Distributed Simulation for Structural VHDL Netlists

Werner van Almsick[1], Wilfried Daehn[1], David Bernstein[2]

[1] SICAN GmbH, Germany [2] Vantage Analysis Systems, USA

Abstract:

This article describes the current state of the project to develop distributed simulation. The reader will have an introduction to the most commonly used algorithms in this research field. The paper deals in particular with the influence of the partitioning of VHDL netlists affecting the control of a distributed simulation tool.

1. Introduction

This paper deals with the actual state of a joint venture project between Vantage and SICAN. Its task is to provide an interface tool for the *VantageSpreadsheet* VHDL simulator in order to make a distributed simulation of netlists described in structural VHDL possible. In particular, this paper describes, analyzes and compares the different algorithms and techniques which can be used for realizing a distributed or parallel digital simulation tool.

Synthesized systems often contain several hundred thousand gates. A simulation of these systems costs quite a lot of computation time or may even be impossible. One way to simulate such big systems is by distributing the simulation. For this, several processors are used for the simulation of one system. Such a simulation, which might exceed the technical bounds of a single system, has the ability to solve the simulation problem with multiple times of addressable memory and other resources. Due to its parallel character, the user receives the result considerably faster. The efficiency of a distributed simulation can be specified with the help of a **speedup** value. This value depends on the system that is to simulate. It is defined as the ratio of t_0 to t_{dist}, where t_0 is the user time that is needed for a single-processor simulation and t_{dist} is the user time needed for the distributed or parallel simulation. In some cases, it is possible to reach a speedup higher than N by using only N processors. The reason for this is the minimization of memory swaps which are very expensive with respect to computation time.

At present, the following three different strategies are favorable for a distributed or parallel digital simulation. The first one is the parallel simulation by parallelizing the program code of the simulator. It assumes that the user has a multi-processor machine. In nearly all cases, a parallel code simulation tool works with one global simulation time. Therefore, all simulation cycles that occur at the same simulation delta can be calculated in parallel by different processors because of their data independence. However, if the event list of the tool is also parallized, a great speedup can be noticed. The big advantage of this strategy is that no partitioning of the VHDL description is necessary. On the other side, there is a maximum speedup given by the number of available processors as well as the number of parallel events in the circuit [5,6].

The second strategy is a distributed simulation which works with one global simulation time. It can be used for tightly (multi-processor systems) and loosely (LAN) connected processor systems. For that, a partitioning of the VHDL system that will be simulated into N VHDL subsystems is required (chapter 3.). The characteristic of the strategy is that after partitioning, one main process starts and controls N independent digital simulators on N processors. Each of the N simulators simulates one of the N subsystems. All necessary signal and time messages of the cut data dependencies will be sent to the main pro-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission. (c) 1994 ACM 0-89791-687-5/94/0009 3.50

cess. This main process distributes these messages to the concerned simulators again. With the help of the global time a synchronization of all simulators can be done. The problem of this strategy is the multitude of messages, which must be handled by the main process. This leads to a high probability of network crashes by sending and receiving messages from/to the main process. Depending on the system that is to be simulated, this reduces the achievable speedup in a LAN.

By replacing central communication control with distributed control, we achieve the third strategy. Here, each simulation process has the ability to communicate with every other simulation process. The only task of the main-control process is to start all sub-processes, give them the information about the single partitions and receive the final results. Each process sends and receives its own messages about a changed signal. In this way communication will be distributed among all processors. As in to the second strategy, simulation of non-simultaneous states is both possible and a goal. On the basis of this distributed control, the speedup can be substantially higher than the two strategies described before. Therefore, a distributed simulation working with this strategy will be discussed in the following chapters.

2. Parsing and Analyzing

For such a distributed digital simulation the VHDL code must be parsed and analyzed. The parser provides a description of the VHDL in graph form. This can be done by parsing the VHDL code or a VHDL Intermediate Format (VIF). VIF is created by a VHDL compiler such as the one used. By using an intermediate format one saves the work for a direct parsing of the complete VHDL standard. In our project we confine ourselves to the parsing of structural VHDL code. Synthesized VHDL systems are mainly described in a netlist form with instantiated COMPONENTs. If it is necessary to simulate an ARCHITECTURE with a few behavioral statements, the embedding of these statements into sub-ENTITIES produces an ARCHITECTURE without behavioral statements. So in most cases this represents no restriction.

VHDL supports multiple sources for a signal; the value of the signal is computed by a resolution function. This function determines the value based on one or more driving values inside the circuit. Since it is impossible to distribute this function between two or more independent simulators, cutting a resolved signal is never allowed. So, the parser builds a union node for all those nodes which are connected to such a signal, or it tries to resolve all components which are connected to this signal in order to get signal connections without any resolution function.

