the lifetime of the pipeline. When a large chunk of memory is allocated for data processing, it takes some time to load in all the requested pages into the physical memory due to the demand paging policy of the operating system. This explains the linear growth of the resident memory curves. Figure 6.4 is the same as Figure 6.3 except that it uses log scale for the time axis and it effectively zooms in on the first few seconds of the plots in Figure 6.3.

Figure 6.5 shows the memory usage versus granularity, with the time axis drawn on log scale. There is a continuous memory usage drop with the increase of the granularity. This can be used as a guideline to decide the "best" granularity for specific machines with known input data set and physical memory size. The memory usage drops quickly as the granularity increases starting from 1, and then smooths out when granularity becomes fine enough.

Figure 6.6 shows how different granularities affect the speedup. The "granularity" axis is drawn using logarithmic scale. The speedup of the SDTM approach when granularity is one chunk, which is a coarse-grain pipeline in itself, is a little higher than 1.0 because the SDTM implementation of the `slice` module uses a different algorithm for slicing the data. The curves start to drop when the overhead involved with fine-grain processing outweighs the gain. Two different datasets, both in binary format, were used in the test.

Figure 6.7 shows the CPU utilization versus granularity. The CPU utilization approaches asymptotic upper limits when the granularity increases. This graph also contains performance data for the coarse-grain approach. These are the two points corresponding to granularity of 1, and are very close to each other and to the CPU utilization values of the SDTM pipelines at granularity of 1.

Figure 6.8 shows speedups achieved by taking different slicing planes on the shock-wave data set. The speedups for slices of different orientations are different because the `slice` module generates different patterns of *IPO* cycles (in terms of input, processing, and output ratios in each cycle)[2], but slices of the same orientation exhibit roughly the same speedup. The slicing algorithm implemented in the `slice` module in the fine-grain pipeline does not buffer the output it generates. In other words, when the algorithm finds, from the input chunk, elements that belong to the selected slice, it outputs them immediately. Among the three different slices, the Y slice generates the most balanced *IPO* cycles (see Figure 4.4 for an example).

Figure 6.9 compares CPU utilization of the two approaches with different slicing planes, using the shock-wave data set. It can be seen that the fine-grain approach achieves higher CPU utilization (or more concurrency) than the coarse-grain approach (ref. Figure 3.5), which is the main reason for the shorter execution time for the SDTM pipeline of fine granularities. The granularities in both Figure 6.8 and Figure 6.9 are 159 chunks for `slice` and 85 chunks for `norm` and `mapcolor`.

[2] Note that in the case of the Data Slicer, the numerator, the execution time of the apE pipeline, does not change because of the way the apE module `slice` is implemented. So the difference in speedup is only due to the difference in the execution time of the SDTM pipeline.

Figure 6.10 shows the speedup versus the number of processors. Here the speedup for n processors is defined as the execution time of a pipeline running on 1 processor over the execution time of the same pipeline running on n processors. Both curves show the same speedup pattern: speedup increases linearly with the number of processors for the fine-grain pipeline but is flat for the coarse-grain pipeline. This is because the upper bound for the CPU utilization of a coarse-grain pipeline is 2, and that of a fine-grain pipeline is K, the number of stages in the pipeline (ref. Section 3.3).

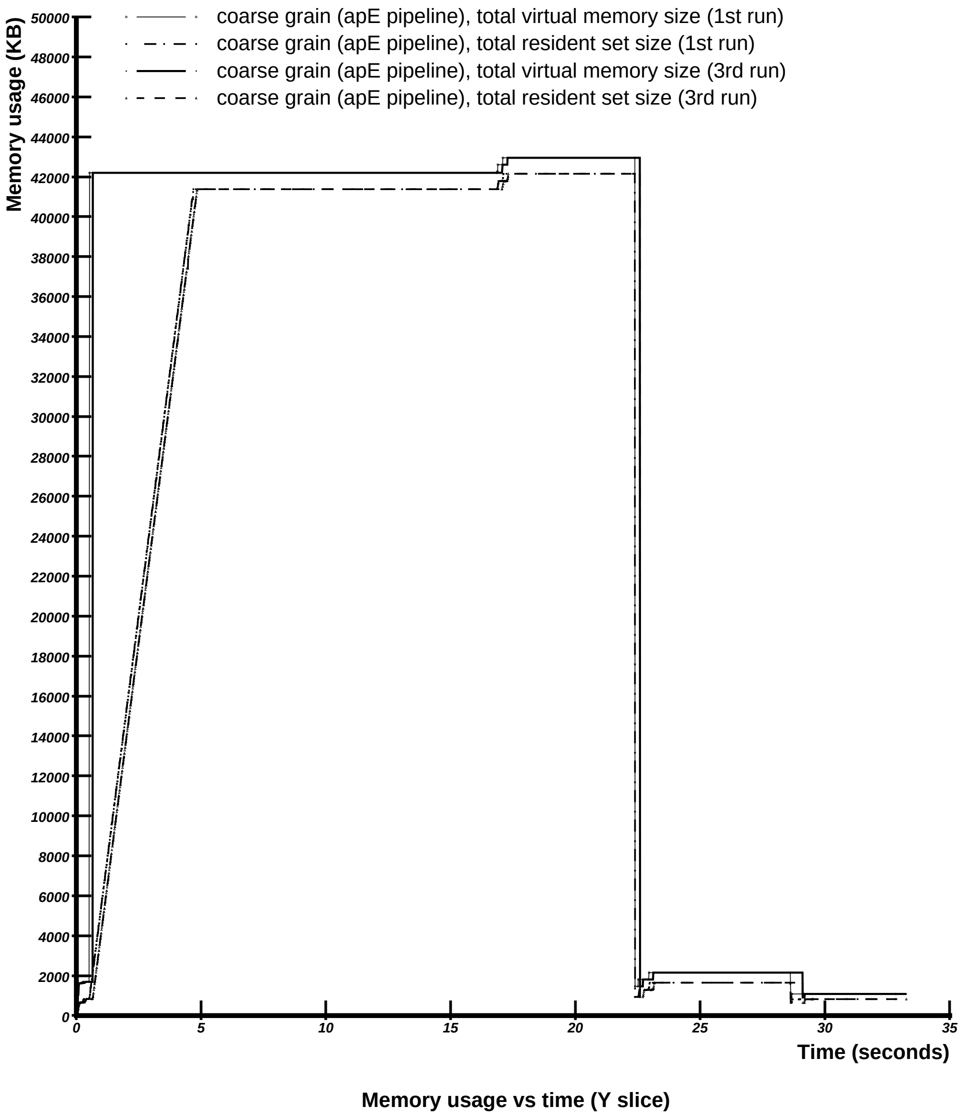


Figure 6.1: Memory usage of the apE Data Slicer pipeline. The input data was the shock-wave data in binary format.

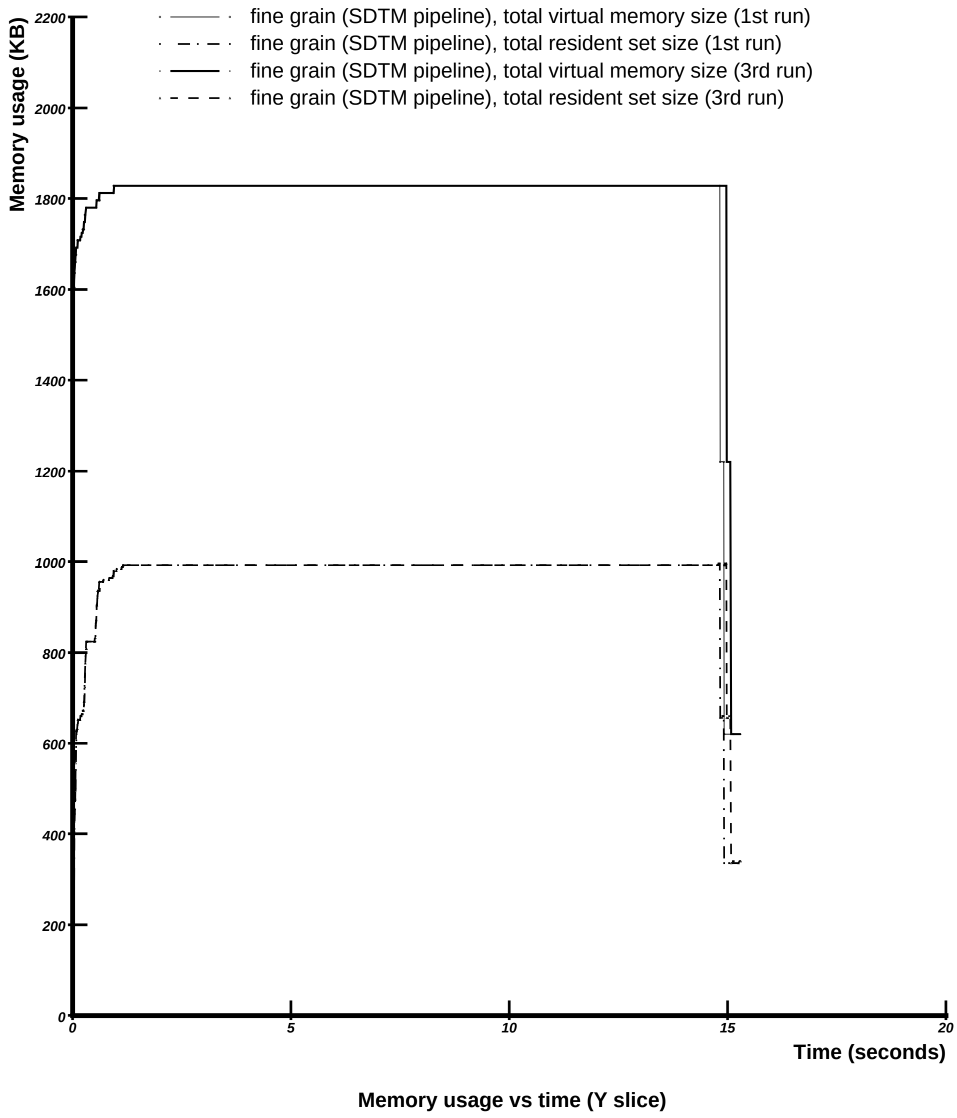


Figure 6.2: Memory usage of the SDTM Data Slicer pipeline. The input data was the shock-wave data in binary format.

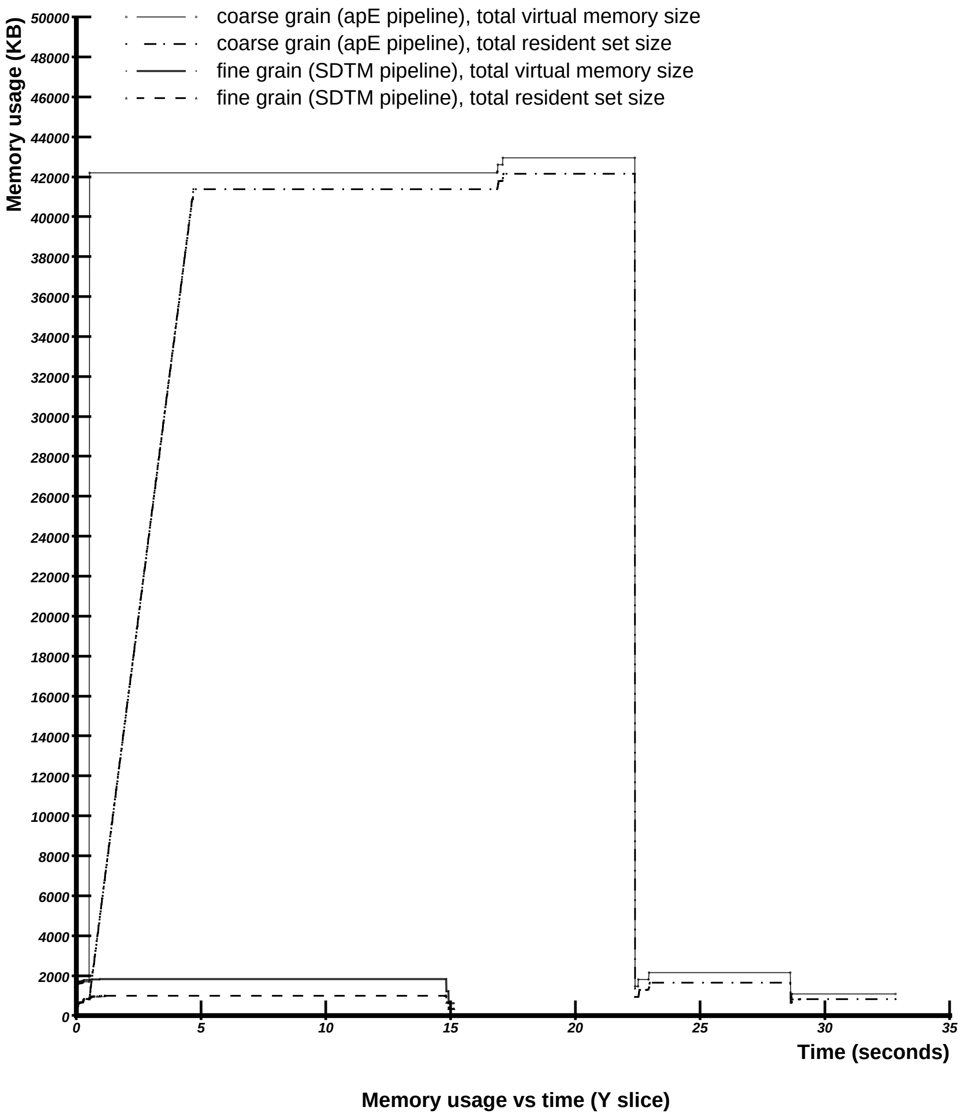


Figure 6.3: Memory usage of the apE and SDTM Data Slicer pipelines. The input data is the same as in the previous two plots. $MUR_v = 96\%, MUR_r = 98\%$.

