PROGRESS IN SIMULATION

VOLUME 1

edited by

George W. Zobrist

James V. Leonard

PROGRESS IN SIMULATION

VOLUME I

edited by

George W. Zobrist
Department of Computer Science
University of Missouri-Rolla
Rolla, MO

James V. Leonard
P.O. Box 1075
Florissant, MO

ABLEX PUBLISHING CORPORATION
NORWOOD, NEW JERSEY

Printed in the United States of America

Library of Congress Cataloging-in-Publication Data

Progress in simulation / edited by George W. Zobrist, James V. Leonard.
p. cm.
Includes bibliographical references and index.
ISBN 0-89391-652-8
1. Computer simulation. I. Zobrist, George W. (George Winston), 1934- . II. Leonard, James V.
QA76.9.C65P77 1992
003'.3—dc20 91-44710
CIP

Ablex Publishing Corporation
355 Chestnut Street
Norwood, New Jersey 07648

To Jessica, Tori, Roy, and Ryan
they make life worthwhile

Contents

1

Analysis of the Structure of Simulation Software Systems

Sorin Davidovici

O'Neill Communications, Inc.
Princeton, NJ

Paul W. Lee

Bell Laboratories
Holmdel, NJ

1 INTRODUCTION

Multiprocessor systems represent an approach used to obtain the computational speed of a very fast computer by using several slower computers operating in parallel. No optimum methods exist by which N computers or processors can be made to operate together on a single problem in order to achieve speed increases on the order of N. It is conceivable to put together several computers or processors. The fundamental questions lie in how to interconnect them and program them in order to obtain fast, reliable, and economical computation.

The multiprocessor represents a very flexible computer architecture. But flexibility does not itself guarantee high efficiency. There are still many open research problems concerning how to organize parallel computations on a multiprocessor system so as to make the best use of N cooperating processors in the solution of a single problem. Here we investigate a software analysis and design methodology for the case where the multiprocessor is the targeted execution environment. Ultimately we address the issue of scheduling computations such that the computer system resources are utilized efficiently. The software design

and analysis approaches presented in this chapter are arrived at through elementary graph theoretical approaches applied to data flow representations of simulation software systems. The usefulness of the graph theoretical analysis approach as applied to the data flow representation of a software system lies not only in greatly simplifying the derivation of the algorithms in this chapter, but also in clarifying the philosophy which underlies the software design and analysis tools developed here.

The chapter starts with a review of basic concepts and terminology followed by a short review of graph theoretical concepts as they pertain to this subject matter. This is followed by a description of the characteristics of simulation software which justify this approach.

2 BASIC CONCEPTS

Computer simulation is a widely used tool in all engineering and scientific fields. Starting from simple, well understood interactions, the output of a simulation describes the behavior of systems far too complex to quantize using purely analytical techniques. Thus the larger and more complex the contemplated system, the more important the simulation process to the design and evaluation process. The simulation of large systems can tax the resources of even the largest computers. In many applications it is desired (and cost effective) to reduce the simulation time. One available alternative is to use a multiprocessor which, if executing the proper software, could provide supercomputer performance at a fraction of the cost.

Multiprocessors fall in the class of MIMD (multiple instructions, multiple data) machines. This class of machines tend to be extremely effective provided that the software exhibits a *sufficient* degree of concurrency and that this concurrency is properly exploited. The meaning of sufficient above depends primarily on the number of processors which will concurrently execute the simulation. The improvement in the simulation run time obtained by utilizing more processors is referred to as *speedup*. As long as enough inherent concurrency exists in the simulation software, adding an additional processor to the resource pool available to the simulation will decrease the execution time (or speedup the processing). The existence of a degree of concurrency in the simulation software which is sufficient to efficiently support N processors will produce a *linear speedup* as the N processors are introduced one at a time. Thus, the addition of each additional processor reduces the simulation execution time to a fraction approximately equal to $1/M$ of the original simulation execution time, where M is the number of processors utilized. Utilizing more than some number of processors, N, which can be supported by the available degree of concurrency in the simulation software will provide diminishing returns. Thus, before either, choosing a multiprocessor to run some specific simulation software, or during the design of

simulation software, it is useful to ascertain the degree of concurrency inherently present in the software structure.