The parser should also have the ability to work hierarchically. Based on the size of the parsed graph, the parser decides whether a resolving of an instantiated COMPONENT should be resolved. One occasion for resolving a COMPONENT is that on the one hand the number of nodes must be quite larger than the number of partitions that should be built and on the other hand a very large graph slows the whole distributed simulation.

Inbetween providing the graph and building up the partitions, an analysis of the graph is advisable to determine the simulation loads of the nodes and the communication costs between the nodes. An exact estimation of the simulation loads from the VHDL code is very difficult or probably impossible. A previous simulation of the circuit for calculating the probable simulation load is often unreliable and needs a lot of computation time. Therefore, in most cases it is better to assume that all nodes have the same simulation load. Besides, the designer must be able to assign special simulation loads for instantiations whose loads strongly deviate from the average. This can be done by a configuration file. The second value, the communication cost of one edge, i.e. a direct connection between two COMPONENTs, can be calculated by using the signal width. In addition to the cost of one bit B an offset A (constant cost) is required. This varies according to the protocol and synchronization information between the two processes if this edge is cut. For a LAN its value is very large and for a multi-processor machine it is very small.

In addition to this all cycles inside the graph will be marked. A cycle represents a feedback in the system that is to be simulated. A node that is inside of such a cycle is called a *strong component* [1]. The localization of all *strong components* that build one cycle is important for the further work; in particular it has a strong influence on the control system of the simulators (chapter 4).

Up to now this paper assumes that the number of partitions N that should be built is a fixed value that is given by the computer environment. However, the efficiency of a distributed simulation can rise if an adaptation of this value to the graph and computer environment is possible. In reality only the upper limit of N is a fixed value and this is equal to the number of available processors P. A distributed simulation has the goal that all simulators run as parallel as possible. This assumes that all partitions have nearly the same simulation load. Hence, the following value N_{SL} describes the second limit for N if N_{SL} is less than or equal to P. In the formulas, SL_i is the simulation load of the node or union i and m is the number of nodes inside the VHDL graph. It is

$$N_{SL} = \frac{\sum_{i=1}^{m} SL_i}{SL_{max}} \quad -(1)$$

with

$$SL_{max} := Max\{SL_1, SL_2, \ldots, SL_m\}$$

the second limit that determines a valid interval for N.

$$N_{SL} \leq N \leq P \quad if \quad P \geq N_{SL} \quad -(2.1)$$
$$1 < N \leq P \quad if \quad P < N_{SL} \quad -(2.2)$$

Since a big value of N increases the possible speedup, N should be set to the upper interval limit.

3. Partitioning

The task of partitioning is to divide the graph into N subgraphs. By projection of these subgraphs onto VHDL systems the N partitions will be built up. The **quality** and the **relationship** (*RS*) between the partitions are responsible for the achievable speedup of the distributed simulation. Two factors are important for the **quality**. The first one is the deviation of the *simulation loads* of the single partitions from the optimal values. If only computers with equal performance are used, the optimal value is given by the overall *simulation load* divided by the number of partitions N. By using different computer types the distribution of the *simulation loads* should depend on the different performances. The second value that influences the quality of partitioning is the sum of the *communication costs* among all cut signals including the offsets for the first cut signal between two partitions. For a given number of partitions N and a given optimal distribution for the *simulation loads* this sum is to be minimized.

The feedbacks of the circuit or the cycles of the graph respectively characterize the **relationship** (RS_{1-2}) between the partitions. As it will be shown in chapter 4, the **relationship** determines whether it is possible to apply an effective kind of simulation control. The following four different relationships should be defined. For the case that all connections between two partitions P_1 and P_2 have the same direction, the **relationship** is called unidirectional ($RS_{1-2} = UD$). This implies that no external cycle exists between the partitions P_1 and P_2. If connections in both directions exist but no external cycle between the elements of P_1 and P_2, the relationship is called bidirectional ($RS_{1-2} = BD$). The third case is that the elements of the partitions P_1 and P_2 build one or more external cycles. This case is called feedback-full ($RS_{1-2} = FB$). The case that there are no connections between the partition P_1 and P_2 is called unconnected ($RS_{1-2} = UC$). If the partitions, but not their internal graphs, are visible, then the cases $RS_{X-Y} = BD$ and $RS_{X-Y} = FB$ are identical.

The rest of this chapter describes a way to produce partitions which increases the percentage of unidirectional and unconnected relationships. We call this partitioning algorithm *Levelizing*.