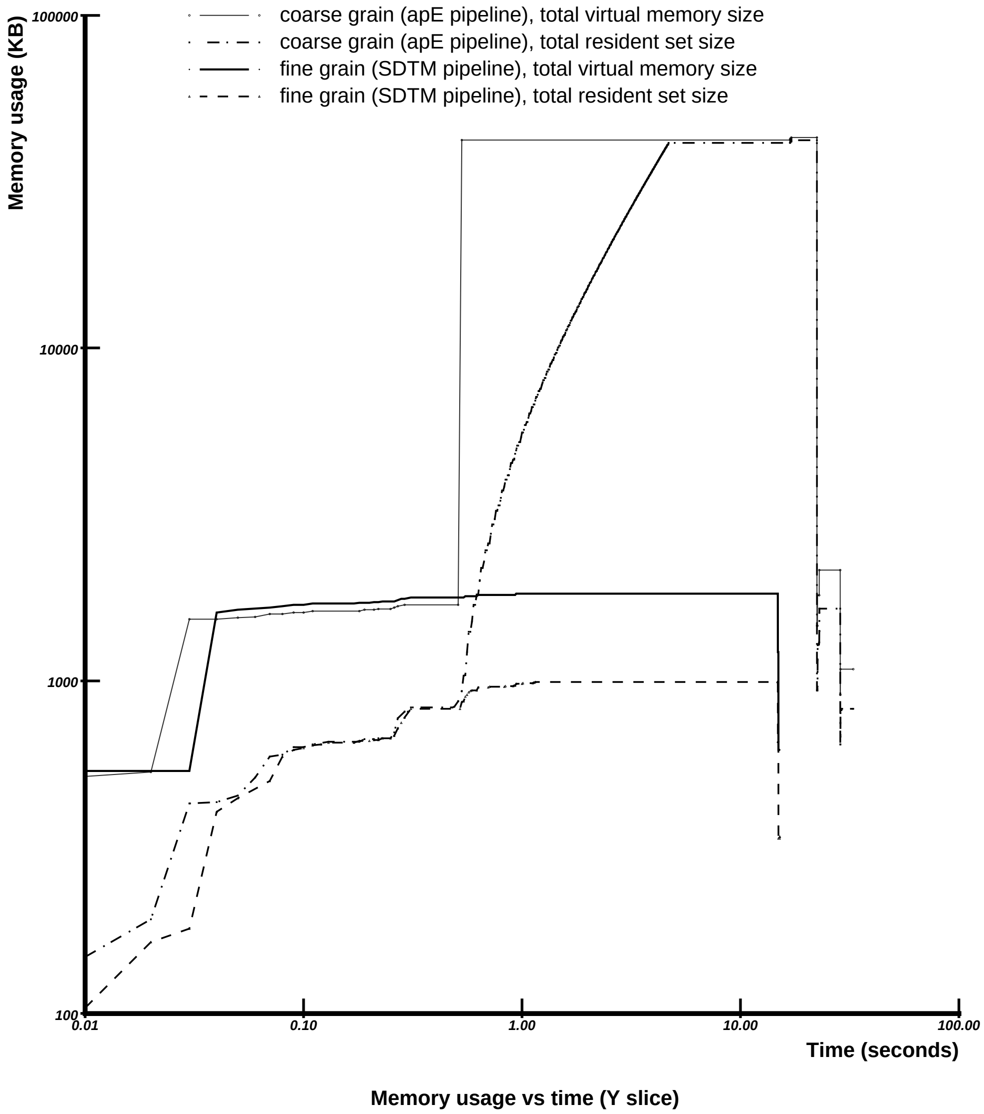


Figure 6.4: A closeup view of the memory usage of the Data Slicer pipelines.

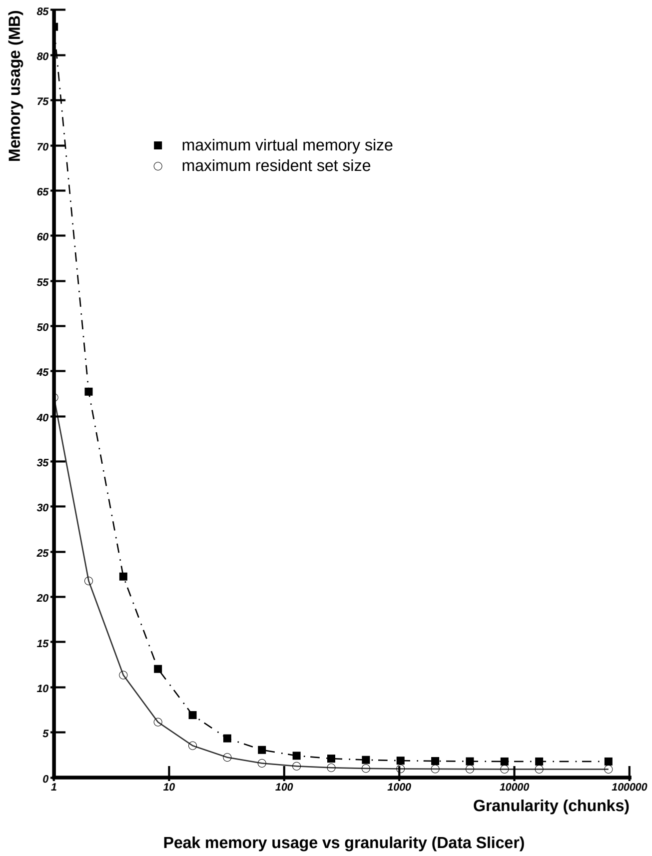


Figure 6.5: Peak memory usage vs granularity for the Data Slicer. Input data is in binary format.

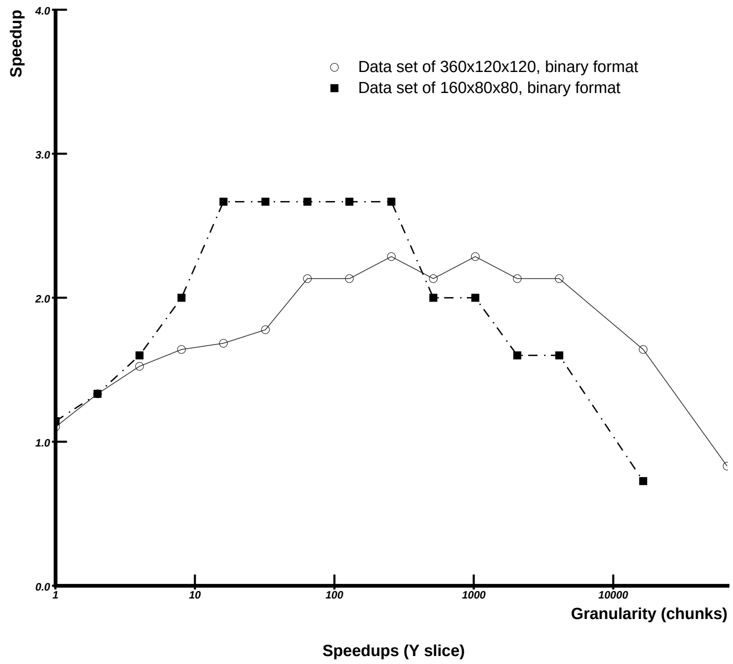


Figure 6.6: Speedup vs granularity for the Data Slicer.

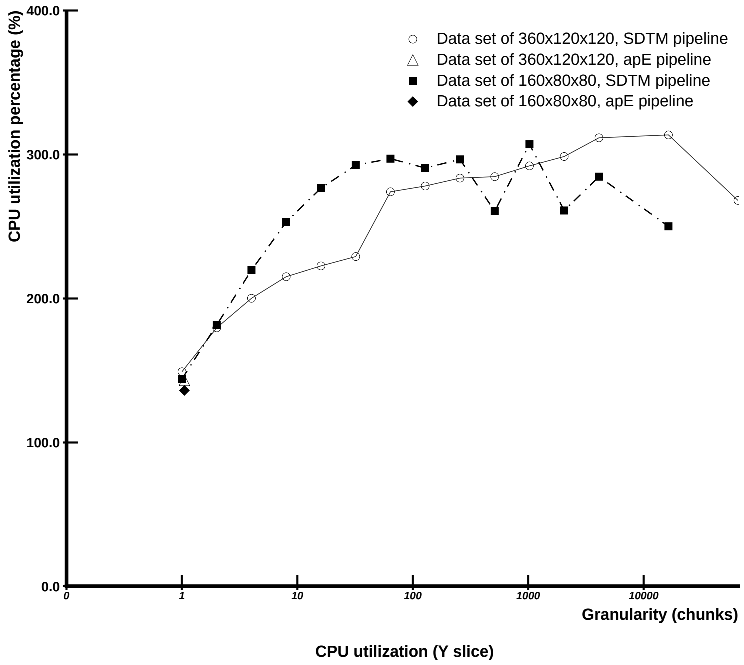


Figure 6.7: CPU utilization vs granularity for the Data Slicer. Input data is in binary format.

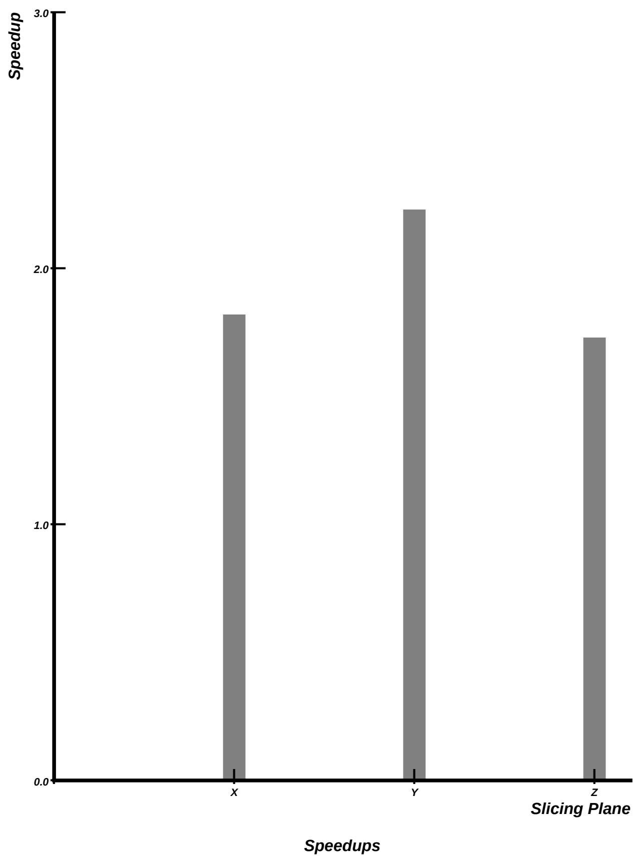


Figure 6.8: Speedups of different slicing planes. The input data was the shock-wave data in binary format.

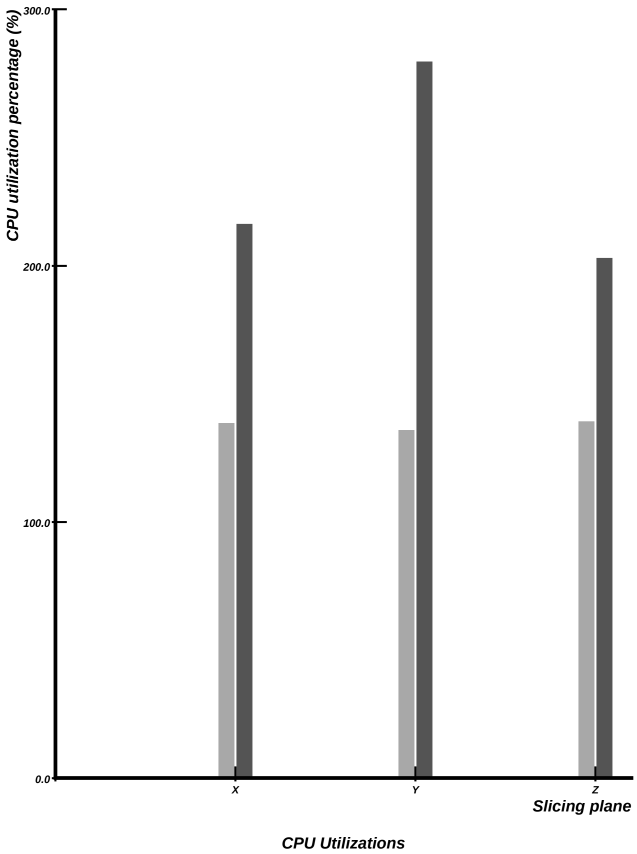


Figure 6.9: CPU utilization of different slicing planes. For each axis, the left bar represents the utilization of the coarse-grain approach and the right bar represents the utilization of the SDTM approach.

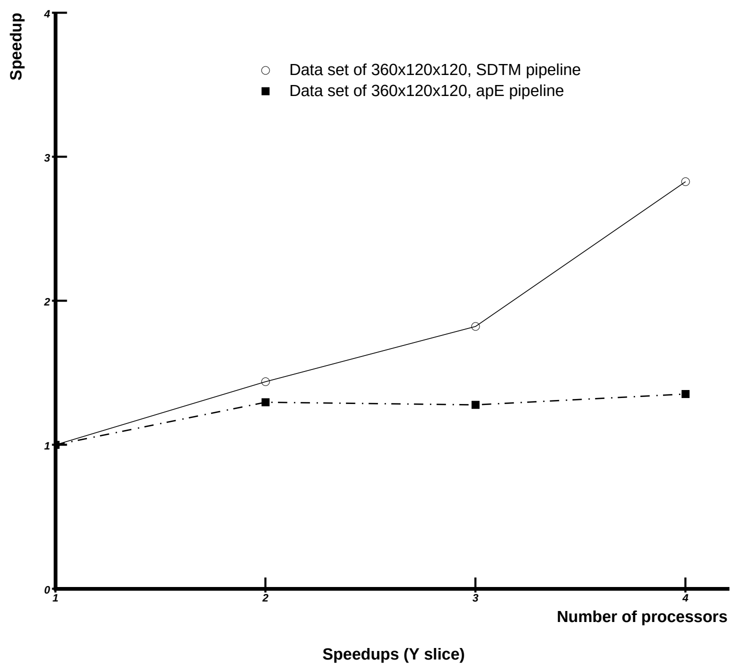


Figure 6.10: Speedup vs number of processors. Data is in binary format

6.2.2 The Iso-Surface Tool

The Iso-Surface tool is a linear pipeline shown in Figure 5.6. It is implemented as a Unix pipeline: `read | norm | onion | display`. The two data sets used for the Iso-Surface tool and the Volume Rendering tool are stored as single-precision floating point numbers both in binary format and in ASCII format.

This section discusses the performance of the Iso-Surface tool and addresses the following issues:

- How the coarse-grain pipeline is compared with the fine-grain pipeline in terms of memory usage during the execution. Measured memory usages are plotted and compared.
- How the granularity affects the performance. Memory usage, CPU utilization, and speedup are plotted as functions of granularity.
- Data-scalability: speedup and CPU utilization curves with input data sets of different sizes are compared and their similarities are shown.
- How lookahead symbols affect the performance. We show an example of how to decide the right number of lookahead tokens so as to reduce the idle waiting and to increase the performance. This is done with the help of recording synchronization events and drawing timelines.
- How the Iso-Surface tool performs under different hardware configurations. The same pipeline and data set was tested on different hardware platforms, and the performance results are compared.

The virtual and resident memory usage of a coarse-grain pipeline (at granularity of 1) and of a fine-grain pipeline (at granularity of 120) is shown in Figure 6.11, and the peak memory usage versus granularity is shown in Figure 6.12. The input data to the two pipelines is the shock-wave data set in binary format. The three peaks on the virtual memory curve for the coarse-grain pipeline represent the memory allocations requested by the modules. The rise of the first peak is caused by the `norm` module when it starts to receive input data from the `read` module, and the drop of the first peak is caused by the termination of the first module, `read`. Similarly, the rise of the second peak is caused by `onion` receiving data from `norm`, and the fall of that peak is caused by the termination of `norm`. The memory usage curves for the fine-grain pipeline are flat, again, because the lifetimes of the fine-grain modules are about the same as the lifetime of the pipeline.

