VHDL-based Methodology for Modeling Performance of Parallel Systems

Praveen Chawla and Herbert L. Hirsch
MTL Systems, Inc.
3481 Dayton-Xenia Road
Dayton, Ohio 45432-2796
pchawla@mtl.com,hhirsch@mtl.com • (513) 426-3111

Abstract

Due to continuing growth in the complexity of computing applications, the conventional uniprocessor computers often do not provide adequate processing power to obtain a desired system response in an acceptable amount of elapsed time. Parallel processing techniques offer a potential for speed-up in a variety of applications. However, in order to exploit parallel processing techniques for an application of interest, a parallel computing system designer has to correctly make several important decisions, such as selection of an appropriate parallel computing platform, configuration of the chosen platform and assignment of application software to processors of the platform. This paper describes a VHDL-based methodology for modeling performance of parallel systems which can be utilized to make such decisions.

Section 1. Introduction

Although parallel computing systems have been successfully employed to speed-up a variety of applications, they have still not become a viable solution for accelerating all applications. This has been mainly because of one or both of the following reasons:

- Large interprocessor communications overheads among commercially available parallel computing platforms
- Lack of inherent parallelism in the application of interest.

Therefore, before committing to a parallel computing platform and expending efforts in porting an application to execute on the chosen platform, it is important to evaluate the available parallel platforms in terms of their responsiveness, effectiveness and failure-related behavior with respect to the application of interest. Such evaluation, often termed as *performance modeling*, enables appropriate parallel system selection, design, upgrade, tuning and analysis.

Although performance modeling can be done through various means [10,11], this paper will only address the modeling and analysis scheme based on discrete event simulation. Under this scheme, a performance model is created for each system component. These components are then interconnected to create a performance model of the system. Simulation of the system model leads to generation of performance statistics, such as throughput, latency and utilization, which aid in making decisions related to selection, design, upgrade, tuning and analysis.

A VHDL-based [8] methodology for modeling the performance of parallel computing systems is particularly suitable because of the following reasons.

A. It is easy to create a hierarchical model of the parallel computing platform and the application of interest using structural and behavioral VHDL constructs. A system component can be easily described using an entity-architecture pair.

Furthermore, a component can be described at varying levels of abstraction and details using multiple architecture definitions for an entity.

B. Availability of parallel constructs in VHDL such as process, block, component, and concurrent signal assignment make it easier to model parallel computing systems. Note that parallel constructs are executed concurrently during VHDL simulation.

C. The concept of time is built into the VHDL language. Therefore, VHDL models do not require code to maintain a simulation clock as opposed to a functional or an object-oriented programming language. In addition, the wait statement and the after clause in a signal assignment statement provide an easy way to model component delays.

D. VHDL is an IEEE standard and, therefore, VHDL models are easily portable across computing platforms.

E. Performance modeling in VHDL offers an opportunity for a single path design environment from concept to implementation because of VHDL's growing usage for designing and documenting digital system designs.

There have been a few VHDL-based performance modeling efforts in the recent past [1,2,5,7,12,13]. Most of the efforts have used VHDL for Petri-net based stochastic models and their discrete event simulation. Srivastava [13] has utilized VHDL models for continuous simulation as well. Although some of the aforementioned efforts have been directed towards modeling parallel systems [5,12], their methodology has been to use a stochastic model for the application software, which drives a queuing network, Petri-net based model for the parallel computing platform. This paper describes a methodology to model parallel computing systems which we believe is more versatile, powerful and generic than the ones previously proposed.

Under our methodology, a parallel computing system is modeled using three major components, namely an *Application Software Object* (ASO), a *Global Operating System Object* (GOSO) and a *Parallel Computing Platform Object* (PCPO). The ASO, in turn, consists of *Application Objects* (AOs) whose interaction represents the execution of application software and a *Stimulus Object* (SO) that drives these application objects with primary inputs. The GOSO represents an agent that provides the basic brokering service between the application software and the parallel computing platform hardware. The PCPO consists of *Processor Objects* (POs), each of which represents a processor model and a *Communication Structure Object* (CSO) that interconnects POs. All component models can be described at any level of detail. However, their interfaces to other component models are not changed (i.e., their interfaces are system- and detail-independent). This allows a modeler to obtain estimates of performance at different stages of system design. The estimates become more accurate as more detailed-level models are used.