The simulation software analysis can be approached using data-flow computing concepts. The concept of data-flow computing is fundamentally different from the conventional Von Neumann sequential control computing. Data flow computing is a data-driven computation mode in which the instruction execution is driven by data availability. Data-driven execution of any software system (including simulation software) inherently exhibits the degree of concurrency inherent in the software. The degree to which this inherent concurrency exists in the specific software will determine the ultimate limitations of the system's throughput regardless of its complexity. This chapter regards the simulation software to be executed as the software implementation of an *algorithm*. The algorithm itself describes the specific way that the simulated system changes its state as a result of external stimuli. Many algorithms used to simulate systems share certain common traits. This commonality of traits enables the building of a common approach which permits the evaluation of the degree of concurrency present in a large class of algorithms.

Simulation software is highly iterative; the system which is simulated is moved from its initial state to a different state during each iteration (or each execution of the simulation algorithm). Associated with the execution of each successive evaluation of the state of the simulated system is the state of the multiprocessor executing the simulation. A complete execution of the algorithm, or one iteration of the simulation, brings the multiprocessor back to its initial state, ready to begin the next iteration. Thus, the execution of all the tasks until the multiprocessor returns to its initial state is referred to as one *basic cycle* of the simulation. The execution of one complete iteration of the simulation usually involves the execution of a number of different tasks (where each task may be, for example, a subroutine). A buffer may be necessary to store data passed from one task to another. An algorithm is derived which is used to obtain these intertask buffering requirements as well as the number of executions of each task during one basic cycle of the simulation. The algorithm uses a graph theoretical approach and, when analyzed, it is shown to exhibit *linear complexity.* Following the determination of a basic cycle (i.e., the determination of the number of repetitions of each task within a basic cycle) a second algorithm is derived which generates the precedence relationships among these tasks within *one* or *more* basic cycles. The number of basic cycles over which these task precedence relationships are determined constitute a *scheduling cycle*. The simulation consists of the repeated execution of these scheduling cycles. Various scheduling criteria which can be applied to this scheduling cycle in order to generate a feasible *deterministic, nonpreemptive* schedule are discussed and examples of the entire procedure are given. Finally, the resulting feasible schedule is used to extract such figures of merit of the multiprocessor simulation software implementation as *average processor utilization* and *execution time of a complete scheduling cycle.*

These figures of merit are obtained as a function of the number of processors used in the concurrent execution of the simulation software, N. An examination of these figures of merit is shown to reveal such quantities of interest as approximate run time of the simulation versus the number of processors employed as well as the maximum number of processors which can be efficiently supported by the degree of concurrency present in the simulation software. If the simulation software is in a development stage it is shown how these (and other intermediate) results can be used to optimize the simulation software package for execution on multiprocessor systems.

2.1 Basic Concepts in Graph Theory

Any discussion of graph theory must start with the definition of a graph.

Definition: Let N be a finite set, and L be a binary operation on N. A graph is defined as an ordered pair (N, L). The elements in N are called nodes, and the ordered pairs in L are called the links of a graph.

A graph can be represented by a matrix, either a node-node matrix or a node-link matrix. A node-node matrix is a matrix in which columns and rows both represent nodes in a graph. In a node-link matrix, either, rows represent nodes and columns represent links or rows represent links and columns represent nodes. The entries in the matrix represent the connection between nodes or between nodes and links.

Consider a graph G with n nodes and m links. For the node-node matrix A, we have:

$$A(i,j) = \begin{cases} 1, \text{ if a link exists between node } A_i \text{ and } A_j; \\ 0, \text{ if a link does not exist between node } A_i \text{ and } A_j; \end{cases}$$

where $1 \leq i,j \leq n$. For the node-link matrix B, we have:

$$B(i,j) = \begin{cases} 1, \text{ if node } B_i \text{ is connected with link } l_j; \\ 0, \text{ if node } B_i \text{ is not connected with link } l_j; \end{cases}$$

where $1 \leq i \leq n$ and $1 \leq j \leq m$. Here rows represent nodes and columns represent links. And for the link-node matrix C, we have:

$$C(i,j) = \begin{cases} 1, \text{ if node } C_j \text{ is connected with link } l_i; \\ 0, \text{ if node } C_i \text{ is not connected with link } l_i; \end{cases}$$

where $1 \leq i \leq m$ and $1 \leq j \leq n$. Here rows represent links and columns represent nodes.