The resident memory curve only reaches about half of the peak virtual memory usage, because binary data is represented by chunks for reasons of parsing efficiency (see Section 5.2.2). Each chunk is a data token recognizable by the lexical analyzer. In a coarse-grain module, the whole binary data set is regarded as one token. An $LL(1)$ parser must always have space for 2 tokens: the current token and the lookahead token. In the coarse-grain case, the second binary chunk never exists, thus the space is not used but is accounted anyhow. The problem is cured in the Volume Rendering tool, but is reported here to demonstrate how memory measurements can actually help find problems in system design and implementation. In reality, this problem brings little penalty because of the demand paging policy employed by most of today's operating systems.

Figure 6.13 shows the speedup of the iso-surface pipeline with both the shock-wave and the jet data sets using ASCII data stream running on the multiprocessor machine. The CPU utilization curves were measured during the same runs (Figure 6.14). Note that there is no performance drop in Figure 6.13 as there is in Figure 6.6. This is because the granularity of the Iso-Surface tool is based on slices, not on single data elements, thus the overhead cost does not outweigh the performance improvement.

Figure 6.15 shows CPU utilization curves for the Iso-Surface tool with correct and incorrect number of lookahead tokens. The two pipelines ran at granularity of 4. Both cases generate the correct results, but differ in the execution time. Timelines of the execution of the pipeline reveal why the incorrect number of lookahead tokens can slow down the execution. In the "incorrect" case, modules `norm` and `onion` both generate fewer lookahead tokens than required by the **Lookahead Rule** (Rule 3.2). This causes the input phases of `onion` and `display` to wait until the beginning of the next output phase of the upstream module just to get one lookahead token needed by the parser. This introduces busy waiting at the end of each input phase, thus decreasing the CPU utilization.

In a perfectly synchronized software pipeline like the top one in Figure 6.15, the jth output phase O_j of module i is equal in time to the jth input phase I_j of module $i+1$, and the end of the jth output phase O_j of module i triggers the jth processing phase of module $i+1$. But in the case of incorrect number of lookahead tokens, illustrated by the example at the bottom of Figure 6.15, the above property is no longer true. For example, the I_1 phase of `onion` takes longer than necessary, and the P_1 phase of `onion` starts shortly after the beginning of the O_2 phase of `norm`, as if the beginning of O_2 of `norm`, instead of the end of O_1 as in the first case, triggers P_1 of `onion`. The same relationship exists between `norm` and `onion` in other cycles. The delay inflicted on `onion` in the first cycle is approximately $I_2 + P_2$ of `norm`. The delay is propagated to the second cycle and so on, because the O_2 of `norm` is also delayed waiting for P_1 of `onion` to finish, and the delay of the O_2 of `norm` causes I_2 of `onion` to delay, too. Since `onion` is the bottleneck in the pipeline, the total execution time is increased due to the cumulative delays in each cycle caused by the incorrect number of lookahead tokens.

Pipeline	Total memory	Single-processor Speedup	Single-processor CPU utilization	Multi-processor Speedup	Multi-processor CPU utilization	Distributed Speedup	Distributed CPU utilization
Coarse grain	14.7 MB	1.0	94%	1.0	140%	1.0	139%
Fine grain	0.53 MB	1.17	99%	1.78	252%	1.76	241%

Table 6.1: Performance comparison of coarse-grain and fine-grain implementations of an iso-surface construction pipeline in different hardware environments.

Table 6.1 summarizes the performance improvements by the fine-grain pipeline in different hardware environments. The data set used to produce these results is the jet data in ASCII format. The coarse-grain scenario is the Iso-Surface tool running at granularity of 1. The fine-grain scenario is the Iso-Surface tool running at granularity of 80 (finest granularity possible). From the table, we can see that fine-grain pipelines perform better than coarse-grain pipelines in all measurements except the speedup in the single-processor environment. The most impressive improvement is probably the reduction in memory usage: a 96% reduction. 78% improvement in execution time is also a noticeable gain. The 100% plus CPU utilization by coarse-grain pipelines is due to the overlap between the output of one module and the input of the next module. Fine-grain pipelines utilize more of such overlaps. The table also shows that the speedup is fairly consistent in both the multiprocessor and distribute systems. The little speedup on the single-processor system is due to fewer page swaps by the operating system when running fine-grain pipelines. A more dramatic case of this nature will be discussed in the next subsection involving a larger data set and far more page swaps. Note that all the tests were done using a single time step. If multiple data sets are fed into the pipeline, there will be no noticeable speedup, but the same degree of reduction in memory usage should still be observed.

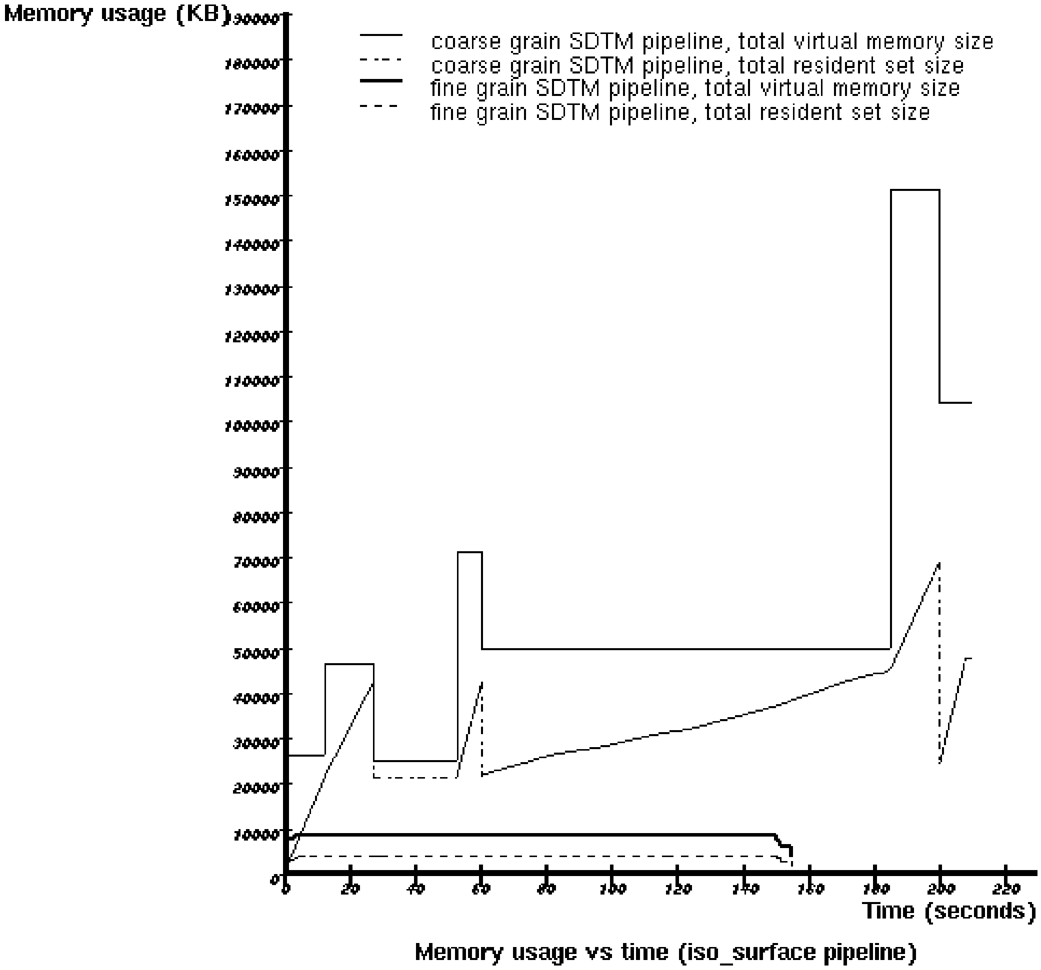


Figure 6.11: Memory usage of the Iso-Surface tool. The input data was the shock-wave data in binary format. $MUR_v = 95\%, MUR_r = 95\%$.

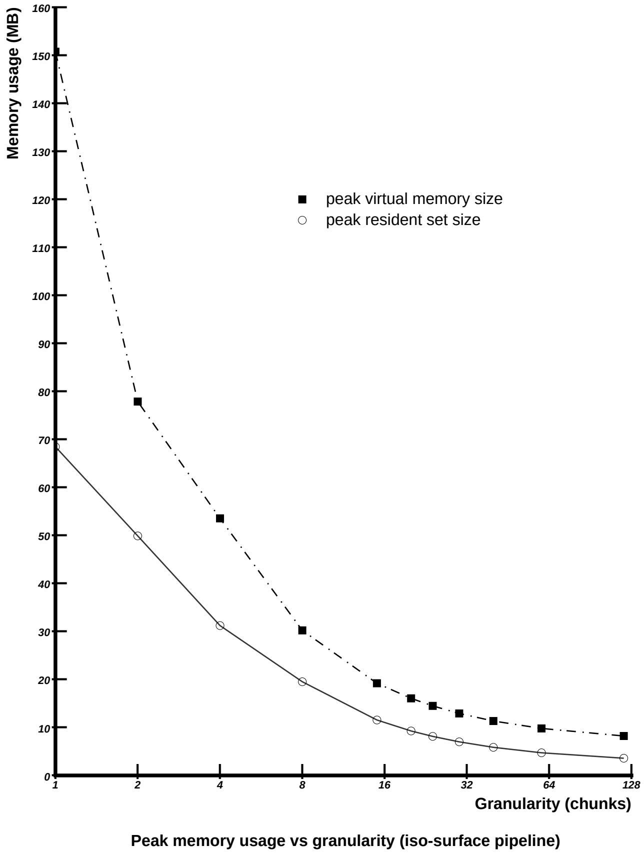


Figure 6.12: Peak memory usage vs granularity for the Iso-Surface tool. The input data is the shock-wave data set in binary format.

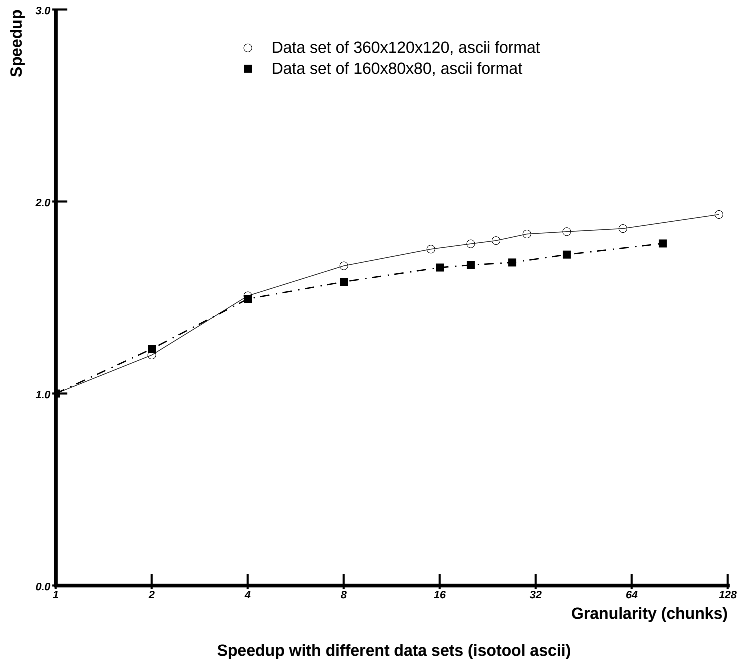


Figure 6.13: Speedup vs granularity for the Iso-Surface tool.

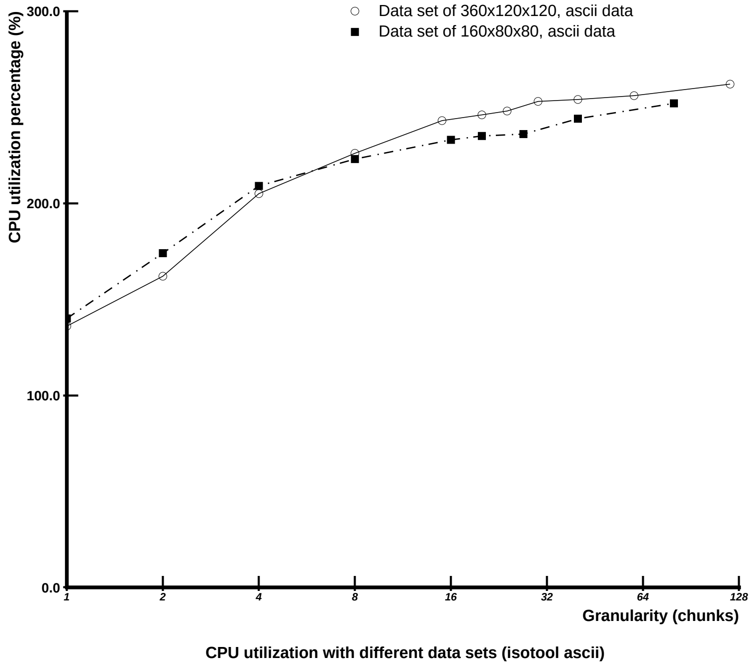
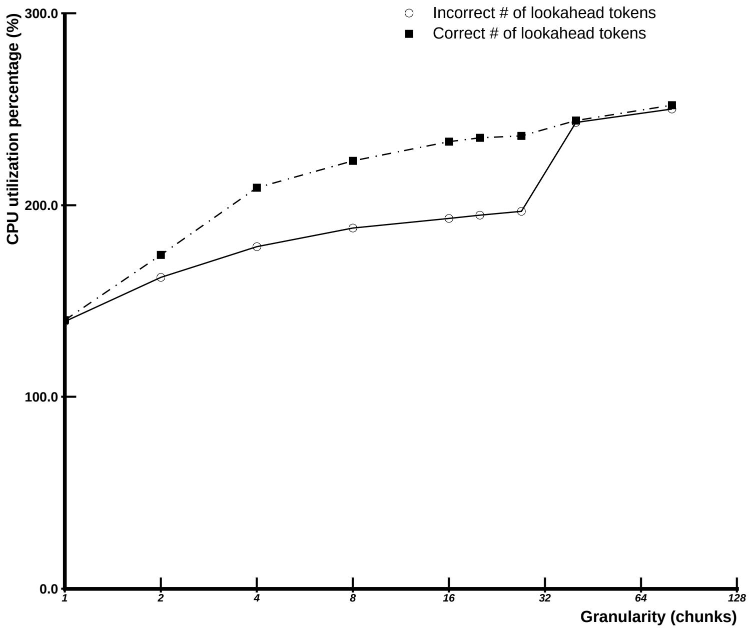


Figure 6.14: CPU utilization vs granularity for the Iso-Surface tool.



CPU utilization with different # of lookahead tokens (isotool jet ascii)

Figure 6.15: Correct and incorrect number of lookahead tokens.