It can be proved that for every directed acyclic graph G with n nodes a partitioning with $p \in [1, n]$ partitions of the relationships $RS_{X-Y} = UD$ and/or $RS_{X-Y} = UC$ can always be found. The proof can be done as follows: Each node gets a help value called level. If a node is unconnected, its level-value will be set to '0'. Otherwise, if a node has only output edges, its level gets the value '1'. For each other node n_i the level will be set to $MAX(n_i) + $'1', where $MAX(n_i)$ is the maximum level of all nodes which have output-edges to this node n_i. All nodes of one level will be united into one partition. After that the required number of partitions p can be achieved by dividing partitions or by building the combination of two partitions if their nodes have monotonically increasing level-values.

Since this is only valid for an acyclic graph, the conditions for a conversion of a cyclic into an acyclic graph will be derived. The analyzer marks each node that is an element of a cycle in a way that it is possible to find all nodes building one complete cycle. Hence it is also possible to calculate the number of nodes *n'* which will build up the graph if one complete feedback is replaced by one union node. Depending on the size of the feedbacks a conversion of a cyclic into an acyclic graph may be useful - a big global feedback prevents this conversion because then a partitioning into *N* partitions would not be effective or might even be impossible.

Now it should be assumed that a given graph with *n* nodes has no cycle. For this, a simple partitioning algorithm which creates only partitions with the relationships $RS_{X\text{-}Y} = UD$ and/or $RS_{X\text{-}Y} = UC$ will be introduced. Equal to the last proof, this algorithm works with the temporary level value. After assignment of the level values, all *N* start partitions will be built. By this each partition gets *m'* elements with $m' \in [m - 1, m + 1]$ and

$$m = INT\left(\frac{n}{N}\right) \qquad - (3)$$

Also the procedure of this proof can be used in practice for creating partitions for an acyclic graph. The selection of nodes for a partition will be done by consideration of the level-value. For this, each level-value ℓ inside the partitions P_i and P_{i+1} must meet the formulas (4.1, 4.2).

$$\ell(n \in P_i) \leq \ell(n \in P_{i+1}) \qquad - (4.1)$$
$$\ell, \ell+1 \in P_i \Rightarrow \ell \notin P_{i+1} \qquad - (4.2)$$

Due to the definition of the level value ℓ all partitions of the graph are of type $RS_{X\text{-}Y} = UD$ and $RS_{X\text{-}Y} = UC$. Since this algorithm takes no notice of the desirable minimization of *Communication Costs*, this partitioning must be revised. This will be done by moving single nodes from one partition into a second partition. The kind of moving nodes must be in a way that no connection of type *BD* or *FB* will be created. For describing the conditions a definition of the partition-stage s is required that is only defined for partitions of type *UD* and/or *UC*. A partition that has no external input edges gets the stage-value '1'. A partition that is unconnected in both directions gets the stage-value '0'. The stage-value of every other partition P_i will be set to $MAX(P_i)$ + '1', where $MAX(P_i)$ is the maximum stage-value of all partitions which have external output edges to this partition. Now it is possible to express the conditions for a valid move of a node by introducing the following terms.

a) *n* is the node looked at and is an element of the partition P_i.
b) $s(P_i)$ is the stage-value of P_i.
c) min_{out} is the minimum stage-value of all partitions which can be reached directly by walking from *n* via its output-edges. If *n* has no output-edge, min_{out} gets the maximum stage-value of all partitions.
d) max_{in} is the maximum stage-value of all partitions which can be reached directly by walking from *n* via its input-edges. If *n* has no input-edge max_{in} gets the value '0'.

Movements: The node *n* can be moved into all partitions P_j if their stage-values meet the condition (5).

$$\max_{in} \leq S(P_j) \leq \min_{out} \qquad - (5)$$

The negation of this sentence is not true. A movement can be allowed without condition (5) being satisfied. This occurs if two or more partitions have the same stage-values as it is shown in the following example. In addition, it should be noticed that the move of one node from a partition P_i into a partition P_j can change the stage-values of all partitions.

Taking the *Communication Costs* and the *Simulation Loads* into consideration, a movement can be assessed. For the calculation of the movements an adaption of the algorithm described in [2] turned out to be suited.

As already described, it is not always possible to project a circuit to an acyclic graph. This becomes true if a complete feedback of the circuit is greater than the maximum size for the greatest partition. In such a case, the *Levelizing*-algorithm cannot be used. However, there are many of algorithms for partitioning cyclic graphs by consideration to various cost functions . An evaluation of all these algorithm would require another paper. Therefore, it should only be noticed that we choose the *MIN-CUT* algorithm described in [4].