As an example of the node-node matrix and the node-link matrix, consider the graph shown in Figure 1.1. The node-node matrix is:

$$A = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array}$$

The node-link matrix is:

$$B = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} L1 & L2 & L3 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \end{array}$$

And the link-node matrix is:

$$C = \begin{array}{c} \\ L1 \\ L2 \\ L3 \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \end{array}$$

For a graph with n nodes and m links, the node-node matrix is a $n \times n$ matrix and the node-link matrix is either a $n \times m$ matrix or a $m \times n$ matrix. Algorithms which use these matrix representations of a graph will require at least $O(n^2)$ time for a node-node matrix and $O(nm)$ time for a node-link matrix.

A graph can also be represented by linked lists structures in computer implementations. This representation faithfully represents the full graph including the nodes and links. An algorithm using this representation has two major advantages: it can be understood more easily since it uses simple graphical operations

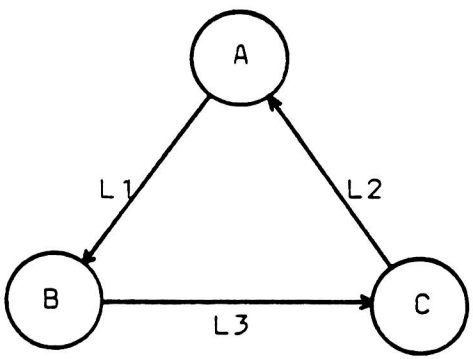

Figure 1.1. Example of a Graph

and the complexity of algorithms using this representation is usually less than the complexity of algorithms using matrix representations. For details of this representation, see [1, 2, 3].

The notion of a *graph* or, equivalently, of nodes connected by links, lends itself to use as a descriptor of program flow. Indeed, if nodes are associated with tasks and links are associated with data transfer paths between tasks, it is easily seen that a graph could naturally describe the *flow of data* in a program. This notion is not novel. Data flow concepts have long been used in visualizing computational activity in parallel machines. This is in contrast to the basic conceptual model used to represent computers which was introduced by John von Neumann (also known as control-driven computation).

Data flow computers are based on the concept of data-driven computation, which is drastically different from control driven computation. The concepts of control flow and data flow computing are distinguished by the mechanism which controls the flow of computation sequences. In the traditional sequential control flow model, there is a single thread of control which is passed from instruction to instruction. This means that the program has complete control over instruction sequencing. Synchronous computations are performed in control flow computers using centralized control. In the data flow computing environment instructions are activated by data availability. Instructions are examined to reveal the operand availability, upon which they are executed immediately if the functional units are available. This implies that many instructions can be executed simultaneously and asynchronously. A high degree of implicit parallelism is expected in a data flow computer.

A data flow program may be mapped to a graph, called a data flow graph, to exploit parallelism in a program more naturally. This asynchronous parallelism can be exploited not only at the instruction level but also at the procedure level. A data flow graph representation of a software unit is a directed graph whose nodes represent tasks and links represent data paths. A task may be an instruction, a procedure, or even a program. The actual meaning of a task is dependent on the application. However, it should be consistent throughout the entire graph.

If the tasks are operators, then the data flow graph demonstrates sequencing constraints, consistent with data dependencies, among instructions. Each task can be performed on the data depending solely on data availability; each instruction can be executed as soon as its operands become available provided that a processing unit is also available. Hence, a maximal concurrency can be found from a data flow graph representation of a set of instructions. Similar results can be obtained if the tasks are procedures or functions in a program. By using data flow concepts and the data flow graph, the sequencing constraints among procedures or functions in a program can be found, as well as the maximal concurrency among them. In this case, the sequencing constraints are an outcome of the existing precedence relationships among these procedures or functions. This level of parallelism in system simulation software can be naturally exploited with

basic functions contained in the simulation algorithm represented as tasks in the data flow graphs. These computational tasks are then suitable for parallel processing using multiprocessor architectures.

Even though the data flow graph representation of software applies to large classes of software systems there are additional constraints which are met by the general class of simulation software systems and which are fundamental to our approach. These additional constraints are characteristics exhibited by system simulation algorithms and rigorously justify our approach.