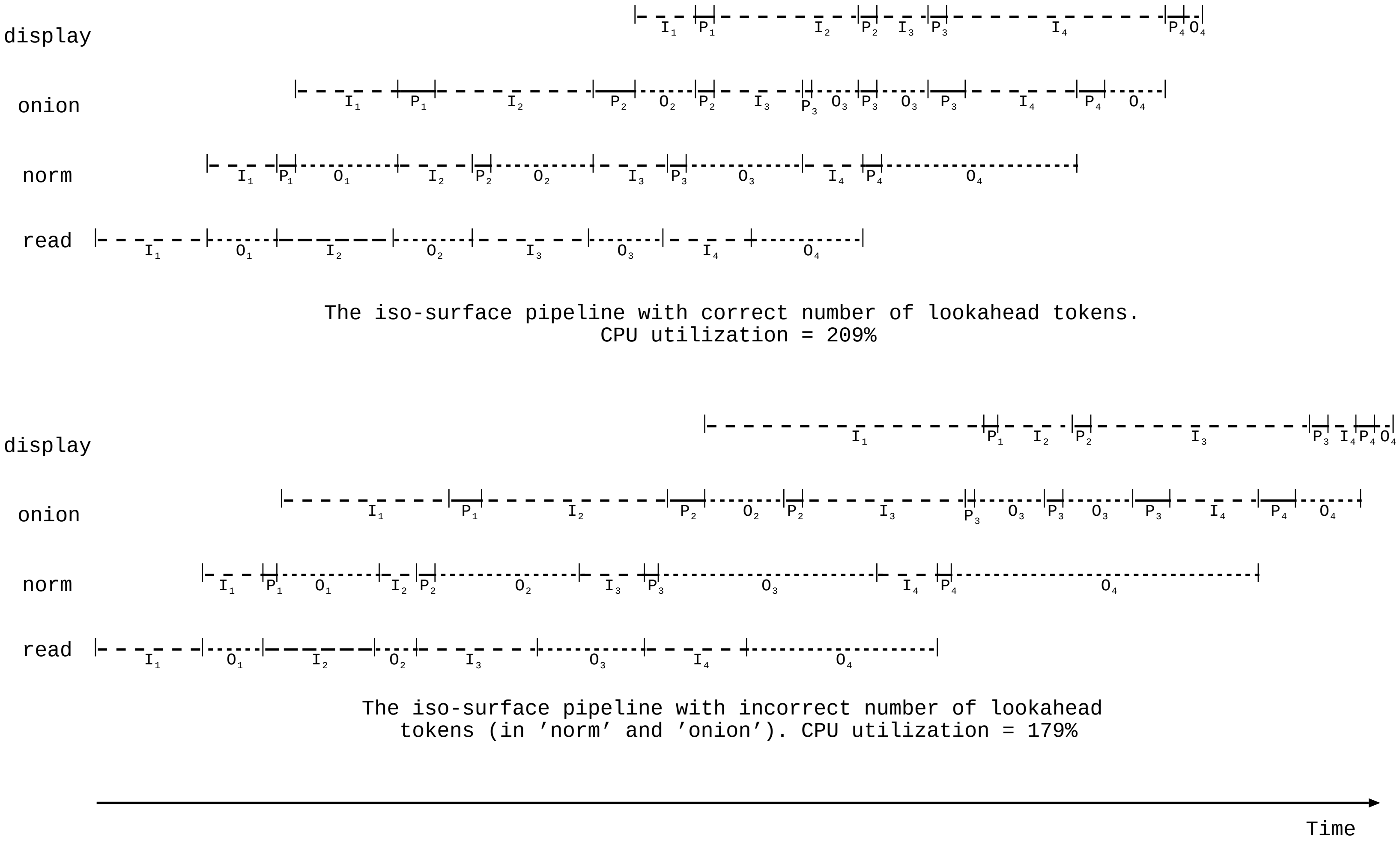


Figure 6.16: Timelines of two pipelines having different number of lookahead tokens.

6.2.3 The Volume Rendering Tool

The volume rendering tool is the linear pipeline shown in Figure 5.9, implemented as a Unix pipeline: `read | norm | class | ray`. This section examines the following issues:

- How the coarse-grain pipeline is compared with the fine-grain pipeline in terms of memory usage during the execution. Again, memory usage measurements are plotted for comparison.

- How the granularity affects the performance. Memory usage, CPU utilization, and speedup are plotted as functions of granularity.

- Data-scalability: The speedup curves for different input data sets are compared, and they appear to be very similar.

- Binary vs ASCII data: The different data formats cause significant difference in measured performance. In either case the pipeline can achieve a certain degree of overlapping among modules. For binary data streams the overlap happens in the processing phase, whereas for ASCII data streams more overlap happens in the I/O phase.

- Large data sets: A large input data set can cause the coarse-grain pipeline to run out of swap space. This problem is addressed by the fine-grain approach.

- How the Volume Rendering tool performs under different hardware configurations. The same pipeline and data set was tested under different hardware configurations, and the performance results are compared.

- How the empirical performance matches the analytical model. The analytical performance model developed in Section 3.3 is compared with the measured performance data. The two are found to match very closely.

The comparison of the memory usage by a coarse-grain pipeline and a fine-grain pipeline is shown in Figure 6.17, and the peak memory usage vs granularity is shown in Figure 6.18. Like the curves for the Iso-Surface tool, there are three peaks on the virtual memory curve of the coarse-grain pipeline, and the memory usage curves for the fine-grain pipeline are mostly flat. One difference with the Iso-Surface pipeline is that the resident set size is no longer half of the virtual memory size (toward the of the curve) because of an improvement in the handling of binary chunks. In Iso-Surface tool, a binary chunk equals a binary token. So an input binary data set of 20 MB will cause a coarse-grain LL(1) module to allocate 40 MB of memory for 2 tokens. In the Volume Rendering tool, chunk size and token size are separately defined – a binary token can be much smaller than a chunk defined by the user. For a coarse-grain module in the Volume Rendering tool taking the same input set as above, if a binary token size is set at 2KB, only 20MB + 2KB memory space is needed.

Figures 6.19 and 6.20 show similar speedup and CPU utilization curves for the pipeline taking different data sets. The two curves match very well. This shows the data scalability of the volume rendering pipeline. The data values are in binary format in these tests.

Figures 6.21 and 6.22 show the pipeline performance with respect to the same data set but in different formats. Here the ASCII format means that the data stream flowing between all the modules is in the ASCII format, and the binary format is similarly defined. Note that in Figure 6.22, the utilization curve with ASCII data stream has a value bigger than 100% at granularity of 1, because of the overlapping of I/O between neighboring modules which takes large portions of the execution time. In the pipeline using a binary data stream, the I/O time is so insignificant that at granularity of 1 there is virtually no overlapping between modules. Thus the CPU utilization in this case is 100%. The execution time of the pipelines using binary and ASCII formats can differ by a factor of 4.

Figure 6.23 is a comparison of the speedup curve on shock-wave data in binary format shown in Figure 6.21 with the speedup curve predicted by the analytical model represented by Equation 3.12. The analytical model matches very well with the empirical result, which means that the analytical model has included major factors determining the performance of a fine-grain pipeline.

		Single-processor		Multi-processor		Distributed	
Pipeline	Total memory	Speedup	CPU utilization	Speedup	CPU utilization	Speedup	CPU utilization
Coarse grain	20.4 MB	1.0	96%	1.0	124%	1.0	138%
Fine grain	1.88 MB	1.01	98%	1.84	228%	1.44	202%

Table 6.2: Performance comparison of coarse-grain and fine-grain implementations of a volume rendering pipeline in different hardware environments.

Table 6.2 shows the maximum speedup and CPU utilization achieved using the fine-grain volume rendering tool in different hardware environments. The data set used is the jet data set in ASCII format. This table is very similar to Table 6.1. So all the conclusions reached there can be applied here as well.

Pipeline	Time in user mode	Page swap-in/sec	Page swap-out/sec	Ave. run-queue length	In-memory time	Speedup
Coarse grain	54%	15.4	0.57	1.1	57%	1.0
Fine grain	98%	0.03	0	1.9	100%	1.88

Table 6.3: Performance comparison of coarse-grain and fine-grain volume rendering pipelines on a single processor using a large data set.

Another experiment was done on a Personal Iris with 16 MB of main memory and 75 MB of swap space. The purpose of this experiment was to test the performance of the volume rendering pipeline under heavy memory strain. The large data set (shock wave) in binary format was used. When the pipeline was run with granularity of 1, it eventually ran out of swap space and aborted prematurely. The pipeline was run again with a granularity of 2, and it succeeded this time. The system activities during the execution of the pipeline at granularity of 2 was recorded using the IRIX command `timex`. The result was compared with that of the pipeline with the finest granularity of 120. The numbers are presented in Table 6.3. Because of too many page swaps, modules in the coarse-grain pipeline spent almost half of their time on block state waiting for pages transferred between the main memory and the hard disk. This significantly slows down the execution of the pipeline. From both Table 6.2 and Table 6.3, it can be seen that fine-grain pipelines have no apparent advantage over coarse-grain pipelines in a single-processor environment unless the data set is large enough to cause page swaps.

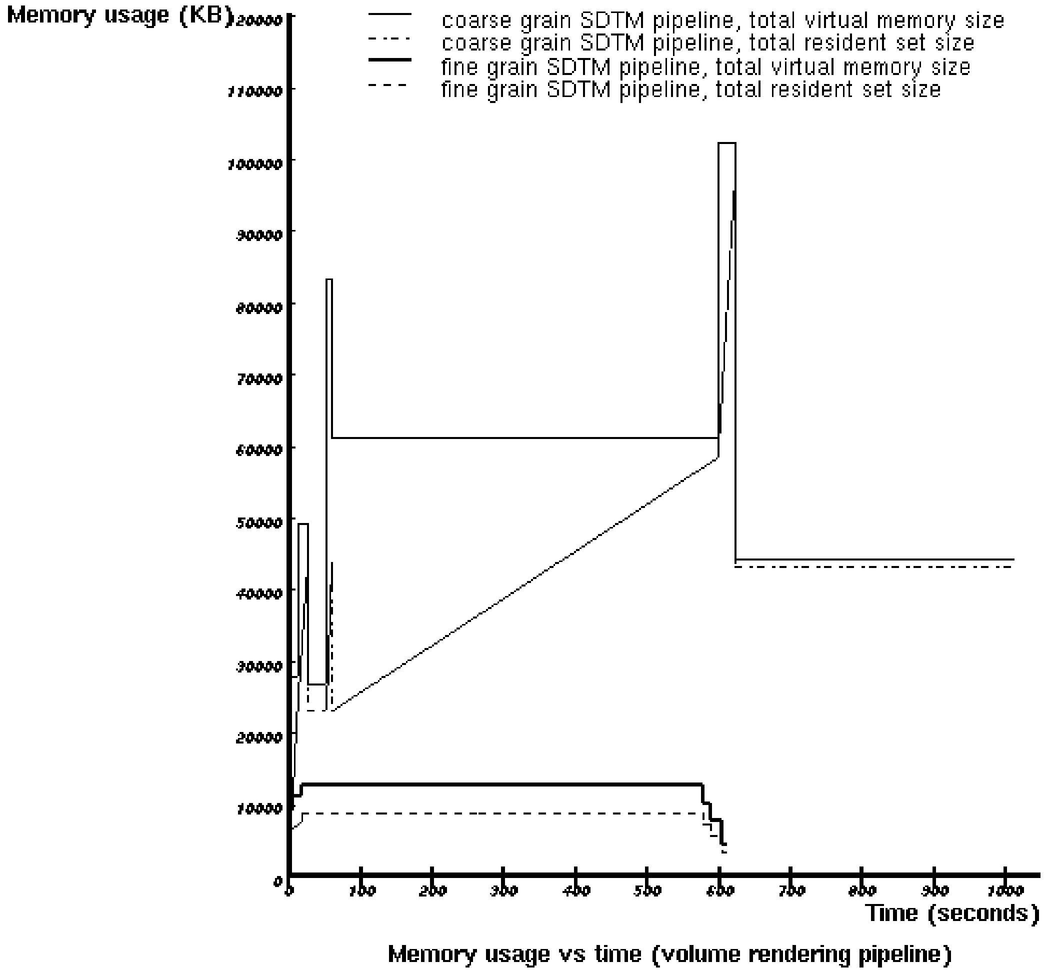


Figure 6.17: Memory usage of the Volume Rendering tool. Input data is the shock-wave data set in binary format. $MUR_v = 89\%, MUR_r = 91\%$.

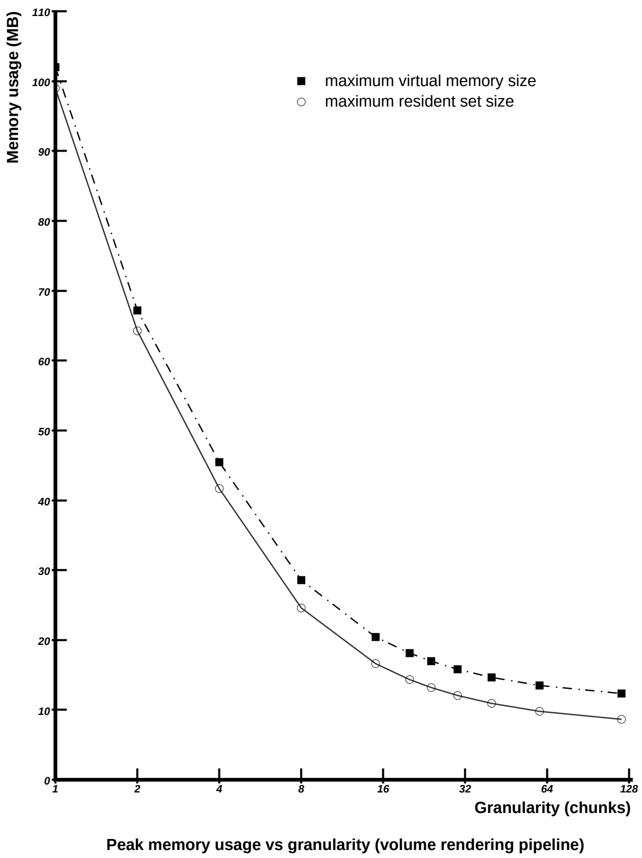


Figure 6.18: Peak memory usage vs granularity for the Volume Rendering tool.

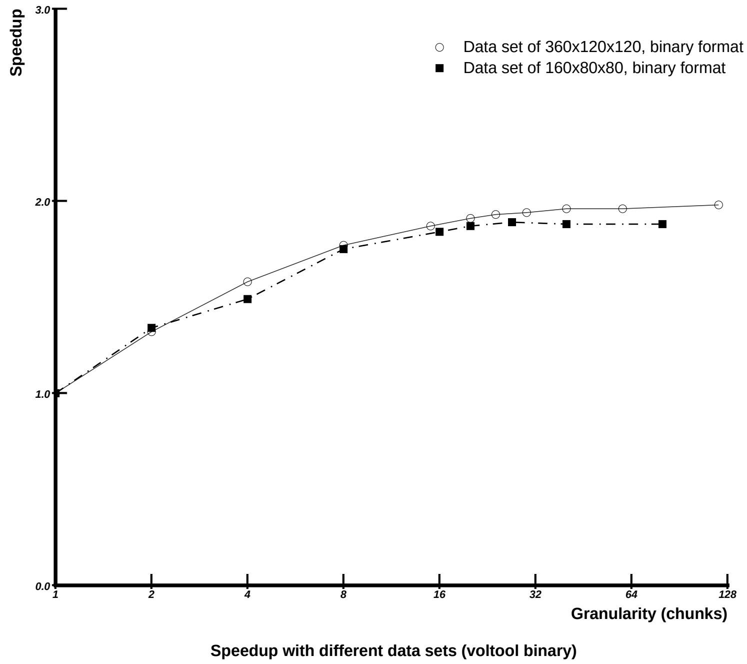


Figure 6.19: Speedup vs granularity for the Volume Rendering tool.

