VLSI DESIGN
1999, Vol. 9, No. 3, pp. 253–270
Reprints available directly from the publisher
Photocopying permitted by license only

Concurrency Preserving Partitioning Algorithm for Parallel Logic Simulation

HONG K. KIM and JACK JEAN*

Department of Computer Science and Engineering, Wright State University, Dayton, Ohio 45435, USA

(Received 26 May 1998)

A partitioning algorithm for parallel discrete event gate-level logic simulations is proposed in this paper. Unlike most other partitioning algorithms, the proposed algorithm preserves computation concurrency by assigning to processors circuit gates that can be evaluated at about the same time. As a result, the improved concurrency preserving partitioning (iCPP) algorithm can provide better load balancing throughout the period of a parallel simulation. This is especially important when the algorithm is used together with a Time Warp simulation where a high degree of concurrency can lead to fewer rollbacks and better performance. The algorithm consists of three phases and three conflicting goals can be separately considered so to reduce computational complexity.

To evaluate the quality of partitioning algorithms in terms of preserving concurrency, a concurrency metric that requires neither sequential nor parallel simulation is proposed. A levelization technique is used in computing the metric so to determine gates which can be evaluated at about the same time. A parallel gate-level logic simulator is implemented on an INTEL Paragon and an IBM SP2 to evaluate the performance of the iCPP algorithm. The results are compared with several other partitioning algorithms to show that the iCPP algorithm does preserve concurrency pretty well and reasonable speedup may be achieved with the algorithm.

Keywords: Parallel logic simulation, partitioning algorithm, discrete event simulation, concurrency, load balancing, time warp

1. INTRODUCTION

Logic simulation is a primary tool for validation and analysis of digital circuits. To reduce the simulation time of large circuits, parallel logic simulation has attracted considerable interest in recent years [1]. As in many other parallel simulations, a good partitioning algorithm is a key to achieve good performance in parallel logic simulation, especially since the event granularity is relatively small compared to other types of simulations. Partitioning algorithms can usually

* Corresponding author. Tel.: 937-775-5106, Fax: 937-775-5133; e-mail: {hkim,jjean}@cs.wright.edu

be classified into two categories: static and dynamic. Static partitioning is performed prior to the execution of the simulation and the resulting partition is fixed during the simulation. A dynamic partitioning scheme attempts to keep system resources busy by migrating computation processes during the simulation. Since dynamic partitioning involves a lot of communication overhead [16], static partitioning is considered in this paper.

A good partitioning algorithm is expected to speed up parallel simulations. This may be achieved by focusing on three competing goals: to balance processor workload, to minimize interprocessor communication and synchronization, and to maximize concurrency. Among those goals, the maximization of concurrency is mostly overlooked by previous partitioning algorithms for logic simulation. Maximizing concurrency means partitioning a circuit such that at any time instance as many independent logic gates as possible are assigned to different processors. This can be achieved if workload is balanced among the processors all the time. However, most previous algorithms balance the accumulated amount of workload over the whole simulation period instead of the workload at any time instance.

The improved concurrency preserving partitioning (iCPP) algorithm proposed in this paper for parallel logic simulation takes the above three goals into consideration. It achieves a good compromise with a high degree of concurrency, a balanced workload, and reasonable amount of interprocessor communication. The compromise leads to a significant speedup obtained with a Time Warp parallel logic simulator implemented on an Intel Paragon and an IBM SP2. In Section 2, previous partitioning works in parallel logic simulation are summarized. Section 3 describes the rationale and the three phases of the proposed algorithm. A concurrency metric is proposed in Section 4 for the evaluation of partitioning algorithms. The algorithm is evaluated and compared to several other algorithms in Section 5. Section 6 concludes the paper.

2. PREVIOUS WORKS

A number of partitioning algorithms have been proposed for circuit simulations with slightly different emphasis. The iterative improvement algorithms perform a sequence of bi-partitioning. During each bi-partitioning, either a vertex exchange scheme or a one-way vertex moving scheme is used to iteratively reduce the interprocessor communication subject to some constraints on processor workload [7, 13]. In [23], Sanchis applied iterative improvements to multiple-way partitioning with a more frequent information updating after each vertex exchange or moving and obtained better results. Nandy and Loucks applied this iterative improvements to the initial random partitioning for the parallel logic simulation [22]. Sanchis's algorithm is adapted and used in the third phase of the algorithm proposed in this paper.

To achieve a compromise between interprocessor communication and processor load balancing, simulated annealing may be used for partitioning [3, 10]. However, methods based on simulated annealing are usually slow, especially when high quality solutions are to be found.

Partitioning algorithms such as the cone partitioning method or the Corolla method are based on clustering techniques [4, 20, 21, 24, 26]. These algorithms consist of two phases: a fine grained clustering phase and an assignment phase. In the first phase, clustering is performed to increase granularity. In the second phase, clusters are assigned to processors so to either reduce inter-processor communication or achieve load balancing.

