# A Concurrency Preserving Partitioning Algorithm for Parallel Simulation of Hierarchical, Modular Discrete Event Models

Ki Hyung Kim
Department of Computer Engineering
Yeungnam University
214-1 Dae-dong Kyungsan-si, Kyungsangbook-do 712-749, Korea.
khkim@coregate.kaist.ac.kr

Tag Gon Kim and Kyu Ho Park
Department of Electrical Engineering
Korea Advanced Institutes of Science and Technology
373-1 Kusong-dong Yusong-gu, Taejon 305-701, Korea.
{tkim,kpark}@ee.kaist.ac.kr

## Abstract

*This paper presents a concurrency preserving partitioning algorithm for the optimistic parallel simulation of hierarchical, modular discrete event models. The proposed algorithm pursues the following three goals to achieve the overall objective of the minimum simulation time: (1) balance the computational loads of partitions, (2) maximize the parallel execution of independent models, and (3) minimize inter-processor communication. To estimate the parallelism inherent in models, the proposed algorithm utilizes the inherent hierarchical structural information of DEVS models. This paper describes how the proposed algorithm works through an example partitioning process.*

## 1 Introduction

Parallel and distributed simulation (PADS) has been an active research area for more than a decade. As identified by previous performance studies of PADS, the partitioning problem of models is one of the most important issues that may affect the performance of parallel simulation. Thus, finding the best possible partition of the model for obtaining the fastest simulation time has been the goal of many ongoing research efforts. Much research has been conducted for devising partitioning strategies for the partitioning problem[1]. Basically, the partitioning problem in parallel simulation belongs the class of NP-complete problems[1] if we want to utilize perfect knowledge about simulation, such as the sequence of events that have to be executed to simulate each process, the duration of each event, the precedence imposed by event messages, and an optimal schedule for execution of events. Looking at the problem practically, it is not possible to accurately predict such knowledge as the sequence and duration of run-time simulation events at compile time. In addition, there is no known polynomial algorithm to evaluate goodness of a partitioning even if sequence and duration of simulation events could be predicted based on compile-time information. Thus, polynomial time algorithms that find a partition which is near-optimal based on incomplete information at compile time would be the reasonable.

This paper proposes a new partitioning algorithm for parallel simulation of the models specified by the DEVS(Discrete EVent Specification) formalism[5]. The proposed algorithm is based on the simplifying assumption which adopts the following three goals to achieve the overall objective of the minimum simulation time: (1) balance the computational loads of partitions, (2) maximize the parallel execution of independent models, and (3) minimize inter-processor communication. To estimate the parallelism inherent in models, the algorithm utilizes the hierarchical structural information of models.

The remainder of this paper is as follows. Section 2 analyzes the partitioning problem of the optimistic parallel simulation in general. Then, in section 3, we derive the above mentioned three goals of the partitioning and propose the hierarchical partitioning algorithm. Finally, section 4 concludes this paper.



0-8186-7901-8/97 $10.00 

## 2 Partitioning Problem in Parallel Simulation

In this section, we analyze the characteristics of the partition problem in the parallel simulation. We assume that the underlying simulation algorithm is an optimistic one with the time window which is an interval in simulated time such that only events within the window are eligible for execution[2, 4].

A simulation task, $i$, performs one of the following activities during simulation.

1. True computation, $T_i$, which is the simulation cycle that will never be involved in roll-back process. True computation under optimistic parallel simulation consists of two parts : model execution time ($M_i$) and state saving time ($S_i$).

2. Local synchronization, $L_i$, which implies either one of the following three simulation cycles: (1) a simulation cycle that will eventually be rolled back (false computation, or $F_i$), (2) a rollback upon receipt of a straggler message ($R_i$), and (3) an idle cycle during which no work is performed ($I_i$).

3. Global synchronization, $G_i$, which manages global simulation progress (*e.g.*, GVT calculation, flow control, and memory management *etc.*).