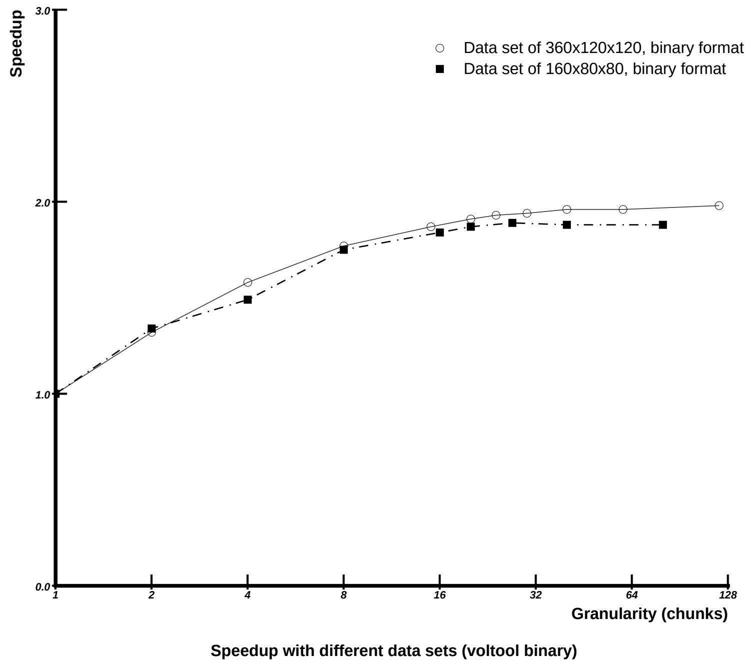


Figure 6.20: CPU utilization vs granularity for the Volume Rendering tool.

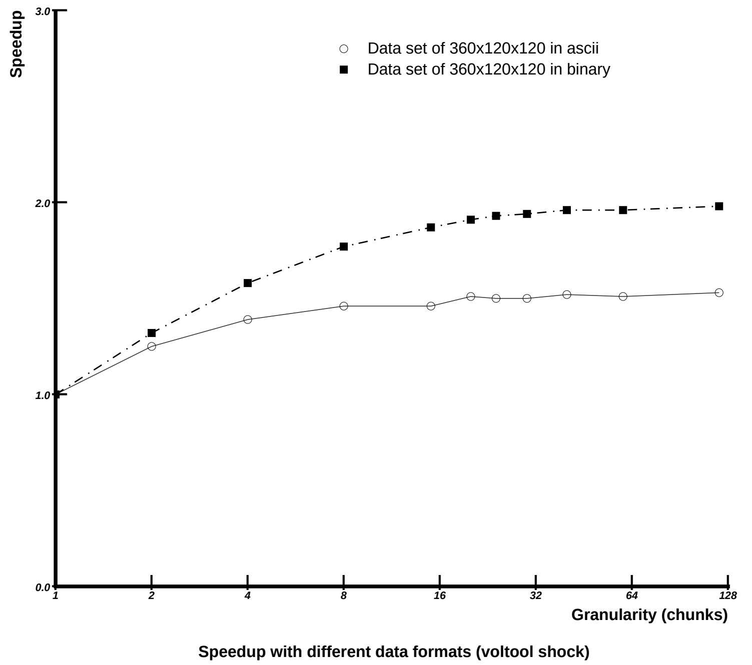


Figure 6.21: Speedup of the volume rendering tool with different data formats.

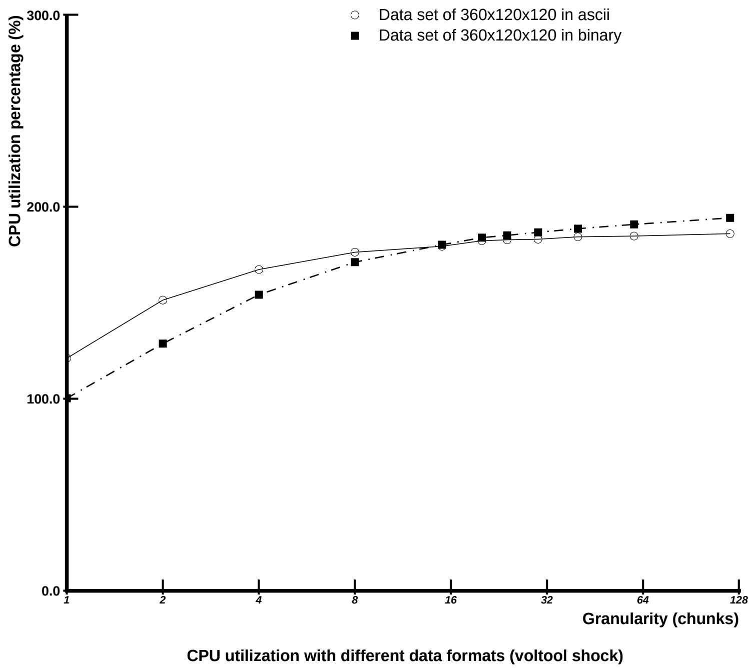


Figure 6.22: CPU utilization of the volume rendering tool with different data formats.

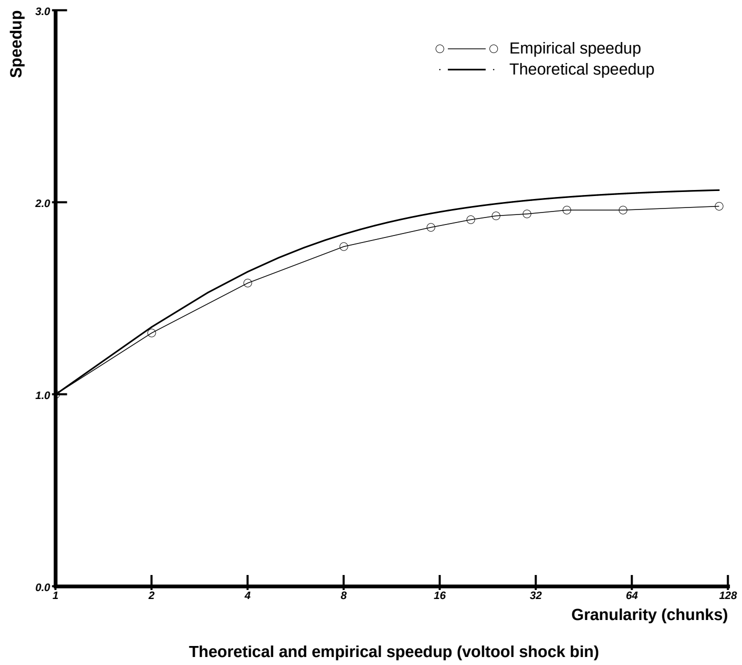


Figure 6.23: Comparison of an empirical speedup curve with the analytical curve.

6.3 Conclusions

Reduced memory usage is the most obvious advantage of using fine-grain pipelines. This advantage is available regardless of the hardware environment and of whether the pipeline is running in the experimental phase or the production phase. The memory usage reduction for the three test pipelines ranges from 89% to 98%.

The speedup advantage may disappear under some circumstances, such as in a single-processor system or when the pipeline is running in production mode. But if the hardware system is under heavy memory strain such as processing large data sets or competing with other memory-intensive jobs, then fine-grain pipelines can outperform coarse-grain pipelines again.

A key question for the implementation of a module is: what is the granularity of the data that will yield the best results? The question is not easy to answer, because there are many factors that can affect the performance of a pipeline. It is possible that there is no single "optimal" granularity that will provide the best performance in all situations. A rule of thumb is to avoid granularities at the two extreme ends. Sometimes the control of granularity is limited by the algorithm impelmented in a module.

One impact of the fine-grain dataflow computing model is that the module writer needs to choose the best algorithm for working with streams of data. The fine-grain algorithms designed for the SDTM pipelines in this thesis are examples of this.

Chapter 7

Conclusion

7.1 Summary

In this thesis we developed a programming model (SDTM) for fine-grain dataflow computing, a performance model for analyzing the speedup of fine-grain pipelines over coarse-grain pipelines, several fine-grain visualization algorithms, a module generator based on the SDTM model, the implementation of three real-life visualization pipelines, and performance measurements of these pipelines.

The SDTM model combines the advantages of the dataflow programming paradigm with the syntax-directed translation techniques. It allows fine-grain computing in a dataflow environment. Specification and granulating of scientific and geometric data, as well as association of fine-grain computing functions to data chunks, are the essential ideas in SDTM. The attribute grammar-based automatic module generator converts these high-level descriptions into conventional language source code.

The theoretical performance model shows how much improvement can be obtained by using fine-grain models. In a multicomputer environment where each module in a pipeline is guaranteed a processor, linear speedup with respect to the number of stages in the pipeline can be obtained if the fine-grain model is used. The performance models are based on some sound assumptions. The second model was tested both in a contrived pipeline and in a real-life pipeline, and is proved to be fairly accurate. The three fine-grain visualization pipelines implemented fine-grain algorithms and showed performance improvement over coarse-grain pipelines in terms of memory consumption, execution time, and CPU utilization. Memory usage reduction of over 90% and speedup of 2 times have been achieved by all the test cases.

Reduced memory usage is probably the most significant feature of the fine-grain dataflow system. Several visualization algorithms have been proposed and their data locality property was exploited. The data locality property allows a dynamically adjusted moving window on the input data stream. This results in the sublinear memory requirement for these algorithms which hold the promise of processing very large data sets.

This work also shows that ease of programming and run-time efficiency are sometimes conflicting goals. It is much easier to program a coarse-grain module than a fine-grain module. In a coarse-grain module, the module builder provides only one compute function, having access to the entire data set. In a fine-grain module, there are many small semantic functions associated with small chunks of data. Their access to data is limited to a certain locality within the data stream.

Fine-grain modules can co-exist with existing coarse-grain modules, because the interface between modules is well defined by the data language. So existing coarse-grain systems can gradually migrate to fine-grain systems. This shows another advantage of the dataflow systems, that a single module can be unplugged and replaced by a better one, as long as the input and output data format conforms to the specifications of the module.

But in order to realize fine-grain algorithms, some modifications must be made to the existing data languages designed originally for coarse-grain dataflow models. We have made several changes to apE's Flux language to accommodate the needs of the fine-grain algorithms. However, the changes are selective, and there is no need to redesign the whole language.

7.2 Contributions

Current dataflow visualization systems are all built on the pure dataflow model. The adoption of this simple model has let to inefficient use of the computer memory. Because scientific data sets are inevitably large, this inefficiency problem has hindered current systems' ability to process very large data sets. The fine-grain dataflow model (SDTM) was proposed to provide a solution to this problem. In trying to do so, we've brought about other benefits such as speedup and improved CPU utilization. The fine-grain model is intended to increase the run-time efficiency of visualization pipelines, not to improve the speed of any individual algorithm.

In summary, the research described in this thesis has significance in the following areas:

- Visualization Software Systems
  - The SDTM system improves the execution speed of visualization pipelines in the experimental phase. Speedups of over twofold have been observed. Greater speedups are expected of longer pipelines to be implemented in the future.
  - The SDTM system drastically reduces the memory requirement of a visualization pipeline. Sublinear-memory visualization algorithms were designed under this framework. There are fine-grain algorithms whose minimum memory requirements are independent of input data size. There are algorithms whose minimum memory requirements are sublinear in terms of input data size. There are also coarse-grain visualization algorithms in which data decomposition cannot be done. The SDTM model provides a framework within which all kinds of algorithms can be implemented.
  - The automatic module generation tool provides a way of defining visualization modules at a high level, and relieves programmers from concerning about data parsing, error handling, communications, and synchronization.

- Dataflow Computing

  The fine-grain dataflow model is a contribution to dataflow computational models. Previous research on dataflow computing has studied determinancy [88, 85], equivalence [88, 57], load balancing [27], basic models [15, 16] and various extended models [56] concerning node types and firing rules. But little has been done to provide ways of describing complex and hierarchical data tokens and expressing computations inside a node [6]. Our model enhances the notions of data tokens and nodal firing. SDTM extends the traditional task-level dataflow model by introducing the concept of granularity. It distinguishes from other models in that it provides a way to specify complex, hierarchical token structures and computations associated with them. Task-level dataflow systems are often labeled coarse-grain. We have demonstrated that fine data grain computation is feasible and in fact more efficient.

- Fine-Grain Visualization Algorithms

  Several popular visualization algorithms have been invented or redesigned to work within the fine-grain dataflow model. These algorithms are one-pass algorithms and require sublinear memory space. The superiority of these new algorithms have been demonstrated by the performance measurements. The impact of the fine-grain dataflow model is to force the module designers to carefully choose the appropriate algorithms and to implement them in a more efficient way. SDTM is the first visualization system that's built on a fine-grain dataflow model and being thoroughly studied. The implementation of an SDTM system shows that the concept of fine-grain dataflow computing can be applied to a variety of visualization applications.

- Attribute Grammar

  - Attribute grammar techniques have been successfully applied to a visualization system. An existing compiler construction tool was extended in its ability to handle attribute specifications. A new attribute storage reduction technique called "attribute substitution" was developed.
  - A new property concerning the back-propagation of the number of lookahead tokens was discovered for parsers connected into a linear pipeline. A formula was given to calculate the exact number of lookahead tokens that must be generated by each module in the pipeline.
  - The lexical analyzer of PCCTS was extended to handle binary data tokens. Traditionally scanners only handle ASCII data. Scientific data stored in ASCII format takes much more space than in binary format, and takes significantly longer time to convert and communicate than binary data.

7.3 Future Work

The approach studied in this thesis should be applicable and beneficial to all kinds of dataflow visualization systems regardless of their execution models. Both systems like IBM Data Explorer, which use the all-modules-in-one-executable model, or systems like AVS, which use the one-executable-per-module model, can benefit from using this fine-grain model. This computational model is also flexible enough to be implemented on a wide variety of architectural platforms such as shared-memory, message-passing multiprocessor systems, or loosely coupled distributed systems.

The SDTM method is also applicable to the visualization of nested-grid simulations, because the context-free grammar and attributed definitions provide a natural means to represent the nested data structures and operations on them.

Dividing the data stream into a more refined granularity increases the CPU utilization and the concurrency among modules. The longer the pipeline, the bigger potential there is to improve performance using fine-grain modules. It is not difficult to find long pipelines (over 50 modules) nowadays in real-life applications [74, 96]. It would be interesting to see how much improvement can be achieved with long pipelines. It is expected that the results should be more dramatic than those of short pipelines. With longer pipelines, the calculation of the number of lookahead tokens should be done automatically. These numbers are calculated manually now.

In distributed systems, running a fine-grain visualization pipeline can also reduce the burden on the network. Coarse-grain pipelines tend to generate bursts of large-volume data traffic on the network, whereas fine-grain pipelines generate small but more continuous data streams. Although this issue is not pursued in this thesis, it is an interesting topic by itself.