2.2 Characteristics of Algorithms Used in System Simulation

An inspection of algorithms used in system simulation reveals some useful characteristics. The nature of the algorithms themselves combines with the programming language characteristics to generate some common traits which can be used to broadly characterize a class of software used to implement system simulations. The programming language characteristics used in this characterization are general in nature and are not used to restrict the programming language to the more sophisticated set of languages. These characteristics are:

1. Periodic with infinite number of cycles. In systems simulation tasks, a continuous stream of incoming data samples is processed to produce another continuous stream of outgoing data samples. The same simulation algorithm will be executed repeatedly to handle this endless stream of input data. Each execution of the algorithm can be viewed as a cycle or period. So, the entire simulation can be viewed as a periodic task with an infinite number of cycles with respect to the lifetime of the simulation.
2. Data structures are in block form.
3. Processing of data in block form. One of the natural and efficient ways to handle the simulation's input data is to divide this data into blocks and process them one block at a time. Each data block is then naturally represented by a one or multiple dimension array.
4. Heavily iterative. Due to the concept of state (or signal) space and to the convenient way of representing a system of simultaneous linear equations using matrices, the matrix is the dominant tool used in system simulation. A matrix can be naturally represented by a multiple dimensional array. The processing of any matrix operation requires a fixed number of repeated operations. These iterative operations can be naturally handled by DO loops. Hence, the algorithm is highly iterative.
5. Nonrecursive. Recursion in a programming language requires a dynamic run-time storage management scheme. However, in many programming languages, only a static run-time storage management scheme is used; recursion, in the programming language sense, can not be implemented.

This is not a major constraint in the design of simulation software since the major trait of such software systems is their iterative feature.

6. Numerical computation intensive. One obvious property of digital simulation algorithms is that they are numerical computation intensive. The performance of this type of algorithm is computation-bound and not input/output (I/O) bound.
7. Decomposable into a set of meaningful basic tasks. Each digital simulation algorithm is composed from a set of basic tasks. A basic task can be implemented as a subprogram.
8. Fixed amount of input and output sample data. Each task takes a fixed amount of input data samples to perform a specific function and generates a fixed amount of output samples. This is not only desirable but it is also imposed by a characteristic shared by many programming languages.
9. Data value independent. Each sample processed by a digital simulation processing task will go through exactly the same operations. So, the value of the sample data will not affect the sequencing of operations. This is a very desirable characteristic.
10. Constant execution time. Since each task will perform the same amount of operations each time it executes, the execution time of each task is fixed for each execution. Hence, the execution time of the entire algorithm is also fixed for each execution.

Having developed all the preliminary concepts necessary for the remainder of the chapter, we are now ready to apply them to the development of the simulation software design and analysis approaches.

3 DATA FLOW GRAPHS IN SIMULATION SOFTWARE

The data flow graph discussed earlier is adapted to represent system simulation algorithms with the restriction that the nodes of the data flow graph now represent basic tasks or functions contained in the algorithm. The general concepts are discussed and two associated problems are outlined. The following definitions will facilitate the rest of the presentation. Whenever possible, a common nomenclature is maintained with previous literature.

Definition: A cycle of a system simulation algorithm is defined as the complete execution of the algorithm without repeating its state.

Definition: The sample rate is defined as the required number of samples either for input or output of a node in a link.

The required number of input samples for a node is called the input sample rate and the number of samples generated at the output of a node is called the

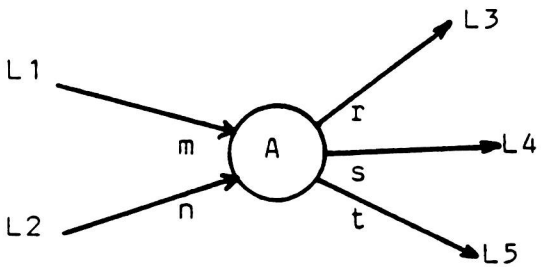

Figure 1.2. Input and Output Sample Rates Associated with a Graph Node

output sample rate. The input and output sample rates are given for a particular link of a particular node. Therefore, different links of a node may have different input or output sample rates. On the data flow graph, the sample rates are attached to the links for easy reference. For example, a node with two input links and three output links will look as shown in Figure 1.2. The links are labeled $L1$, $L2$, $L3$, $L4$, and $L5$. $L1$ and $L2$ are input links and $L3$, $L4$, and $L5$ are output links. The symbols m and n denote the input sample rates for node A from links $L1$ and $L2$, respectively. The symbols r, s, and t denote the output sample rates from node A to links $L3$, $L4$, and $L5$, respectively. The input samples are consumed by node A and the output samples are produced by node A. Before the start of execution of node A we need m samples from $L1$ and n samples from $L2$. At the end of the execution, there are r, s, and t samples produced by node A on link $L3$, $L4$, and $L5$, respectively.

Definition: A buffer is defined as a temporary storage device for a link to hold the output samples produced by a node.