We have organized the rest of the paper into four sections. Section 2 provides a detailed description of the methodology. Section 3 describes an implementation of the methodology in a CAD tool. Section 4 illustrates our methodology with an example. Section 5 presents our anticipated future work.

Section 2. Methodology

This section is divided into two subsections. The first subsection defines some of the important terms and provides an understanding of the important issues. The second subsection presents a detailed description of the methodology employed to model the performance of parallel computing systems.

2.1 Terminology

In general, a *computing system* is a collection of hardware and software components that have been interconnected to solve a problem. The software

components are usually classified into *application software* components and *system software* components. A *parallel computing system* is a computing system in which multiple software components (application or system) can execute simultaneously. Some form of parallelism is usually provided in almost all contemporary computers. However, we usually refer to a computing system as a parallel computing system if it is equipped with multiple hardware components called *Central Processing Units* (CPUs). Loosely speaking, performance of a parallel computing system can be defined as a measure of its *efficiency* in serving its purpose. Efficiency is usually measured in terms of its *latency, utilization* and *throughput*.

The *application software* that executes on a parallel computing platform has to be decomposed into parts. Each resulting part executes on a processing element and communicates with other parts when needed. Such an execution of application software can be described in terms of a set of interacting *processes*. In such a description, each process represents a unique part of the application software in terms of its input-output transformations. Each process has a set of inputs. A change in an input that belongs to a subset of process inputs, referred to as the *sensitivity set*, triggers the process to execute. Each process execution requires zero time and may result in outputs. These outputs become *effective* after a time delay. The outputs can be all produced for the same time in the future or may become effective after different time delays. The outputs generated during a process execution are sent to other processes whose execution depends on them. However, the receiving processes do not use the arriving outputs until they become effective.

A *parallel computing platform* is defined here as a collection of *processing elements* which communicate with each other over an *interconnection network*. Processing elements can have the same computational capability, thereby offering a *homogeneous computing environment*, or can be of different computational capability and thus offer a *heterogeneous computing environment*. A processing element can be a general-purpose processor (e.g., Intel i860, Intel 80486) or a whole computer (e.g., a Sun Workstation, an IBM PC). Processing elements may or may not share memory. Processing elements may be interconnected by a fast communication network like a 2-D mesh or a k-nary n-cube. On the other hand, they might communicate with each other over telephone lines.

An *operating system* serves as a broker between the application software and the parallel computing platform. It provides access to parallel computing hardware when application software components request them and resolves any conflicts that might arise because of multiple requests for the same resource e.g., a processing element.

2.2 Description

We now describe a methodology to evaluate the performance of parallel computing systems. Under the methodology, we create VHDL models of hardware and software components and interconnect them to create a system model. The system model is simulated, which results in the generation of performance data. The methodology is unique because it defines system-independent interfaces for all categories of components. By doing so, we are able to connect the same set of application software components to a variety of system software and hardware components (parallel computing platforms). This aids the selection of an appropriate parallel computing platform from the ones that are available. Furthermore, we can simulate a variety of system configurations by interconnecting software and hardware components in a different manner (assignment). This assists us in assigning (connecting) software components to hardware components in an optimal manner. In addition, if the number of software components is greater than the number of hardware components, we can increase the number of hardware components and reassign software components. This helps us in exploiting the parallelism inherent in the application software by maximizing concurrency of execution and thereby possibly leading to lower latency, higher throughput and higher utilization.

As indicated in the previous section, we model hardware and software components

of a parallel computing system using three major components, namely ASO, GOSO and PCPO. In the next three subsections that follow, we describe each major component and its subcomponents. In addition, we provide details about each component interface. Figure 1 shows the interaction between these three components.