The central concept of distributed systems is “pluggability” of hardware, software, and application structures across distances of arbitrary choice [62]. The dataflow model seems to be a good choice to achieve that goal. However, the performance of many existing dataflow systems has becomes the biggest obstacle in claiming their superiority over traditional systems. Using the one-process-per-module approach as in AVS and Iris Explorer has led to extremely inefficient, although very flexible, systems. On the other hand, putting all modules in one executable as in IBM Data Explorer does not work well in distributed environments. Perhaps the future system would retain the dataflow flavor at the user interface level and employ sophisticated techniques to “compile” the pipelines before running them. The intelligent compiler will have automatic load balancing capabilities. It can regroup the module functions and generate schedulable units in such a way that the some optimal measurements can be achieved.

Future research opportunities also exist in the refinement and extension of the fine-grain model. The fine-grain model and the module generation tool provide a good framework to start with. But there are many other issues that could have better solutions. Multiple input and output ports could be added to the SDTM system. This has been touched upon in Section 3.2.2.2 but more work needs to be done. Selecting a good chunk size is still an art at this point. We are planning to work on ways that can determine the best chunk size automatically. In the current implementation, data granularity is decided at the pipeline construction time. To decide the optimal granularity we may have to take into consideration such things as the size of the input data set, synchronization cost, and the underlying communications mechanism.

Fine-grain algorithms can also be parallelized. A balance has to be found between the granularity of the input data and the granularity of parallelization. This has practical values in massively-parallel systems. IBM Data Explorer is a typical medium-grain visualization system that utilizes a special parallel architecture called the POWER Visualization Server (PVS). SDTM is a fine-grain system using pipelined execution model. The two models can be combined to suit a specific architecture. Take the fine-grain module `norm` and the jet data set, for example, the minimum chunk size is one data element. Using the minimum chunk size as the data granularity adds too much overhead, as the module would do input, compute, and output for 1 million times. This choice doesn't yield the best performance (See Figure 6.6). The best choice is to make the grain size about 1000 data elements and let the module do *IPO* for 1000 times. This still leaves some room for parallelization if the hardware allows it. The same strategy is applicable in the iso-surface extraction module and the volume rendering module, where the minimum chunk size is a 2D slice from a 3D array. Finding the balance between fine granularity and intramodule parallelism can be an interesting topic in the future.

Appendix A

The Flux Format

Flux is a data specification language designed for the dataflow system apE. Its primary design goals are: machine independency, extensibility, and grouping mechanisms for hierarchical definition of data types. Flux is a general format capable of representing arbitrary data structures. Anything that is transmitted between modules is defined in Flux. The Flux format consists of three basic building blocks: fields or commands, arrays, and groups. A *field* or *command* takes up a single line. It begins with a keyword followed by a set of arguments. Fields and commands are identical and are distinguished by use alone. Fields store data and are usually part of group structure, whereas commands direct the action of programs. *Arrays* have the form keyword[integer] type, which is similar to the syntax of arrays in conventional programming languages. A *group* is a collection of any Flux entities surrounded by begin and end lines. Examples of Flux format are given in the following diagram:

```
(1) a field representing the minimum and maximum of a data array:
     range 0 7

(2) an integer array represented in ASCII:
     data[2] int ascii
     0
     1

(3) a variable group:
     begin variable "foo"
            notes[1] string ascii
                "This is a variable group"
```

```
        ^spatial_ref grid:"foo.grid"
        range 0 7
        data[2][2][2] float ascii
        0.1
        0.2
        0.3
        0.4
        0.5
        0.6
        0.7
        0.8
end variable "foo"
```