Definition: A delay is defined as the initial number of available samples stored into a buffer before the execution of a cycle.

The size of delay present in the link is written in the middle of the link between two nodes in a data flow graph. For example, in Figure 1.3, d is the size of the delay of the link between node A and node B, m is the input sample rate for node A, and n is the output sample rate for node B. The minimum size of a buffer on a link between two nodes depends on the input and output sample rates of the link between these two nodes. This will be discussed later in more detail.

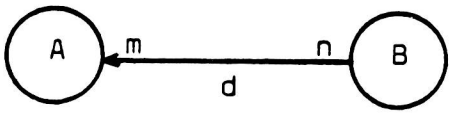

Figure 1.3. Delay between Two Nodes

Definition: The repeat factor of a node is defined as the number of times that the node will be executed within one basic cycle.

The repeat factor is unique for each node. This is due to the characteristics of a system simulation program. In the rest of this chapter, the repeat factor will be attached to a node as an exponent to the name of the node. For example, if the repeat factor for node A is m, then we represent this as A^m. The repeat factor is always a positive number. Its minimum value is 1. That is, we do not consider dead nodes in the data flow graphs (tasks which will never be executed).

Definition: A consistent sample rate is said to exist when the number of samples produced by the initial node equals the number of samples consumed by the terminal node for each link between these nodes during a complete cycle.

For example, consider the graph shown in Figure 1.4 where m is the input sample rate, n is the output sample rate, k is the repeat factor for node A, h is the repeat factor for node B, and d is the size of the delay for the link between nodes A and B. For a consistent sample rate, we have:

$$k \times m = h \times n$$

Another example is a simple circuit with two nodes as shown in Figure 1.5. For a consistent sample rate, we have:

$$k \times m_2 = h \times n_2$$

$$k \times m_1 = h \times n_1$$

When all sample rates are consistent, at the completion of a cycle the size of allocated storage in the buffer is equal to the size of the delay of a link. In other words, the size of the delay of a link at the beginning of each cycle is constant. If the sample rate is not consistent for a link, then the size of the sample stored in the buffer will grow indefinitely for that link. This will create an unstable situation where all available memory will eventually be consumed at which time the system will malfunction. In this chapter, we only consider data flow graphs with consistent sample rates. Even after the introduction of these definitions, the data

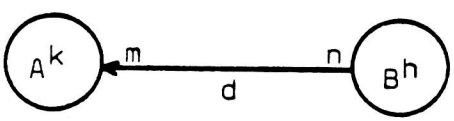

Figure 1.4. Example of Consistent Sample Rate

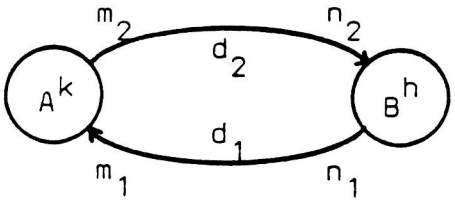

Figure 1.5. Example of Consistent Sample Rates

flow graph can still be represented by a matrix form, the node-link representation. Each column represents a node and each row represents a link. The input sample rate is represented by a negative number and the output sample rate is represented by a positive number. Unfortunately, the delay cannot be represented in the matrix. For example, consider the data flow graph shown in Figure 1.6. Its node-link matrix will be:

$$M = \begin{array}{c} \\ L1 \\ L2 \\ L3 \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} -1 & 0 & 2 \\ 1 & -2 & 0 \\ 0 & 1 & -1 \end{pmatrix} \end{array}$$

For more on these representation, see [4]. This representation has a significant drawback since each algorithm using this matrix representation will have a complexity of at least $O(nm)$, where n is the number of nodes and m is the number of links.

For a data flow graph of a given simulation software system, there are two issues which may significantly affect performance: the minimum size of the buffer of each link and the repeat factor for each node. The second problem has

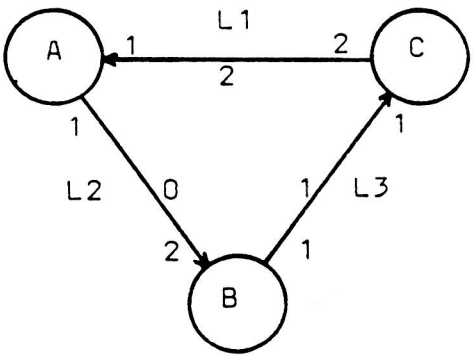

Figure 1.6. A Data Flow Graph for NodeLink Matrix