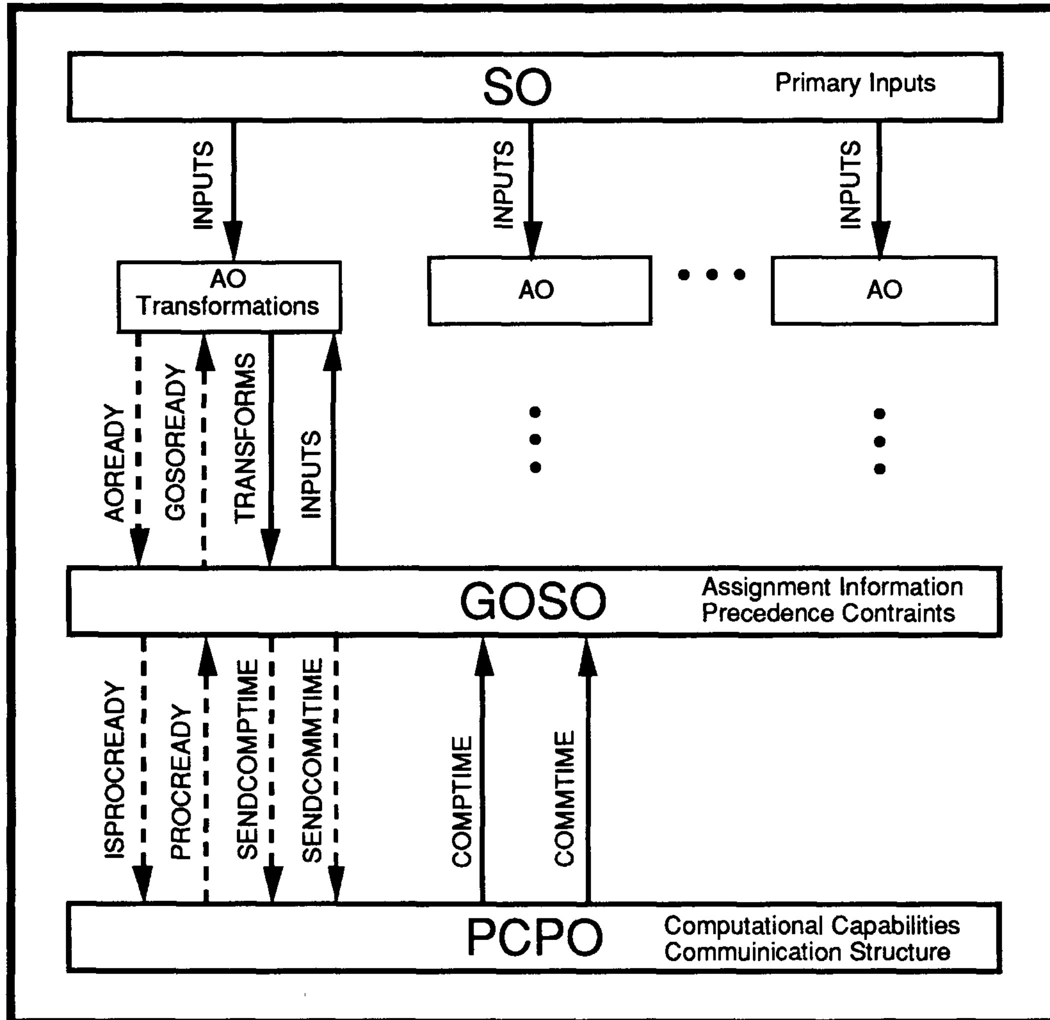

Figure 1. PSDA Data and Control Flow

2.2.1 Application Software Object. The ASO consists of AOs and a SO. An AO represents a component of the application software. Each AO has an interface consisting of a set of data inputs, data outputs and control signals. An AO receives its inputs at different points of time. An event on any of its inputs belonging to a subset called the sensitivity set triggers its execution. When ready to execute, an AO requests the GOSO to schedule its execution on the assigned processor by sending a control signal called *AOREADY*. When the GOSO is ready to schedule execution of an AO, it sends that AO a *GOSOREADY* signal. Upon receipt of a *GOSOREADY* signal, the AO begins its execution. During its execution, a set of transformations are executed upon satisfaction of appropriate boolean conditions on inputs and outputs. A transformation is of the form:

Output := expression (inputs, outputs) after instructions

Where: "instructions" is the number of instructions required to execute the transformation.

"expression" may represent an actual computation that is performed in the application software or may be a boolean expression of inputs and outputs.