Appendix B

An Attribute Grammar Specification of the Module norm

```
/*
 * AG grammar for the module 'norm'.
 */
#header <<
#include "apE_hs.h"
#include <math.h>
#include "attrib.h"
#include "var.h"
>>
<<
#include <sys/types.h>
#include <malloc.h>
#include <string.h>
extern FILE *fp;
int  SDTM_lookahead = LL_K;
int  bc_size  = SDTM_BUF_SIZE; /* binary chun size */
int     type     = TYPE_LINEAR;
int     format   = FORMAT_ASCII;
float   min_val  = MINFLOAT,
        max_val  = MAXFLOAT;
static  float *buf[LL_K];         /* binary input token buffer */
static  float *outbuf;            /* both input and output buffer */
static  int   idx = 0;            /* how many items are buffered in outbuf */
static  int   buf_size = SDTM_BUF_SIZE;
                                  /* buffer size of both inbuf and outbuf */
static  int   count;              /* # of elements in the binary token buffer */
static  float *ptr;               /* pointer to the token buffer */
static  first = 1;                /* whether the current output chunk is
                                     the first one */
main(argc, argv)
```

```
int argc;
char **argv;
{
    /* parse the command-line options */
    parse_args(argc, argv, &SDTM_lookahead, &min_val, &max_val, &type,
       &format, &buf_size, &bc_size);
    ANTLR(norm_frame(), stdin);
}
>>

/* literals */
#token FLOAT_CHUNK
#token TYPE_FLOAT
#token TYPE_INT
#token TYPE_STRING
#token FORMAT_ASCII
#token FORMAT_XDR
#token FORMAT_BINARY
#token GRID_TYPE_UNIFORM
#token GRID_TYPE_HYBRID
#token GRID_TYPE_SCATTER
#token GRID_SYSTEM_RECTANGULAR
#token GRID_SYSTEM_CARTESIAN
#token GRID_SYSTEM_POLAR
#token GRID_SYSTEM_SPHERICAL
#token GRID_SYSTEM_CYLINDRICAL
#token GRID_SYSTEM_GEOCENTRIC
#token AXIS_X
#token AXIS_Y
#token AXIS_Z
#token AXIS_REGULAR
#token AXIS_IRREGULAR
#token AXIS_EXPLICIT

/* tokens */
#token                      "[\t\ ]"         <<zzskip();>>
#token                      "\n"             <<zzline++; zzskip();>>
#token INUM[int val]        "{[\+\-]}[0-9]+" <<$val = atoi(zzlextext);>>
#token DIM[int val]         "\[[0-9]+\]"     <<$val = scarfDIM(zzlextext);>>
#token FNUM[float val]
              "{[\+\-]}([1-9][0-9]*{.[0-9]*}|{[0]}.[0-9]+){[Ee]{[\+\-]}[0-9]+}"
                                             <<$val=(float)atof(zzlextext);>>
#token ID[char *lexeme]     "\"[a-zA-Z][a-zA-Z_.0-9]*\""
                                             <<$lexeme = save_lexeme(zzlextext);>>
#token STRING[char *lexeme] "[a-zA-Z][a-zA-Z_0-9]*|[\"][a-zA-Z_0-9\ ]*[\"]"
                                             <<$lexeme = save_lexeme(zzlextext);>>

/* rules */
norm_frame: frame_start (norm_group)+ frame_end "@" <<send_frame_end($3.num, 0);>> ;

frame_start>[int num]: "frame" "start" INUM <<
     send_frame_start($3.val,0);
```

```
     $0.num = $3.val;
    >>;

frame_end>[int num]: "frame" "end" INUM << $0.num = $3.val; >> ;

/* variable group */
norm_group: var_group | grid_group;

var_group:  var_begin var_header var_array var_end <<
                $3.range = $2.range;
                send_end("variable", $4.lexeme);
            >> ;

var_begin>[char *lexeme]:
            "begin" "variable" ID <<
                $0.lexeme = $3.lexeme;
                send_begin("variable", $3.lexeme);
            >> ;

var_header>[Range range, Note notes, char *spatial_ref]:
            var_field var_header <<
              { register int i;
                if ($2.range.count > 0) {
                    $0.range.count = $2.range.count;
                    for (i=0; i<$2.range.count; i++)
                    $0.range.val[i] = $2.range.val[i];
                }
                else if ($1.range.count > 0) {
                    $0.range.count = $1.range.count;
                    for (i=0; i<$1.range.count; i++)
                    $0.range.val[i] = $1.range.val[i];
                }
                else
                    $0.range.count = 0;
                if ($2.notes.count > 0)
                    $0.notes = $2.notes;
                else if ($1.notes.count > 0)
                    $0.notes = $1.notes;
                else
                    $0.notes.count = 0;
                if ($2.spatial_ref != NULL)
                    $0.spatial_ref = $2.spatial_ref;
                else if ($1.spatial_ref != NULL)
                    $0.spatial_ref = $1.spatial_ref;
                else
                    $0.spatial_ref = NULL;
              }
            >>
            | <<
                $0.range.count = 0;
                $0.notes.count = 0;
                $0.spatial_ref = NULL;
```

```
            >> ;

var_field>[Range range, Note notes, char *spatial_ref]:
            var_notes <<
                send_notes($1.notes.count, $1.notes.type,
                           $1.notes.format, $1.notes.str_list);
                $0.notes.count = $1.notes.count;
                $0.notes.type = $1.notes.type;
                $0.notes.format = $1.notes.format;
                $0.notes.str_list = $1.notes.str_list;
                $0.range.count = 0;
                $0.spatial_ref = NULL;
            >>
            | "range" var_range  <<
              { register int i;
                if (min_val != MINFLOAT && max_val != MAXFLOAT) {
                    $0.range.count = 2;
                    $0.range.val[0].type = TYPE_FLOAT;
                    $0.range.val[1].type = TYPE_FLOAT;
                    $0.range.val[0].val.fval = min_val;
                    $0.range.val[1].val.fval = max_val;
                }
                else {
                    $0.range.count = $2.count;
                    for (i=0; i<$2.count; i++)
                        $0.range.val[i] = $2.val[i];
                }
              }
                $0.notes.count = 0;
                $0.spatial_ref = NULL;
            >>
            | "^spatial_ref" "grid:" ID <<
                send_spatial_ref($3.lexeme);
                $0.spatial_ref = $3.lexeme;
                $0.range.count = $0.notes.count = 0;
            >> ;

var_range>[int count, Num val[10]]:
            (var_range_num <<
                if (zzcnt == 1) {
                    $0.count = 1;
                    $0.val[0] = $1.num;
                }
                else {
                    $0.count += 1;
                    $0.val[$0.count-1] = $1.num;
                }
            >>
            )+ > [int count, Num val[10]] <<
              { int i;
                $0.count = $1.count;
                for (i=$1.count-1; i>=0; i--)
```

```
                $0.val[i] = $1.val[i];
          }
        >> ;

var_range_num>[Num num]:
        INUM <<$0.num.type = TYPE_INT; $0.num.val.ival = $1.val;>>
        | FNUM <<$0.num.type = TYPE_FLOAT; $0.num.val.fval = $1.val;>> ;

var_notes>[Note notes]:
        "notes" DIM flux_type flux_format var_n_list <<
            $0.notes.count = $2.val;
            $0.notes.type = $3.type;
            $0.notes.format = $4.format;
            $0.notes.str_list = $5.list;
        >> ;

var_n_list>[int count, Str_list *list]:
        (STRING  <<
          if (zzcnt == 1) {
            Str_list *p;
            $0.list = (Str_list *)calloc(1, sizeof(Str_list));
            require($0.list, "called failed in var_n_list");
            p = (Str_list *)calloc(1, sizeof(Str_list));
            require(p, "called failed in var_n_list");
            p->lexeme = $1.lexeme;
            p->next = NULL;
            $0.list->next = p;
            $0.list->lexeme = (char *)p;
            $0.count = 1;
          }
          else {
            Str_list *p, *last;
            p = (Str_list *)calloc(1, sizeof(Str_list));
            if (p == NULL) {
                fprintf(stderr, "calloc failed\n");
                exit (-1);
            }
            p->lexeme = $1.lexeme;
            p->next = NULL;
            last = (Str_list *)$0.list->lexeme;
            last->next = p;
            $0.list->lexeme = (char *)p;
            $0.count += 1;
          }
        >>
        )+ >[int count, Str_list *list] <<
            $0.count = $1.count;
            $0.list = $1.list;
        >> ;

var_array[Range range]:
        "data" var_dims flux_type flux_format var_data <<
```

```
                $3.dummy = expect_chunk();
                $5.dummy = send_range($3.type,$0.range);
                $5.dummy = send_string("data");
                $5.dummy = send_dims($2.dims);
                $5.dummy = send_type($3.type);
                $5.dummy = send_format(format);
                $5.dummy = fprintf(fp, " \n");
                $5.min = ($0.range.val[0].type == TYPE_INT) ?
                         (float)$0.range.val[0].val.ival :
                         $0.range.val[0].val.fval;
                $5.max = ($0.range.val[1].type == TYPE_INT) ?
                         (float)$0.range.val[1].val.ival :
                         $0.range.val[1].val.fval;
                $5.type = $3.type;
                $5.format = $4.format;
                if (idx > 0) {
                    var_normalize(outbuf, idx, $5.min, $5.max);
                    idx = 0;
                }
            >> ;

var_dims>[Dim dims]:
            (DIM <<
                if (zzcnt == 1) {
                    $0.dims.ndim = 1;
                    $0.dims.count = $1.val;
                    $0.dims.dims[0] = $1.val;
                }
                else {
                    $0.dims.ndim += 1;
                    $0.dims.count *= $1.val;
                    $0.dims.dims[$0.dims.ndim-1] = $1.val;
                }
            >>
            )+ >[Dim dims] << $0.dims = $1.dims; >> ;

var_data[float min, float max, int type, int format]:
            (
                FLOAT_CHUNK <<
                    sdtm_decode_ptr(&count, (char **)&ptr,
                           (unsigned char *)zzlextext);
                    NORM_BIN(ptr, count, $$0.min, $$0.max);
                >>
                | INUM << NORM((float)$1.val, $$0.min, $$0.max); >>
                | FNUM << NORM($1.val, $$0.min, $$0.max); >>
            )+ ;

var_num>[float val]: INUM <<$0.val = (float)$1.val;>> | FNUM <<$0.val = $1.val;>> ;

var_end>[char *lexeme]: "end" "variable" ID <<$0.lexeme = $3.lexeme;>> ;

/* grid group */
```

```
grid_group >[grid_gp grid]:
            grid_begin
            (grid_field)+ >[int type, int system, int dim]
            (axis_group)+
            grid_end << send_end("grid", $4.lexeme); >> ;

grid_begin>[char *lexeme]:
            "begin" "grid" ID <<
                $0.lexeme = $3.lexeme;
                send_begin("grid", $3.lexeme);
            >>;

grid_field>[int type, int system, int dim]:
            flux_notes
            | "type" grid_type <<send_grid_type($2.type); $0.type = $2.type;>>
            | "system" grid_system <<send_grid_system($2.system);>>
            | "dimensions" INUM <<fprintf(fp,"\tdimensions %d\n",$2.val);>> ;

grid_type>[int type]:
            "uniform"  <<$0.type = GRID_TYPE_UNIFORM;>>
            | "hybrid"  <<$0.type = GRID_TYPE_HYBRID;>>
            | "scatter" <<$0.type = GRID_TYPE_SCATTER;>> ;

grid_system>[int system]:
            "rectangular"  <<$0.system = GRID_SYSTEM_RECTANGULAR;>>
            | "cartesian"  <<$0.system = GRID_SYSTEM_CARTESIAN;>>
            | "polar"  <<$0.system = GRID_SYSTEM_POLAR;>>
            | "spherical" <<$0.system = GRID_SYSTEM_SPHERICAL;>>
            | "cylindrical" <<$0.system = GRID_SYSTEM_CYLINDRICAL;>>
            | "geocentric"  <<$0.system = GRID_SYSTEM_GEOCENTRIC;>> ;

grid_end >[char *lexeme]:
            "end" "grid" ID <<$0.lexeme = $3.lexeme;>> ;

/* Axis group */
axis_group: axis_begin (axis_field)+ axis_end <<
                send_end("axis", $3.axis==AXIS_X?"\"X\"":
                                ($3.axis==AXIS_Y?"\"Y\"":"\"Z\""));
            >> ;

axis_begin >[int axis]:
            "begin" "axis" axis_name <<
                send_begin("axis", $3.axis==AXIS_X?"\"X\"":
                                 ($3.axis==AXIS_Y?"\"Y\"":"\"Z\""));
                $0.axis = $3.axis;
            >> ;

axis_name >[int axis]:
            "\"X\"" <<$0.axis = AXIS_X;>>
            | "\"Y\"" <<$0.axis = AXIS_Y;>>
            | "\"Z\"" <<$0.axis = AXIS_Z;>> ;
```

```
axis_end >[int axis]:
            "end" "axis" axis_name << $0.axis = $3.axis; >> ;

axis_field: axis_spacing | axis_size | axis_origin | axis_length | axis_position ;

axis_spacing>[int spacing]:
            "spacing" spacing << $0.spacing = $2.spacing; >> ;

spacing>[int spacing]:
            "regular" <<
                fprintf(fp, "\tspacing regular\n");
                $0.spacing = AXIS_REGULAR;
            >>
            | "irregular" <<
                fprintf(fp, "\tspacing irregular\n");
                $0.spacing = AXIS_IRREGULAR;
            >>
            | "explicit" <<
                fprintf(fp, "\tspacing explicit\n");
                $0.spacing = AXIS_EXPLICIT;
            >> ;

axis_size>[int size]:
            "size" INUM <<fprintf(fp, "\tsize %d\n", $2.val); $0.size = $2.val;>>;

axis_origin>[float origin]:
            "origin" flux_fnum <<
                fprintf(fp,"\torigin %f\n",$2.val);
                $0.origin = $2.val;
            >>;

axis_length>[float length]:
            "length" flux_fnum <<
                fprintf(fp,"\tlength %f\n",$2.val);
                $0.length = $2.val;
            >>;

axis_position: "position" DIM flux_type flux_format position_data <<
                printf("WARNING: position not implemented\n"); >> ;

position_data: (flux_fnum)+;

flux_fnum>[float val]:
            INUM  <<$0.val = (float)$1.val;>>
            | FNUM <<$0.val = $1.val;>> ;

flux_type>[int type]:
            "int"   <<$0.type = TYPE_INT;>>
            | "float" <<$0.type = TYPE_FLOAT;>>
            | "double" <<$0.type = TYPE_FLOAT;>>
            | "string" <<$0.type = TYPE_STRING;>> ;
```

```
flux_format>[int format]:
            "ascii" <<$0.format = FORMAT_ASCII;>>
            | "xdr"   <<$0.format = FORMAT_XDR;>>
            | "binary" <<$0.format = FORMAT_BINARY;>>
            | "ieee" <<$0.format = FORMAT_BINARY;>> ;

flux_notes: "notes" DIM "string" flux_format
            (STRING <<send_string($1.lexeme); send_string("\n");>>)+ <<
                $5.dummy = send_string("notes");
                $5.dummy = send_dim($2.val);
                $5.dummy = send_string(" string");
                $5.dummy = send_format($4.format);
                $5.dummy = send_string("\n\t");
            >>;

any_num>[float val]: INUM <<$0.val = (float)$1.val;>> | FNUM <<$0.val = $1.val;>> ;


/*
 * Auxiliary C routines.
 */
#include <stdio.h>
#include <math.h>
#include <stdio.h>
#include <sys/types.h>
#include <malloc.h>
#include <string.h>
#include "var.h"

extern float min_val, max_val;
extern type;
extern format;
FILE *fp = NULL;
char *out_file = "norm.output";
static int xyz;
static int first_chunk;
static char tstring[32];

expect_chunk()
{
    int i;
    if (strcmp(LATEXT(2),"ascii") != 0) {     /* not ascii */
        zzchunk = 1;
        zzchunksize = bc_size;
        for (i=0; i<LL_K; i++) {
            buf[i] = PfmMem(float, zzchunksize);
            zzarray[i] = (char *)buf[i];
        }
    }
    else
        bc_size = 1;
    outbuf = PfmMem(float, buf_size+SDTM_lookahead*bc_size);
```

```
}

pass_range(r1, r2)
Range   r1, r2;
{
    register i;
    r2.count = r1.count;
    for (i=0; i<r1.count; i++)
        r2.val[i] = r1.val[i];
}

send_range(type, range)
int     type;
Range range;
{
    switch (type) {
        case TYPE_INT:
            if (range.val[0].type == TYPE_INT)
                fprintf(fp, "range %d", range.val[0].val.ival);
            else if (range.val[0].type == TYPE_FLOAT)
                fprintf(fp, "range %d", (int)range.val[0].val.fval);
            if (range.val[1].type == TYPE_INT)
                fprintf(fp, " %d\n", range.val[1].val.ival);
            else if (range.val[1].type == TYPE_FLOAT)
                fprintf(fp, " %d\n", (int)range.val[1].val.fval);
            break;
        case TYPE_FLOAT:
            if (range.val[0].type == TYPE_INT)
                fprintf(fp, "range %f", (float)range.val[0].val.ival);
            else if (range.val[0].type == TYPE_FLOAT)
                fprintf(fp, "range %f", range.val[0].val.fval);
            if (range.val[1].type == TYPE_INT)
                fprintf(fp, " %f\n", (float)range.val[1].val.ival);
            else if (range.val[1].type == TYPE_FLOAT)
                fprintf(fp, " %f\n", range.val[1].val.fval);
            break;
        default:
            fprintf(stderr, "unknown type\n");
            break;
    }
}

send_dim(d) int d; { fprintf(fp, "[%d]", d); }

send_dims(dims)
Dim dims;
{
    int i;
    xyz = 1;
    for (i=0; i<dims.ndim; i++) {
        fprintf(fp, "[%d]", dims.dims[i]);
        xyz *= dims.dims[i];
```

```
    }
    first_chunk = 1;
}

send_string(text) char  *text; { fprintf(fp, "%s", text); }

send_int(i) int   i; { fprintf(fp, "\t%d\n", i); }

send_float(f) float f; { fprintf(fp, "\t%f\n", f); }

send_notes(count, type, format, list)
int count;
int type;
int format;
Str_list *list;
{
    Str_list *p;
    fprintf(fp, "notes[%d]", count);
    send_type(type);
    send_format(format);
    fprintf(fp, "\n");
    p = list->next;
    while (p) {
        fprintf(fp, "\t%s\n", p->lexeme);
        p = p->next;
    }
}

send_spatial_ref(str) char  *str; {fprintf(fp,"^spatial_ref grid:%s\n",str);}

send_type(type)
int     type;
{
    switch (type) {
        case TYPE_INT:
            fprintf(fp, " int");
            break;
        case TYPE_FLOAT:
            fprintf(fp, " float");
            break;
        case TYPE_STRING:
            fprintf(fp, " string");
            break;
        default:
            break;
    }
}

send_format(format)
int     format;
{
    switch (format) {
```

```
        case FORMAT_ASCII:
            fprintf(fp, " ascii");
            break;
        case FORMAT_BINARY:
            fprintf(fp, " ieee");
            break;
        case FORMAT_XDR:
            fprintf(fp, " xdr");
            break;
        default:
            break;
    }
}

var_normalize(buf, num, min, max)
float *buf;
int   num;
float min, max;
{
    switch (type) {
        case TYPE_LINEAR:
            do_linear(buf, num, min, max);
            break;
        case TYPE_LOG:
            do_log(buf, num, min, max);
            break;
        case TYPE_EXP:
            do_exp(buf, num, min, max);
            break;
        default:
            break;
    }
    switch (format) {
        case FORMAT_ASCII:
            output_float_ascii(buf, num);
            break;
        case FORMAT_BINARY:
            output_float_binary(buf, num);
            break;
        case FORMAT_XDR:
            fprintf(stderr, "xdr not implemented\n");
            break;
        default:
            break;
    }
}

output_float_ascii(buf, num)
float   *buf;
int             num;
{
    int i;
```

```
    for (i=0; i<num; i++)
        fprintf(fp, "\t%.10g\n", buf[i]);
    fflush(fp);
}

output_float_binary(buf, num)
float   *buf;
int             num;
{
    if (first_chunk) {
        wt_xdr_long(fp, xyz*sizeof(float));
        first_chunk = 0;
    }
    fwrite(buf, sizeof(float), num, fp);
    fflush(fp);
}

do_linear(data, count, min, max)
float   *data;
int             count;
float   min, max;
{
    register int i;
    register float val;
    for (i=0; i<count; i++) {
        val = min+(data[i]-min)/(max-min);
        data[i] = ZeroOneClamp(val);
    }
}

do_log(data, count, min, max)
float          *data;
int            count;
float                   min, max;
{
    static double       l_lo, l_hi, lo, hi;
    static double       l_range, hi_lo;
    static short        first = 1;
    double              range;
    register double             f, val;
    register int        i;
    if (first) {
        lo = exp(-2.0);
        hi = exp(2.0);
        l_lo = log(lo);
        l_hi = log(hi);
        l_range = l_hi - l_lo;
        hi_lo = hi - lo;
        first = 0;
    }
    range = max - min;
    if (range != 0.0 && l_range != 0.0) {
```

```
        for (i = 0; i < count; i++) {
            val = (hi_lo*(data[i] - min)/range) + lo;
            f = (val <= 0.0) ? 0.0 : log(val);
            val = (f - l_lo)/l_range;
            data[i] = ZeroOneClamp(val);
        }
    }
}

do_exp(data, count, min, max)
float   *data;
int     count;
float   min, max;
{
    static double       e_lo, e_hi, lo = -2.0, hi = 2.0;
    static double       e_range, hi_lo;
    static short        first = 1;
    double                              range;
    register double     f, val;
    register int        i;
    if (first) {
        e_lo = exp(lo);
        e_hi = exp(hi);
        e_range = e_hi - e_lo;
        hi_lo = hi - lo;

        first = 0;
    }
    range = max - min;
    if (range != 0.0 && e_range != 0.0) {
        for (i = 0; i < count; i++) {
            f = exp((hi_lo*(data[i] - min)/range) + lo);
            val = (f - e_lo)/e_range;
            data[i] = ZeroOneClamp(val);
        }
    }
}

send_grid_type(type)
int     type;
{
    switch (type) {
        case GRID_TYPE_UNIFORM:
            fprintf(fp, "\ttype uniform\n");
            break;
        case GRID_TYPE_HYBRID:
            fprintf(fp, "\ttype hybrid\n");
            break;
        case GRID_TYPE_SCATTER:
            fprintf(fp, "\ttype scatter\n");
            break;
        default:
```

```
            break;
    }
}

send_grid_system(system)
int     system;
{
    switch (system) {
        case GRID_SYSTEM_RECTANGULAR:
            fprintf(fp, "\tsystem rectangular\n");
            break;
        case GRID_SYSTEM_CARTESIAN:
            fprintf(fp, "\tsystem cartesian\n");
            break;
        case GRID_SYSTEM_POLAR:
            fprintf(fp, "\tsystem polar\n");
            break;
        case GRID_SYSTEM_SPHERICAL:
            fprintf(fp, "\tsystem spherical\n");
            break;
        case GRID_SYSTEM_CYLINDRICAL:
            fprintf(fp, "\tsystem cylindrical\n");
            break;
        case GRID_SYSTEM_GEOCENTRIC:
            fprintf(fp, "\tsystem geocentric\n");
            break;
        default:
            break;
    }
}

scarfDIM(text)
char *text;
{
    int ret;
    char *p, c;
    p = text;
    while (*p != '\0') p++;
    p--;
    c = *p;
    *p = '\0';
    ret = atoi(text+1);
    *p = c;
    return ret;
}

char  *
save_lexeme(text)
char    *text;
{
    char    *ret;
    ret = (char *)calloc(1, strlen(text)+1,sizeof(char));
```

```
    if (ret == NULL) {
        fprintf(stderr, "calloc(%d,1) failed in save_lexeme.\n",
                        strlen(text)+1);
        exit (-1);
    }
    strcpy(ret, text);
    return (ret);
}

send_frame_start(num, nosend)
int     num;
int nosend;             /* do not send data to standard out */
{
    if (fp == NULL) {
        fp = stdout;
    }
    if (!nosend)
        fprintf(fp, "frame start %d\n", num);
}

send_frame_end(num, nosend)
int     num;
int     nosend;
{
    if (!nosend)
        fprintf(fp, "frame end %d\n", num);
}

send_begin(group, name)
char    *group, *name;
{
    if (fp == NULL) {
        fp = stdout;
    }
    fprintf(fp, "begin %s %s\n", group, name);
}

send_end(group,name) char *group,*name; {fprintf(fp,"end %s %s\n",group,name);}

/*
 * Include file
 */
#define TYPE_LINEAR 0
#define TYPE_LOG 1
#define TYPE_EXP 2
#define require(expr, err) if ( !(expr) ) {             \
            fprintf(stderr, "%s (file %s, line %d).\n", \
                    (err),__FILE__, __LINE__);  \
            exit (-1);  \
}
#endif
#define NORM(v,min,max) {  \
```

```
        outbuf[idx++] = (v);                          \
        if (first&&idx==buf_size+SDTM_lookahead || !first&&idx==buf_size) {\
            var_normalize(outbuf, first?(buf_size+SDTM_lookahead):buf_size, \
  (min),(max));\
            idx = 0;                                    \
            first = 0;                                  \
        }                                                  \
}
#define NORM_BIN(p,c,min,max) {                            \
       int i;                                              \
       for (i=0; i<(c); i++) {                          \
           outbuf[idx++] = p[i];                       \
           if (first&&idx==buf_size+SDTM_lookahead*bc_size || !first&&idx==buf_size) {\
               var_normalize(outbuf, first?(buf_size+SDTM_lookahead*bc_size):buf_size,\
                            (min),(max));                  \
               idx = 0;                                \
               first = 0;                              \
           }                                           \
       }                                               \
}
typedef struct _Num {
    int type;
    union {
        int  ival;
        float  fval;
    } val;
} Num;
typedef struct _Range {
    int count;
    Num val[10];
} Range;
typedef struct _Dim {
    int ndim; /* number of dimensions */
    int count; /* total number of elements */
    int dims[10]; /* each individual dimension size */
} Dim;
typedef struct _Str_list {
    char *lexeme;
    struct _Str_list *next;
} Str_list;
typedef struct _Note {
    int count;
    int type;
    int format;
    Str_list *str_list;
} Note;
```

Bibliography

[1] W. B. Ackerman and J. B. Dennis. VAL – a value-oriented algorithmic language: Preliminary reference manual. TR 218, Laboratory for Computer Science, MIT, July 1979.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Pub. Co., Reading, Mass., 1986.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS 1967 Spring Joint Computer Conf.*, pages 483–485, Atlantic City, NJ, April 30, 1967.