A problem with most previous partitioning algorithms is that they do not produce a high degree of concurrency [5]. They usually try to balance the *accumulated* workload instead of trying to balance the *instantaneous* workload. This leads to performance degradation caused by rollbacks when a Time Warp parallel simulation is performed. Two exceptions are the random partitioning which has a good chance of producing

a high degree of concurrency and the string partitioning [17] which did take concurrency into account. However, either one tends to generate a large amount of interprocessor communication [24]. The proposed iCPP algorithm is designed to take concurrency into consideration and achieve a good compromise among the different competing goals.

3. PARTITIONING ALGORITHM

For partitioning purposes, a circuit may be represented as a directed graph, $G(V, E)$, where V is the set of nodes, each denoting a logic gate or a flip-flop, and $E \in (V \times V)$ is the set of directed edges between nodes in V. Three special subsets of V are I, O, and D, representing the set of primary input nodes, the set of primary output nodes, and the set of flip-flop nodes, respectively. Several assumptions are made in this paper regarding a circuit graph.

1. For each node, the *in-degree* and the *out-degree* are bounded by a constant. This assumption corresponds to the finite amount of fan-ins and fan-outs of logic gates in a circuit.
2. Associated with each node v_i, there is an *activity level*, a_i, that denotes the estimated number of events to be simulated for the node.
3. Associated with each edge e_i, there is an *edge weight*, w_i, that denotes the estimated number of events to be sent over the edge.

Note that both activity levels and edge weights may be obtained through circuit modeling or through pre-simulation [6]. They are assumed to be unity when neither modeling nor pre-simulation is attempted. Another point to mention is that the graph may contain cycles due to the existence of flip-flops.

The partitioning algorithm proposed in this paper assigns nodes in a directed graph to processors of an asynchronous multiprocessor. It consists of three phases so that conflicting goals may be considered separately and computational complexity may be reduced. For a graph with $|V|$ nodes these three phases are as follows.

Phase 1: Divide a graph into a set of disjointed subgraphs so that (1) each subgraph contains a primary input node (or a flip-flop) and a sizable amount of other nodes that can be reached from the primary input node (or the flip-flop) and (2) the *weighted* number of edges interconnecting different subgraphs is minimized. The computational complexity of this phase is a linear function of the total number of vertices.

Phase 2: Assign the subgraphs to processors in two steps. In the first step, the number of subgraphs is repeatedly cut into half by merging pairs of subgraphs until the number of subgraphs left is less than five times the number of processors. A heavy-edge matching is used during the merging process to find pairs of subgraphs with high inter-subgraph connectivity and each pair is merged into a single larger subgraph. In the second step, the remaining subgraphs are assigned to processors with considerations of both interprocessor communication and load balancing. This phase is different from the second phase of the original CPP algorithm and it uses the HEM (Heavy Edge Matching) technique in [11].

Phase 3: Apply iterative improvement to fine tune the partitions and reduce interprocessor communication. Iterative improvement is applied to two different levels of graph granularity. It is first applied to the graph that contains about $40 * N$ subgraphs, where N is the number of processors. At this level, a subgraph is moved as a whole from one processor to another and two sets of subgraphs may be exchanged to reduce interprocessor communication and to maintain good

load balancing. Afterwards, the iterative improvement is applied to the graph where each node is a logic gate or a flip-flop so that gates with heavy-weighted edge are assigned to the same processor and the workload of processors are balanced within some pre-specified bound.

In the first phase, primary input gates and flip-flops which are the starting points of the parallel simulation are assigned to different subgraphs so as to increase simulation concurrency. At the same time, the goal of minimizing interprocessor communication is considered. In the second phase, minimizing interprocessor communication is still the goal and a rough load balancing is to be achieved. In the last phase, the interprocessor communication is further reduced and the processor workload is better balanced.

The authors proposed the original CPP algorithm in [14] to produce a high degree of concurrency and well balanced partitions. The disadvantage of that algorithm is that the interprocessor communication overhead is still too high. Both the CPP algorithm and the new improved algorithm (so called iCPP) consist of three phases and they have the same first phase. The CPP algorithm produces very well balanced partitions with the emphasis on a linear time algorithm while the iCPP with its new second and third phases produces relatively fewer interprocessor communication. If the partitioning time is not critical, the iCPP produces better quality of partitions. Otherwise, the original CPP may be a better choice.