In the former case, Output represents the actual output generated by the application software. In the latter case, Output is just a boolean variable which attains a true or a false value. Each transformation results in a *TRANSFORM* message for the GOSO. Each *TRANSFORM* message to the GOSO contains the identifier (id) of the sending AO, a measure of the amount of computation performed by the transformation e.g., number of instructions, the id of the output, size of the output value e.g., number of bytes, and value of the output generated. Upon receipt of a *TRANSFORM* message, the GOSO sends the output value contained in the *TRANSFORM* message to each AO that is affected by it (determined from precedence constraints).

Besides AOs, the ASO also includes a SO which provides the stimulus for AOs. The SO is responsible for sending primary inputs to AOs at desired time instants.

2.2.2 Global Operating System Object. The GOSO acts as the middleman between the ASO and the PCPO. When it receives an *AOREADY* signal, it queries the PCPO about availability of the processing element assigned to the requesting AO by sending the PCPO the *ISPROCREADY* signal. The PCPO responds with *PROCREADY* signal when the requested processor is ready for execution. Upon receipt of *PROCREADY* signal, the GOSO sends a *GOSOREADY* signal to the requesting AO. The AO, in turn, executes its transformations and sends TRANSFORM messages to the GOSO. Upon receipt of a *TRANSFORM* message, the GOSO checks the fanout list for the output value contained in the *TRANSFORM* message and sends out the value to all the AOs in the fanout list after a delay of computation and communication time. The computation and communication time delay is obtained from the PCPO by sending *SENDCOMPTIME* and *SENDCOMMTIME* messages. The PCPO responds with *COMPTIME* and *COMMTIME* messages containing computation time for transformation and time to communicate output values between AO pairs, respectively. *COMPTIME* is obtained for each *TRANSFORM* message whereas *COMMTIME* is obtained for each AO in the fanout list of the output value.

Besides synchronizing the execution of AOs on the PCPO, the GOSO also collects performance statistics such as execution times, messages sent between AOs and processor utilization. These statistics are reported at the end of simulation.

2.2.3 Parallel Computing Platform Object. The PCPO is a model of the parallel computing platform. The parallel computing platform is described by the number of processing elements, their processing capabilities, and a communication structure that determines the amount of time it takes to send a message from any given processing element to any other processing element. A PO model is used to describe the processing capability of a processing element whereas a CSO model is used to describe the interconnection network between the POs.

The PCPO interacts only with the GOSO via control and data signals. Control signals to the PCPO consist of *ISPROCREADY*, *SENDCOMPTIME*, and *SENDCOMMTIME*. Control signals from the PCPO consist of *PROCREADY*. Data sent by the PCPO consists of *COMPTIME* and *COMMTIME*. *COMPTIME* can be calculated by using the measure of computation in the corresponding *TRANSFORM* message. For example, the *MIPS* rating of a processing element can be used to calculate *COMPTIME* if a *TRANSFORM* uses the number of instructions to measure the amount of computation represented by the *TRANSFORM*. *COMMTIME* can be obtained from the CSO model and size of the output value that needs to be communicated.

Section 3. Implementation

We have implemented a CAD tool, called the *Parallel System Design Assistant* (PSDA) [4], to demonstrate our methodology. Figure 2 shows the organization of the PSDA. In the current implementation of the tool, the user provides the following:

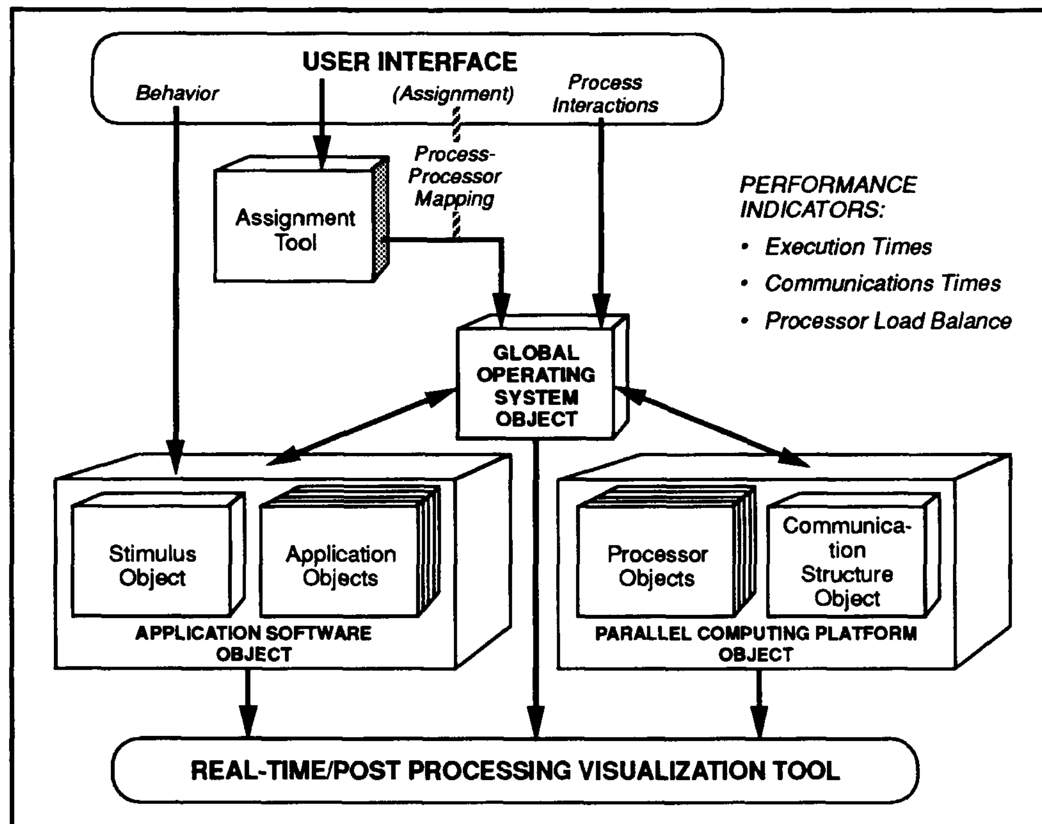

Figure 2. Parallel System Design Assistant (PSDA) Organization

A. A description of application software as a set of interacting processes. The description can be provided by the user in textual or graphical form. The description should clearly specify the inputs to each process, its sensitivity set and its outputs. In addition, connections between inputs and outputs of various processes should be clearly defined (precedence constraints). Behavior of each process is specified by how it transforms its inputs into outputs and the amount of computation required to do so. The behavior can be specified either just in terms of the amount of computation required, without actually describing how the transformation is accomplished (uninterpreted modeling), or by actually describing the transformation (interpreted modeling). The amount of computation required is specified in terms of the number of instructions that will be executed to perform the desired transformation.

B. Assignment of processes to processing elements. The assignment can be specified by the user or can be generated automatically by an assignment tool which has not been implemented at this point. Assignment, if generated using the assignment tool, will attempt to minimize total execution time by minimizing communication, maximizing concurrency of execution and balancing the computational load [3].

C. A model of parallel computing platform (unless it is part of the library provided with the tool). The model should specify computational capability of each processing element in terms of its MIPS (million instructions per second) rate. Furthermore, it should specify the interconnections between processing elements.

Given all the inputs, an application description is translated into a hierarchical VHDL model for the ASO. The ASO, in turn, consists of AOs and a SO. Each AO corresponds to a process in the application software. The SO drives the AOs with an appropriate stimulus. Furthermore, a hierarchical VHDL model is created for the PCPO. The PCPO, in turn, consists of POs and the CSO. The assignment and precedence constraints are combined with the generic portion of the GOSO to form a complete VHDL model of the GOSO. The PSDA simulates execution of application software on the chosen parallel computing platform using the given/generated assignment. Upon completion of a simulation, the PSDA reports back at least the following performance data items:

A. Total execution time,

B. Number of messages sent between processing elements of the parallel computing platform,

C. Average utilization of each processing element.

In addition, a visualization tool will be provided to assist in real-time and post-processing visualization of the system performance. The visualization tool has not been implemented at this point.