Thus, the total execution time for processor $i$ under a partitioning $P$, $ET_{P_i}$, could be expressed as follows,

$$\forall i,\ ET_{P_i} = T_i + L_i + G_i \tag{1}$$
$$= M_i + S_i + F_i + R_i + I_i + G_i. \tag{2}$$

Note that

$$T_i + L_i + G_i = T_j + L_j + G_j, \quad \forall i, j. \tag{3}$$

Speedup is defined by the execution time of the sequential simulation divided by that of the parallel one. Sequential simulation does not perform any operations related to parallel simulation specific operations, such as state saving(S), local synchronization (L), and global synchronization (G). Thus, in the equation (2), the only operation performed in sequential simulation is the model execution time(M), which is unique regardless of simulation methodologies (parallel or sequential) and partitioning algorithms. The total execution time for sequential simulation, $ET_S$, can be expressed as follows,

$$ET_S = (\sum_{i=0}^{m-1} M_i) \tag{4}$$

, where $m$ is the number of processors(or partitions) used in parallel simulation. Then, from equation (2), (3), and (4), speedup can be represented as follows,

$$(\sum_{i=0}^{m-1} M_i)/(M_j + S_j + F_j + R_j + I_j + G_j), \quad \text{for any } j. \tag{5}$$

To achieve a maximum speedup, a partitioning algorithm should minimize each item in equation (2). In equation (2), $M_j$ is the only useful computation for simulation of models. Thus, we define the average model execution time, $M_{avg}$, as

$$M_{avg} = (\sum_{i=0}^{m-1} M_i)/m \tag{6}$$

, where $m$ is the number of processors.

The state saving time (S) and global synchronization time (G) in equation (2) are inevitable overheads for parallelizing the simulation. That is, since we assume that simulation algorithm is optimistic, the state saving should be performed to support the rollback operation, and global synchronization should also be performed to manage simulation progress in each processor. Thus, to maximize speedup, a partitioning algorithm should pursue the following goals in general.

1. Computational loads of processors (or model execution time $M_i$) should be balanced.

$$\max_{j=0:m-1} M_j \rightarrow M_{avg} \tag{7}$$

2. Local synchronization work should be minimized.

$$\forall i,\ L_i = F_i + R_i + I_i \rightarrow 0. \tag{8}$$

Thus, the maximum possible speedup can be expressed as follows,

$$(\sum_{i=0}^{m-1} M_i)/(M_{avg} + S_j + G_j), \text{ for any } j. \tag{9}$$

The partition which enables this maximum speedup is called the *optimal* partition.

Now, let's consider how to minimize $L_i$. Rollbacks occur by a number of reasons. Among them are differences in event processing times and event generation rates between processors, communication delays between processors, and the topology of the model (cyclic dependencies between models are particularly notorious). Ultimately, rollbacks occur because messages arrive out of chronological order at a processor. Thus, minimizing the difference between local simulation times of processors will reduce the frequency

of messages arriving out of chronological order. For this, a partitioning algorithm should pursue the following goals: (a) balance the computational loads of processors, (b) maximize the parallel execution of independent simulation tasks (that can do true computation on a processor at any time), and (c) minimize inter-processor communication. That is, if simulation tasks are independent with each other, there is no communication between them, and rollbacks do not occur($L_i$ becomes zero). Otherwise, simulation tasks communicate with each other, and this induces rollbacks or idle cycle ($I_i$) for waiting messages. The loads of each processor should be balanced to reduce the idle cycle of early ended processors. Moreover, the message communication delay may cause additional rollbacks and wasted lookahead computation; thus the above third goal should be pursued.

To maximize the parallel execution of independent simulation tasks, a partitioning algorithm should know the runtime behavior of the simulation, such as the influence relation between simulation cycles. However, to estimate such run time behavior at compile time is extremely difficult. Moreover, even though we know the runtime behavior of the model for one input data set, if we change input data set, the runtime behavior changes. Thus, to estimate such parallelism in models, we utilize the hierarchical structural information inherent in the hierarchical model design methodology of the DEVS formalism. In the methodology, when a system is modeled, a modeler naturally partitions the system into a set of subsystems while considering parallelism between them. The next section details this approach.

## 3 Partitioning Using the Hierarchical Structural Information

The basic strategy of the proposed algorithm is to insert more larger component in the same partition if possible (*i.e.* to partition a hierarchical composition tree at the higher level if possible). Following this basic strategy, the algorithm tries to find a partition which satisfies the other two goals.

For partitioning, the proposed algorithm transforms the simulation tasks into a weighted task tree which is defined as follows.