In this case, the task of this *MIN-CUT* algorithm is not to partition the whole cyclic graph. Only the cyclic part of the graph will be partitioned by this algorithm. The acyclic part will be partitioned by the *Levelizing* algorithm (figure 1) as before. This way, a partitioning can be achieved of which types *UC* and/or *UD* are preferred.

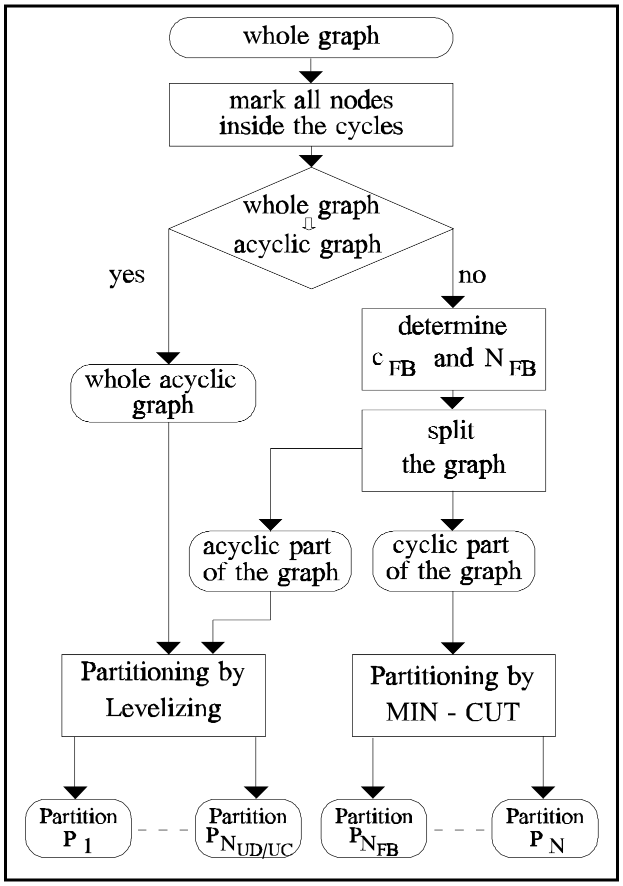

Figure 1: Global partitioning algorithm

The procedure is as follows: The analyzer checks whether the graph can be converted into an acyclic graph. If possible, the whole graph will be partitioned by the Levelizing algorithm. Otherwise the analyzer determines the two values N_{FB} and c_{FB},. N_{BF} is the number of partitions for the cyclic part of the graph; c_{BF} is the weight factor for a node in the acyclic part of the graph. It is

$$c_{FB} \geq c_{UD/UC} \qquad - \; (6.1)$$
$$N_{FB} = f(c_{FB}) = N - N_{UD/UC} \qquad - \; (6.2)$$

$c_{UD/UC}$	:=	weight factor for a node inside the acyclic part of the graph
$N_{UD/UC}$	:=	Number of partitions for the acyclic part of the graph

With help of the values in (6.1) and (6.2) it is possible to fit the sizes of the partitions to their relationship. The intention of this will be shown in the following chapter.

4. Simulation control

This chapter describes one possible means of a distributed simulation that works without the direct use of a global simulation time. For this, each VHDL simulator has its own local shell. These local shells control their own simulator and administer their own communication. The central shell is only used for the main functions such as stopping and starting and other basic procedures.

The communication between a local shell and its simulator depends on the available interfaces of the simulation unit. The simulator used in this project provides the *Styx* interface. Through this interface the user can build functions which are written in the computer language 'C' into his VHDL code. For example it is possible to link a PROCESS written in 'C' into the VHDL code. By listing each partition's output signals in the sensitivity list of this process, the shell will be informed of all events at the partition's outputs. The communication between the different local shells should be done by sockets since they are available on every UNIX machine.

This type of distributed simulation does not use direct global simulation time, so the data availability must be guaranteed by the local shell itself. Otherwise the simulators will work with wrong signal values and this leads to wrong simulation results. In particular, cyclically connected partitions (i.e. $RS_{X\text{-}Y}$ = *BD* and $RS_{X\text{-}Y}$ = *FB*) cause problems for the simulation control. For example, the following partitioning ought be used.