Section 4. Example Application

As a part of an ongoing project at MTL Systems, Inc., we have successfully utilized the PSDA to investigate ways to execute antenna simulation computations [6,9] in real-time. The real-time requirement imposed on this antenna simulation requires an output in less than 2000 μs after the inputs have been received by the antenna model. An antenna simulation executed on a uniprocessor computer does not meet the real-time needs. We have used our VHDL-based methodology to investigate antenna simulation performance on several parallel computing platforms. As a result, we have recommended a parallel computing platform configuration and the assignment of application software to the processors of the parallel computing platform to meet the real-time requirement, minimize cost and maximize speed. In the ensuing discussions, we describe the software components that are required to execute for simulating an antenna and the parallel computing platforms that we investigated. In addition, we provide various combinations of software and hardware interconnections and the simulation statistics that we utilize to iteratively arrive at an optimal assignment.

Figure 3a shows various computations that we are required to perform in order to simulate antenna effects and the precedence constraints between these computations [6]. There are ten components that constitute the application software. Figure 3b and 3c show the assignment of these components to processing elements of two parallel computing platforms. It is assumed that the assignment is static and that the application software components, once assigned to a processing element, do not change their assignment during execution. The parallel computing platforms we evaluated for this application were:

A. An Intel 80386-based personal computer with 4 MB of primary memory and enough disk space to store the application software,

B. A personal computer with two Intel 80386 processors connected by a fast local bus (200 μs required to send a message from one processor to another) and 4 MB of primary memory each,

C. A personal computer with 4 Intel 80386 processors connected by a fast local bus (200 μs required to send a message from one processor to another) and 4 MB of primary memory each (A and B are not commercially available and will have to be custom-made if proven promising).

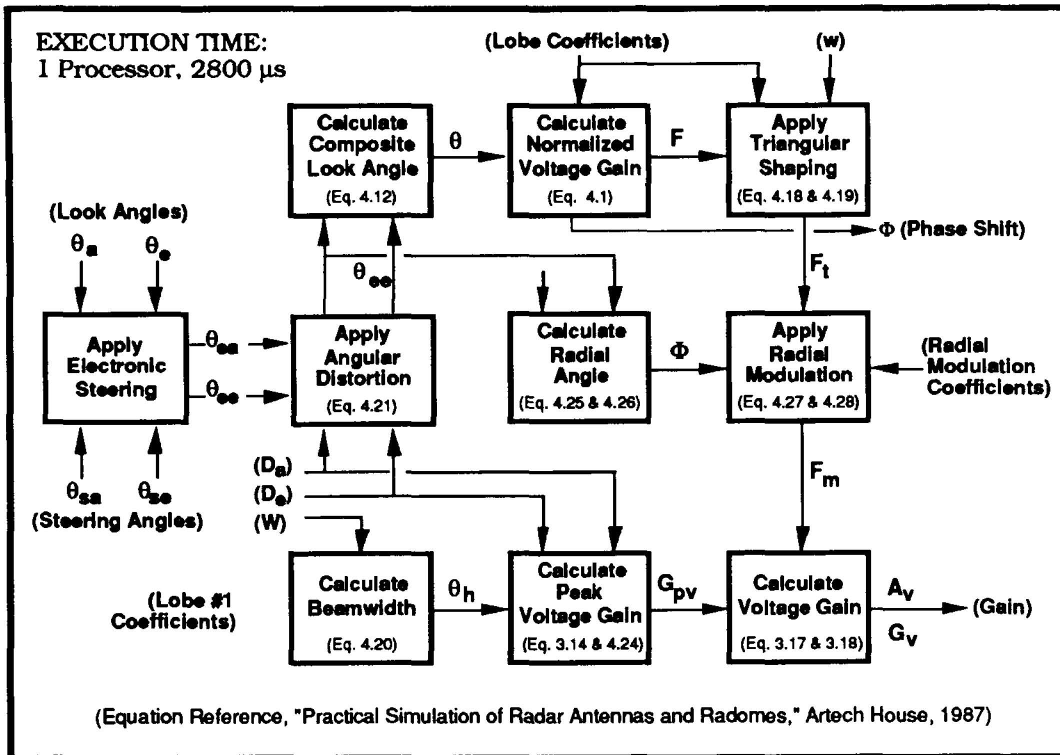

Figure 3A. PSDA Example Application (1 Processor)