[4] H. Stephen Anderson, et al. The animation production environment: A basis for visualization and animation of scientific data. Technical report, Ohio Supercomputer Graphics Project, May 1989.

[5] H. Stephen Anderson, et al. Attacking unsolved problems in visualization of scientific data: A panel presentation. In *Extracting Meaning from Complex Data: Processing, Display, Interaction*, volume 1259, pages 308–314. SPIE, February 14–16, 1990.

[6] Robert G. Babb II. Parallel processing with large grain data flow techniques. *IEEE Computer*, 17(7):55–61, July 1984.

[7] William A. Barrett, et al. *Compiler Construction: Theory and Practice*. Science Research Associates, Inc., Chicago, second edition edition, 1986.

[8] Gregor V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19(2):55–62, February 1976.

[9] Gregor V. Bochmann and P. Ward. Compiler writing system for attribute grammars. *The Computer Journal*, 21(2):144–148, May 1978.

[10] M. Chmilar and B. Wyvill. A software architecture for integrated modeling and animation. In *New Advances in Computer Graphics (Proceedings of CG International '89)*, pages 257–276, 1989.

[11] Jarrett Cohen and Sara Latta. Simulating reality on the metacomputer. *Access*, 5(5):10–13, September-December 1991.

[12] CONVEX Computer Corporation. *Using ConvexAVS to Visualize Data*, first edition, February 1992.

[13] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963.

[14] B. Courcelle. Attribute grammars: Definitions, analysis of dependencies, proof methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction: An Advanced Course*, pages 81–102. Cambridge University Press, Cambridge, England, 1984.

[15] Jack B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

[16] Jack B. Dennis. Models of data flow computation. In Manfred Broy, editor, *Control Flow and Dataflow: Concepts of Distributed Programming*, pages 346–398. 1984.

[17] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1988.

[18] D. Scott Dyer. apE: Providing visualization tools for a statewide supercomputing network. In *Proceedings of the 24th Semi-Annual Cray Users Group Meeting*, pages 237–241, Trondheim, Norway, Sept. 18–22, 1989.

[19] D. Scott Dyer. A dataflow toolkit for visualization. *IEEE Computer Graphics and Applications*, 10(4):60–69, July 1990.

[20] E.F. Elsworth and M.A.B. Parkes. Automated compiler construction based on top-down syntax analysis and attribute evaluation. *SIGPLAN Notice*, 25(8):37–42, August 1990.

[21] Joost Engelfriet. Attribute grammars: Attribute evaluation methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction: An Advanced Course*, pages 103–138. Cambridge University Press, Cambridge, England, 1984.

[22] Joost Engelfriet and Gilberto Filè. Passes, sweeps, and visits in attribute grammars. *Journal of the ACM*, 36(4):841–869, October 1989.

[23] R. Farrow and D. Yellin. A comparison of storage optimizations in automatically-generated attribute evaluators. *Acta Informatica*, 23(4):393–427, 1986.

[24] Jeffrey T. Faust and D. Scott Dyer. An effective data format for scientific visualization. volume 1259, pages 254–260. SPIE, February 1990.

[25] D.D. Gajski, et al. A second opinion on data flow machines and languages. *IEEE Computer*, 15(2):58–68, February 1982.

[26] Guang R. Gao. Exploiting fine-grain parallelism on dataflow architectures. *Parallel Computing*, 13(3):309–320, March 1990.

[27] Guang R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Norwell, Mass., 1991.

[28] Al Globus. Perspectives on the IRIS Explorer visualization environment. Technical Report RNR-92-021, NASA Ames Research Center, NAS Systems Division, Applied Research Branch, May 1992.

[29] V. Bancroft Gordon, et al. FAST: A multi-processed environment for visualization of computational fluid dynamics. In *Visualization '90 : Proceedings of the First IEEE Conference on Visualization*, pages 14–27, San Francisco, California, October 23–26, 1990.

[30] Robert W. Gray, et al. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[31] Robert B. Haber. Scientific visualization and the RIVERS project at the national center for supercomputing applications. *IEEE Computer*, 22(8):84–89, August 1989.

[32] Robert B. Haber, David A. McNabb, and Robert A. Ellis. Eliminating distance in scientific computing: An experiment in televisualization. *The International Journal of Supercomputing Applications*, 4(4):71–89, winter 1990.

[33] Paul E. Haeberli. ConMan: A visual programming language for interactive graphics. *Computer Graphics*, 22(4):103–111, August 1988.

[34] Jayantha Herath, et al. Dataflow computing models, languages, and machines for intelligence computations. *IEEE Transaction on Software Engineering*, 14(12):1805–1828, December 1989.

[35] Daniel D. Hils. Data flow visual programming languages. *Journal of Visual Languages and Computing*, 2(1):1–32, 1991.

[36] Daniel D. Hils. DataVis: A visual programming language for scientific visualization. In *Proceedings of 1991 ACM Computer Science Conference*, pages 439–448, March 1991.

[37] Daniel D. Hils. *A Visual Programming Language for Visualization of Scientific Data.* PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1992.

[38] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[39] Roger Hoover. *Incremental Graph Evaluation.* PhD thesis, Dept. of Computer Science, Cornell Univ., May 1987.

[40] Scott E. Hudson. *A User Interface Management System Which Supports Direct Manipulation.* PhD thesis, Dept. of Computer Science, Univ. of Colorado, 1986.

[41] Scott E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, July 1991.

[42] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.

[43] IBM. *IBM AIX Visualization Data Explorer/6000: Programmer's Reference*, v1.1 edition, November 1991.

[44] IBM. *IBM AIX Visualization Data Explorer/6000: User's Guide*, v1.1 edition, November 1991.

[45] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, Bell Laboratories, Murray Hill, NJ, 1978.

[46] Larry G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462, July 1990.

[47] Martin Jourdan, et al. Design, implementation and evaluation of the FNC-2 attribute grammar system. *SIGPLAN Notice*, 25(6):209–222, June 1990.

[48] Gail E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Transactions on Programming Languages and Systems*, 11(2):169–193, April 1989.

[49] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, 1980.

[50] Uwe Kastens. The GAG-system – a tool for compiler construction. In B. Lorho, editor, *Methods and Tools for Compiler Construction: An Advanced Course*, pages 165 – 181. Cambridge University Press, Cambridge, England, 1984.

[51] Uwe Kastens. Lifetime analysis for attributes. *Acta Informatica*, 24(6):633–651, 1987.

[52] Ken Kennedy and Scott K. Warren. Automatic generation of efficient evaluators. In *Conference Record of the* 3^{rd} *ACM Symposium on Principles of Programming Languages*, pages 32–49, 1976.

[53] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[54] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Pub. Co., Reading, Mass., 1973.

[55] Peter M. Kogge. *The architecture of pipelined computers*. Hemisphere Pub. Corp., Washington, 1981.

[56] Steve P Landry. *System Oriented Extensions to Dataflow.* PhD thesis, Dept. of Computer Science, Univ. of Southwestern Louisiana, Lafayette, May 1981.

[57] Clement Kin Cho Leung. Formal properties of well-formed data flow schemas. Mac Technical Memorandum 66, MIT, June 1975.

[58] Michael Levison and Gregory Lessard. Application of attribute grammars to natural language sentence generation. In P. Deransart and M. Jourdan, editors, *Lecture Notes in Computer Science: Attribute Grammars and Their Applications*, volume 461, pages 298–312. Springer-Verlag, 1990.

[59] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37., May 1988.

[60] P.M. Lewis, D.J. Rosenkrantz, and R.E. Stearns. Attributed translations. *J. Computer and System Sciences*, 9(3):279–307, 1974.

[61] William E. Lorensen and Harvey E. Cline. A high resolution 3D surface construction algorithm. In *SIGGRAPH'87 Conference Proceedings*, pages 163–170, July 27-31, 1987.

[62] Harold Lorin. *Aspects of Distributed Computer Systems.* John Wiley & Sons, New York, 1988.

[63] Michael Marcotty and Henry F. Ledgard. *Programming Language Landscape: Syntax, Semantics, and Implementation.* Science Research Associates, Chicago, IL., 1986.

[64] Robert Marshall, et al. Visualization methods and simulation steering for a 3D turbulence model of Lake Erie. *Computer Graphics*, 24(2):89–97, March 1990.

[65] Bruce H. McCormick, Thomas A. DeFanti, and Maxine D. Brown. Visualization in scientific computing. *Computer Graphics*, 21(6), November 1987.

[66] James McGraw, et al. SISAL: Streams and iterations in a single-assignment language, language reference manual, version 1.1. Technical Report M–146, Lawrence Livermore National Laboratory, July 1983.

[67] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1980.

[68] Tom Nadas and Alain Fournier. GRAPE: An environment to build display processes. *Computer Graphics*, 21(4):75–84, July 1987.

[69] David Nagel, et al. Interface and new interactive systems. In *SIGGRAPH'90 Panel Proceedings*, pages 2–9, 1990.

[70] National Center for Supercomputing Applications. *Viewit v9.10: User's Guide*, December 1990.

[71] Michael Norman. personal communication, February 11, 1992.

[72] The Ohio Supercomputer Graphics Project. *apE Version2.0: Programmer's Manual*, November 1990.

[73] Rieks op den Akker, Bořivoj Melichar, and Jorma Tarhio. Attribute evaluation and parsing. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 187–214. Springer-Verlag, Berlin, 1991.

[74] H. G. Pagendarm and S. I. Choudhry. Visualization of hypersonic flows - exploring the opportunities to extend AVS. In *Workshop Papers of the Fourth Eurographics Workshop on Visualization in Scientific Computing*, Abingdon, United Kingdom, April 21-23, 1993.

[75] T. J. Parr, H. G. Dietz, and W. E. Cohen. *PCCTS Reference Manual.* School of Electrical Engineering, Purdue University, version 1.00 edition, August 1991.

[76] Terence Parr and Henry Dietz. A practical approach to $LL(k)$: $LL_m(n)$. Technical Report TR-EE 92-30, School Of Electrical Engineering, Purdue University, July 1992.

[77] Terence Parr, Henry Dietz, and Will Cohen. Purdue compiler-construction tool set. Technical Report TR-EE 90-14, School Of Electrical Engineering, Purdue University, February 1990.

[78] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[79] James L. Peterson. Petri nets. *ACM Computing Survey*, 9(3):223–252, September 1977.

[80] Michael Potmesil and Eric M. Hoffert. FRAMES: Software tools for modeling rendering and animation of 3d scenes. *Computer Graphics*, 21(4):85–93, July 1987.

[81] John Rasure, et al. Visual language and software development environment for image processing. *International Journal of Imaging Systems and Technology*, 2:183–199, 1990.

[82] Thomas W. Reps. *Generating Language Based Environments.* The MIT Press, Cambridge, Mass., 1984.

[83] William Ribarsky, et al. Object-oriented, dataflow visualization systems – a paradigm shift? In *Proceedings of Visualization '92*, pages 384–388, Boston, Massachusetts, October 19-23, 1992.

[84] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Wm. C. Brown Publishers, Dubuque, Iowa, 1986.

[85] John A. Sharp. *Data Flow Computing.* Ellis Horwood Limited, Chichester, England, 1985.

[86] Silicon Graphics, Inc. *IRIS Explorer Module Writer's Guide*, β draft edition, 1991.

[87] Silicon Graphics, Inc. *IRIS Explorer User's Guide*, β draft edition, 1991.

[88] Donald R. Slutz. The flow graph schemata model of parallel computation. Technical Report MAC-TR-53, MIT, September 1968.

[89] Deyang Song and Eric Golin. Fine-grain visualization algorithms in dataflow environments. In *Proceedings of Visualization '93*, San Jose, California, October 25-29, 1993. To be published.

[90] Deyang Song, Eric Golin, and Michael Norman. A fine-grain dataflow model for scientific visualization systems. In *Workshop Papers of the Fourth Eurographics Workshop on Visualization in Scientific Computing*, Abingdon, United Kingdom, April 21-23, 1993.

[91] Guy Lewis Steele Jr. and Gerald Jay Sussman. Scheme: An interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory, 1975.

[92] Paul Suhler, et al. TDFL: A task-level dataflow language. *Journal of Parallel and Distributed Computing*, 9:103–115, 1990.

[93] P. Suignard. Application builders: Architecture and internal operation. In *Workshop Papers of the Fourth Eurographics Workshop on Visualization in Scientific Computing*, Abingdon, United Kingdom, April 21-23, 1993.

[94] Lloyd A. Treinish. The role of data management in discipline-independent data visualization. In *Extracting Meaning from Complex Data: Processing, Display, Interaction*, volume 1259 of *SPIE*, pages 261–269. 1990.

[95] Lloyd A. Treinish. Siggraph'90 workshop report: Data structures and access software for scientific visualization. *Computer Graphics*, 25(2):104–118, April 1991.

[96] Lloyd A. Treinish. personal communication, June 15, 1993.

[97] Lloyd A. Treinish, et al. Effective software systems for scientific data visualization. In *SIGGRAPH'89 Panel Proceedings*, pages 111–136, 1989.

[98] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins. Data-driven and demand-driven computer architecture. *Computing Surveys*, 14(1):93–143, March 1982.

[99] Kaoru Uchida and Tsutomu Temma. A pipelined dataflow processor architecture based on a variable length token concept. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 209–216, August 1988.

[100] Craig Upson, et al. The Application Visualization System: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.

[101] Craig Upson, et al. Future directions of visualization software environments. In *SIGGRAPH'91 Panel Proceedings*, 1991.

[102] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.

[103] William M. Waite and Gerhard Goos. *Compiler Construction.* Springer-Verlag, New York, 1984.

[104] Douglas Walsten and David Garver. apE evaluation report. System development group report, National Center for Supercomputing Applications, February 27, 1991.

[105] Wavefront Technologies, Inc. *The Data Visualizer: User's Guide*, December 1990.

[106] R. Wilhelm, et al. Design evaluation of the compiler generating system MUG1. In *2nd International Conference on Software Engineering*, pages 308–320, San Francisco, Ca., October 1976.

[107] Maurice V. Wilkes. The long-term future of operating systems. *Communications of the ACM*, 35(11):23–24, 112, November 1992.

[108] Peter L. Williams. *Interactive Direct Volume Rendering of Curvilinear and Unstructured Data*. PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1992.

[109] Mark Young and Carla Williams. *Cantata: Visual Language Programming Environment*. The Khoros Group, Department of Electrical and Computer Engineering, University of New Mexico, release 1.0 edition, September 30, 1991. This is Chapter 2 of *Khoros Manual Volume 1: User's Manual*.

Vita

Deyang Song, also known as Chris Song, was born on June 23, 1964, in Shanghai, China. He graduated with a B.S. degree in Computer Science from Fudan University, Shanghai, China, in July, 1985. He received a M.S. degreee in Computer Science from the University of Lowell, Massachusetts, in May, 1988.