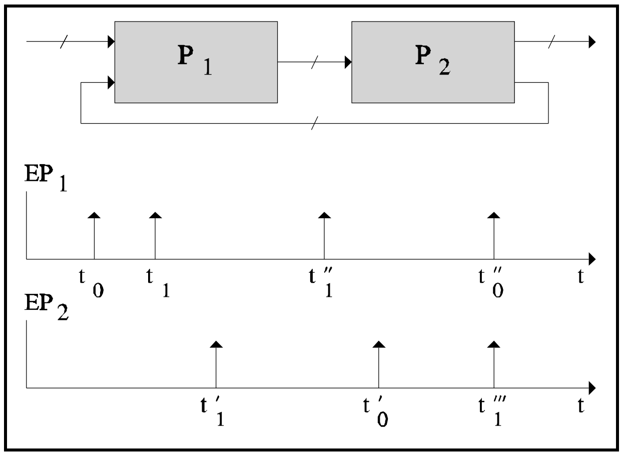

Figure 2: Cyclic connected partitions

EP_1	:=	Input-Event for partition P_1
EP_2	:=	Input-Event for partition P_2

Initially, only the input events at the times t_0 and t_1 are known. The first event causes at t_0' an event at partition P_2 and at t_0'' an event at P_1 once more. This event is not known before the simulation unit for P_2 has reached the time t_0''. Due to different path lengths inside the partitions and the external feedback, it is impossible to predict the partition and the time of the next event. Hence, all simulators must run parallel with respect to the simulation time. Updates of all cut signals for each simulation cycle is required. This means that a high communication rate between all simulators and therefore, between all participated processors. Frequently, the result will be a speedup of less than one.

This can be avoided by using an algorithm that works with a *look-ahead* [3]. The principle is that the simulators store the simulation state after a determined time into a temporary data file. The simulation will create provisional values. If a simulator detects that these values are wrong, it will decipher the last valid stored state, set the correct signal values and reset its simulation time. Also, the simulator must cancel all sent messages which were wrong. This can be done by sending and receiving *anti-messages*. Hence, this algorithm is suited for simulators that can be reset quickly. In addition to this, the transport time for messages should be very small.

Most of the partitioning algorithm create partition type *BD* resulting in a situation as shown in example figure 2. The partitioning, as accounted in the last chapter, tries to avoid partitions with the relationships *BD* and *FB*. In the absence of the above mentioned types, the only other relationships possible are *UD* and *UC*. Then, all simulators can run parallel without a synchronization for the time deltas or without using a *look-ahead* algorithm. This can be shown with the following example.

First, all input events for the system P_1 will be simulated. The local shell of this partition sends its output events to the local shells of partitions P_2 and P_3. To prevent a stepwise simulation, the shells accumulate the input events. When a number of events has reached the shell, the simulation is started. The risk of deadlocks will be avoided by sending request-messages. This occurs when a simulator waits longer for an input event than a predefined time *T*. Then a local shell sends a request-message to the shell concerned. This particular one sends back the information referencing the time which had just been simulated. If all partitions have nearly equal simulation loads, a maximum speedup will be achieved.

It is not always possible for the designer to build up a system without global feedback. In this case the procedure introduced in chapter 3 provides cyclically and acyclically connected partitions in which the priority is placed on the acyclic part. The simulation of the cyclic part will be done step for step. Opposite to this, the local shell of the acyclic connected partition starts its simulation unit blockwise. The result is that their processors can be used for other tasks between the single simulations. In addition to changing the values of the simulation loads (6.1, 6.2), the distributed simulation can be fitted into the computer environment. Given this, a simulation of a system with global feedback can be done with a noticeable speedup.

This paper introduced the main characteristics of one possible way for a distributed digital simulation. The apparent advantage is the absence of a theoretical upper speedup limit for this algorithm. It was shown how a fitted partitioning can increase the efficiency. This paper shows how the proposed methods can be applied without unnecessary exclusion of significant cases.

References

1. A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA, USA: Addison Wesley, 1974.
2. C. M. Fiduccia, R. M. Mattheyses, "A Liniear-Time Heuristic for Improving Network Partitions", Proc. *19th Design Automation Conference*, 1982, pp. 175-181.
3. D. R. Jefferson, "Virtual Time", *ACM Trans. Prog. Lang. Syst. 7*, no. 3, July 1985, pp. 404-425.
4. L. A. Sanchis, "Multiple-Way Newtwork Partitioning", *IEEE Transactions on Computers*, Vol. 38, No. 1, January 1989, pp. 62- 81.
5. L. Soule, T. Blank, "Statistics for Parallelism and Abstraction Level in Digital Simulation", *Proceedings of the 24th Design Automation Conference*, 1987, pp. 588-591.
6. Wong, Frankilin, "Statistics on Logic Simulation", *23rd Design Automation Conference*, ACM/IEEE July 1986, pp.13-19