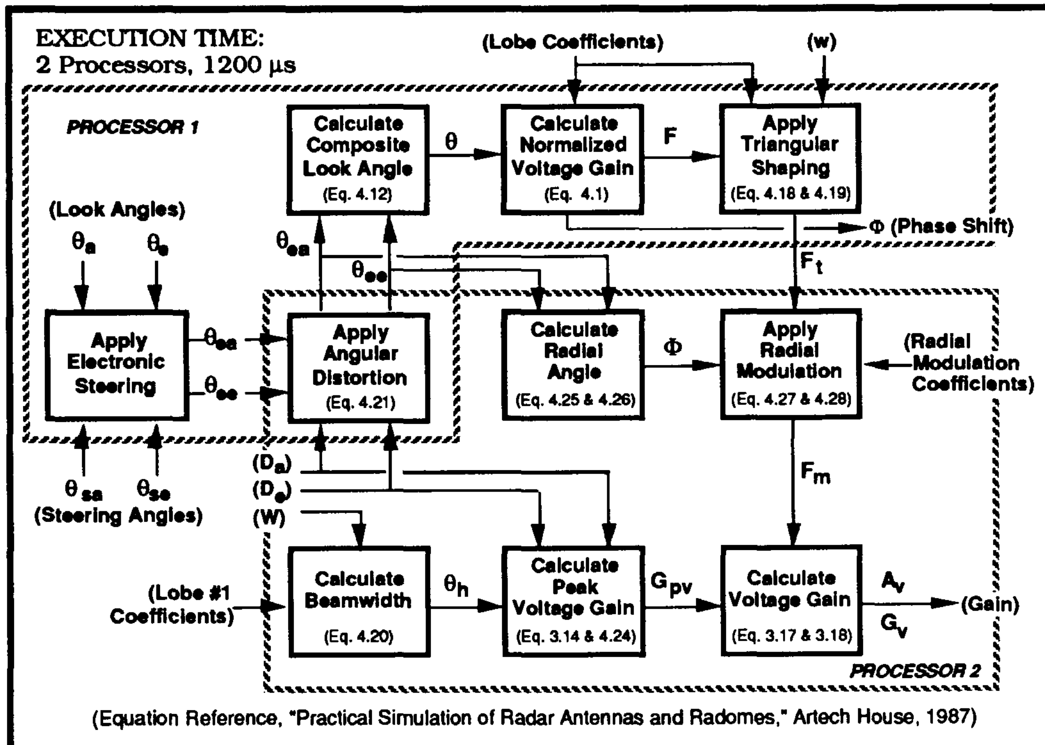

Figure 3B. PSDA Example Application (2 Processors)

VHDL models for hardware and software components of the three aforementioned parallel computing platforms were hand-crafted and simulated. The simulation results showed that a uniprocessor case required 2800 μs whereas the 2 processor and the 4 processor cases required 1200 μs and 1800 μs, respectively. The result did match with what we expected intuitively for this application. Since there are only two parallel paths that can be executed simultaneously, a 2 processor computer was able to exploit that fact and execute at almost twice the speed. However, the four processor case ran slower than a two processor case because of communication delays and lack of inherent parallelism to exploit.