**Definition 1** *A task tree is a tuple $G = (V, E, C, T)$, where $V = \{n_j,\ j = 1 : v\}$, is the set of nodes and $v = |V|$, $E = \{e_{i,j} =< n_i, n_j >\}$ is the set of communication edges. The set $C$ is the set of edge communication costs and $T$ is the set of node costs.*

For describing task trees, the following notations are employed. If $e_{i,j}$ is in $E$, then $n_i$ is called the *parent* of $n_j$, and $n_j$ is a *child* of $n_i$. If there is a path from $n_i$ to $n_j$ and $n_i \neq n_j$, then $n_i$ is the *ancestor* of $n_j$ and $n_j$ is a *descendant* of $n_i$. A node with no descendants is called a *leaf*. The *subtree* of a node $n_i$ is a tree consisting of the descendants of $n_i$ including $n_i$ itself which are all mapped in the same partition. In that case, the node $n_i$ is called the *root* of the subtree. The *depth of a node* $n_j$ in a task tree is the length of the path from the root of the task tree to $n_j$.

The value $c_{i,j} \in C$ is the communication cost incurred along the edge $e_{i,j} \in E$, which is zero if both nodes are mapped in the same partition. The cost $\tau_i \in T$ of a node $n_i$ consists of two weights $< p_i, s_i >$, where $p_i$ is the computation cost of $n_i$ and $s_i$ is the sum cost of the subtree of $n_i$. The computation cost $p_i$ of a node $n_i$ in a task tree is the model execution time taken for all kinds of input messages during sequential simulation. That is, the cost does not include parallel simulation overheads $(L + G)$. The cost can be expressed in our previous notation as follows:

$$p_j = M_j \tag{10}$$

The computation cost of a node does not readily depend on the size of the node's associated model since some nodes would be more often executed and the cost should incorporate such information. Rather, it depends on the probability of receiving input events and the computational complexity for each input event.

The sum cost $s_i$ of a subtree $G_i$ is the sum of computation and communication costs in $G_i$, as shown in the following:

$$s_i = \sum_{n_k \in G_s} p_k + \sum_{n_k \in G_s} (c_{k,v} + c_{w,k}). \tag{11}$$

The meaning of node and edge costs can be better understood by comparing the following extreme cases. If there is full parallelism between partitions (that is, partitions are independent with each other), the sum cost of the root node of a subtree (or partition) is the execution time of the subtree. Thus, the total execution time $ET_P$ of the parallel simulation under a partition $P$ can be determined as follows,

$$ET_P = \max_{j=0:m-1} (s_j + S_j + G_j) \tag{12}$$

, where $S_j$ is the state saving time, and $G_j$ is the global synchronization time. That is, parallel simulation can remove most of parallel simulation overheads such as false computation time ($F_j$), rollback time ($R_j$), and idle time ($I_j$).

In contrast, if there is no parallelism between partitions at all (that is, partitions are fully dependent with each other), $ET_P$ is the sum of the execution time of each partition, as shown in the following,

$$ET_P = \sum_{j=0:m-1} (s_j + S_j + F_j + R_j + G_j). \tag{13}$$

That is, the execution time $ET_i$ of a partition $i$ is $s_i + S_i + F_i + R_i + G_i + I_i$ from the equation (2), where $I_i = ET_P -$

$(s_i + S_i + F_j + R_j + G_j)$. Therefore, these two extreme cases become the upper and lower bounds of the simulation completion time, respectively.

Now, we can represent the partitioning problem by using a task tree. That is, the partitioning is a mapping of the nodes of a task tree $G$ onto maximum $m$ partitions. More specifically, the problem is to determine a mapping

$$map(n_j) = K_i, \qquad j = 1 : v \text{ and } i = 0 : (M-1) \tag{14}$$

for the nodes($n'_j s$) of $G$ onto $M(\leq m)$ partitions $K_0, K_1, ..., K_{M-1}$, with the following objective function

$$min\ (\max_{i=0:M-1}\ (\sum_{n_j \in K_i, n_v, n_w \notin K_i} (p_j + (c_{j,v} + c_{w,j})))). \tag{15}$$

Note that our problem specifies just the maximum number of partitions, $m$. Thus, the resulting number of partitions, $M$, may be smaller than $m$. The final goal of partitioning is to minimize the total simulation time, not to fully utilize the given number of processors. Depending on the characteristics of simulation models, these two goals do not always match up. That is, in parallel simulation, using more processors for simulation does not imply smaller simulation time.

**Algorithm 1** HIPART($n_v$)

▷ *Initially,* $L_{avg} = (s_v/m + (\sum_i \sum_j c_{ij})/|E|)$, *where* $|E|$ *is the number of edges in* $E$.

▷ *Check if there are imminent children* $n'_w s$ *whose loads are greater than* $L_{avg}$.