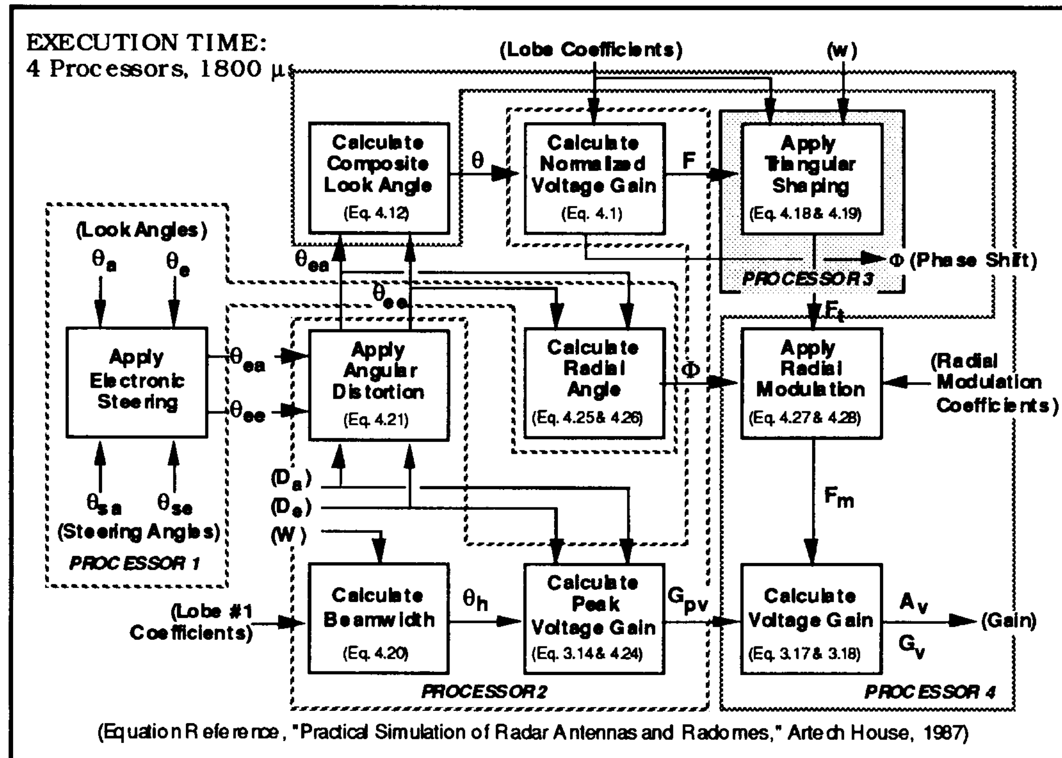

Figure 3C. PSDA Example Application (4 Processors)

In addition to execution times, the PSDA provides a variety of other performance data such as processor utilization, utilization of communication paths/structure, granularity of application (computation to communication time ratio) and total time that an application software component waits after it is ready to execute. These statistics provide answers to questions about responsiveness, effectiveness and failure-related behavior of a parallel computing system

Section 5. Future Work

Our future work includes automating the process of mapping software to hardware. We intend to do this by utilizing research done at the University of Cincinnati as part of a DARPA-sponsored effort [3]. In addition, we are investigating techniques to translate high-level specifications of application software and parallel computing platform, specified graphically or textually, into corresponding VHDL models. We are also developing techniques to visualize the simulation data in real-time and in the post-

processing phase. Furthermore, we are building a library of commercially available parallel computing platform models. Such a library will assist the process of selecting an appropriate platform to meet the application needs.

Acknowledgments:

We acknowledge and thank Dr. Harold W. Carter, of the University of Cincinnati for providing helpful comments and suggestions.

References:

[1] J. Aylor et al. *Performance and Fault Modeling with VHDL.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[2] T. Carpenter, F. Rose, and T. Steeves. VHDL Architectural Assessment. Personal Communications, 1992.

[3] P. Chawla. Assignment Strategies for Parallel Discrete Event Simulation of Digital Systems. Ph.D. Dissertation Proposal, 1992.

[4] P. Chawla, H.L. Hirsch, and D.P. Geis. Parallel System Design Assistant. In *National Aerospace Electronics Conference*, 1993.

[5] E.D. Cutright et al. Modeling an ATAMM-Based Multiprocessor System using VHDL. Technical Report No. 9100111.0, Dept. of Electrical Engineering, University of Virginia, January 1991.

[6] H. Hirsch and D. Grove. *Practical Simulation of Radar Antennas and Radomes.* Artech House, Boston, MA, 1987.

[7] P. Hubbard and J. Torres. Using VHDL for High-Level and Stochastic System Modeling. In *1990 VHDL Users Group Meetings*, pages 61-70, 1990.

[8] IEEE. VHDL Language Reference Manual, 1987.

[9] MTL Systems Inc. Improved Real Time Simulation of Antenna Effects. Technical Report No. MFR-93-002/SDF239, MTL Systems, Inc., January 1993.

[10] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley, New York, NY, 1991.

[11] K. Kant. Introduction *to Computer System Performance Evaluation.* McGraw-Hill, New York, NY, 1992.

[12] Guru Prasad. An approach to performance analysis of computer systems by high level simulation in VHDL. Master's thesis. University of Cincinnati, OH, 1992.

[13] M.B. Srivastava and R. W. Brodersen. Using VHDL for high-level mixed-mode system simulation. *I EEE Design and Test of Computers*, pages 31-40, September 1992.