1 **while** ( ∃ child $n_w$ and $s_w > L_{avg}(1+\alpha)$ ) **do**

▷ $n_w$ *is too big to be inserted into one partition and should be partitioned more.*

2 Hipart($n_w$);

3 **end while**

▷ *Now,* $n_v$ *has only those children whose loads are smaller than* $L_{avg}(1+\alpha)$

4 **while** ( $s_v > L_{avg}(1+\alpha)$ and number of partitions already made $< m$ ) **do**

5 Make a partition $P$ for satisfying the objective function $H = min\ |(2 \cdot c_{v,i} + \sum_{n_i \in P, n_i \in K} s_i) - L_{avg}|$, where $K$ is the set of children of $n_v$;

▷ *Update* $L_{avg}$.

6 $L_{avg} = \sum_{n_j \in P, n_k \notin P} (c_{j,k} + p_j)$;

7 After partitioning, update the computation and communication costs of the parents of $n_v$ including $n_v$ itself;

8 **end while**

**Figure 1. Hierarchical partitioning algorithm**

Figure 1 shows the proposed partitioning algorithm. Since the task graph is a tree, the algorithm is of a recursive form. The algorithm starts at the root node of a task tree. When the algorithm *enters* a node, it checks whether there are children whose sum cost is greater than the average cost of partitions($L_{avg}$). If there is such a child, the child is too big to be inserted in one partition and should be partitioned more. Thus, the algorithm enters the child to partition it. After this recursive process, the algorithm finds a node having children whose sum costs are all smaller than $L_{avg}$. In this case, we call that the algorithm *visits* the node. When the algorithm visits a node, it finds a partition with the minimum sum cost among the children by using some efficient heuristics such as Kernighan and Lin's algorithm[3].

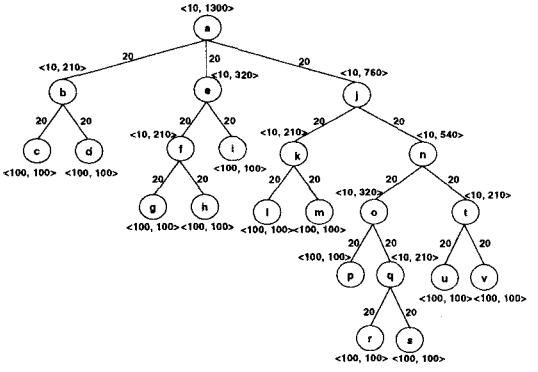


**Figure 2. An example task tree $T$**

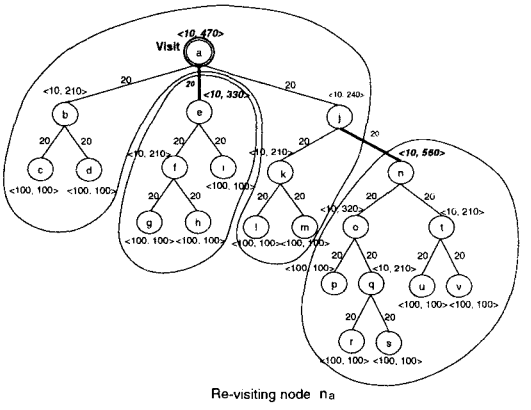


**Figure 3. The partition result for the task tree $T$**

For example, consider a task tree in Figure 2. We want to partition the task tree into three partitions. The algorithm *Hipart* partitions the task tree with the following average load of each partition while maximizing the parallelism of

the models.

$$L_{avg} = s_a/m = 1300/3 = 433.$$

Figure 3 shows the obtained partition result.

## 4 Conclusion

This paper has presented a novel partitioning algorithm for the parallel simulation of hierarchical, modular DEVS models. To maximize parallel execution of models, the proposed algorithm utilized the hierarchical structural information of models. We derived the partitioning algorithm through general analysis of the partitioning problem in the optimistic parallel simulation. Through a partitioning process of an example model, we described how the algorithm works.

## References


[1] P. Chawla. *Assignment Strategies for Parallel Discrete Event Simulation of Digital Systems.* PhD thesis, University of Cincinnati, 1994.

[2] R. M. Fujimoto. Optimistic approaches to parallel discrete event simulation. *Trans. of The Society for Computer Simulation*, 7(2):153–191, Oct. 1990.

[3] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. Journal*, 49:291–307, 1970.

[4] K. H. Kim. *Distributed Simulation Methodology Based on System Theoretic Formalism: An Asynchronous Approach.* PhD thesis, Korea Advanced Institute of Science and Technology, 1996.

[5] B. P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, 1984.