3.1. Phase 1: Subgraph Division

Concurrency in Primary Inputs and Flip-flops

In Figure 1, primary inputs and outputs are indicated for a graph where the majority of edges are assumed to go vertically from primary inputs to primary outputs. Also shown in the figure are three potential partitioning solutions: pipelining,

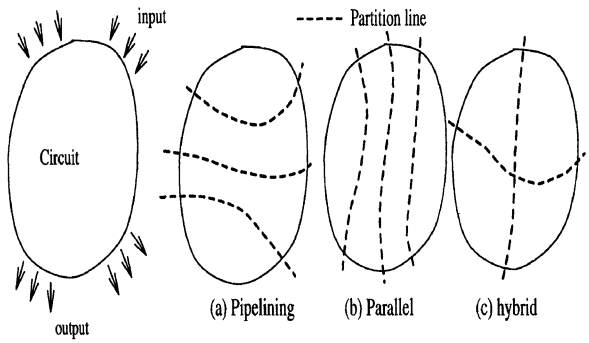

FIGURE 1 Partitions: (a) Pipelining, (b) Parallel, and (c) hybrid.

parallel, and hybrid. With the pipelining scheme, the processors work in a pipelined fashion where a processor cannot start until its predecessor (the one above it) has finished some tasks and sent an event over. Even though this scheme works very well for combinational circuits with large number of input vectors, this may cause erroneous computations and introduce a lot of rollbacks for sequential circuits in a Time Warp parallel simulation. The problem becomes even worse when the circuit size is increased, even for combinational circuits. On the contrary, the parallel scheme produces maximum concurrency at the beginning of execution. It reduces the differences between local virtual times (LVT's) and distances of rollbacks in optimistic simulations. This scheme is adopted in this phase in a way that preserves concurrency and minimizes interprocessor communication.

The first phase divides a graph into a set of disjointed subgraphs, each subgraph containing a primary input node (or a flip-flop) and a sizable amount of other nodes that can be reached from the primary input node (or a flip-flop). It is also desirable that the *weighted* number of edges interconnecting different subgraphs is minimized.

A greedy algorithm is used to achieve these two objectives. Basically, a node is assigned to a subgraph only after all its parent nodes have been assigned. Furthermore, only those subgraphs to which its parent nodes belong are considered in the node assignment.

A description of the detailed algorithm is shown in Figure 2 where an array Visited[·] is used to keep track of the sum of the number of parent nodes that have been assigned and the number of parent nodes that are driven by flip-flop outputs. Therefore a node can be assigned only after its Visited[·] value is the same as its in-degree. At that time, the node is assigned in a greedy way to minimize potential interprocessor communication. To assign a node to one of those subgraphs that its parent nodes belong to, the criterion used is to select the one with the *largest sum of edge weights* between the node and the subgraph. When there is a tie, the node is assigned to the subgraph that contains the parent node with the smallest rank. Here the rank of a node is the length of the shortest path from the root of the subgraph to which it belongs to the node. The tie-breaker is used so to somehow balance the sizes of subgraphs. A simple proof based on induction on node ranks can be used to show that all the nodes will eventually be assigned with the algorithm.

The algorithm used to traverse the graph and assign all the nodes is pretty efficient. The traversing method is called *data-dependency traversing* (DDT) *method.* Since each edge is visited only once and the number of edges per node is assumed to be bounded by a constant, the time complexity of this phase is of $O(|V|)$ where $|V|$) is the number of gates in a circuit. Because a node is not assigned until all its parent nodes have been assigned, the division into subgraphs is not

```
Phase1_Procedure(G, D, I)
 /* G is an input graph, D is the set of flip-flop gates, and I is the set of primary input gates.
    Initially, there are |D| + |I| subgraphs, each one containing either one flip-flop gate
    or one primary input node. At the end, there are still |D| + |I| subgraphs.  */
 for(each v in D ∪ I)
   assign_child_to_subgraphs_recursively(v);

procedure assign_child_to_subgraphs_recursively (v)
 /* The global variables Visited[·] are initially set to zero */
 If (v is not a primary input gate) and (v is not a flip-flop)
   then assign_to_subgraph (v);
 for(each child vertex w of v)
   Increase Visited[w] by 1
   if(Visited[w] is equal to in_degree[w] ) /* all parents have been assigned */
     then assign_child_to_subgraphs_recursively(w);
 endfor

procedure assign_to_subgraph (v)
 For each subgraph that the parent nodes of v belong to,
   compute the sum of edge weights between v and the subgraph.
 If (v is not a flip-flop)
   then Assign v to the subgraph that has the largest sum of edge weights. If there
        is a tie, assign to the one that leads to smaller rank for v.
   /* The rank of v is the length of the shortest path from the root of the subgraph to v.  */
 endif
 Calculate the rank to v from parents in the assigned subgraph.
 Update the connectivity matrix that shows the sum of edge weights between different subgraphs.
 /* The connectivity matrix is used in the second phase of the algorithm.  */
```

FIGURE 2 Phase 1: Divide a graph into disjoined subgraphs.

influenced by the ordering of primary inputs or flip-flops other than during a tie-breaking situation when there is a need to pseudo-randomly select a parent subgraph.

Several interesting differences between this phase and other algorithms are noted here. (1) Since redundant computations as used in cone partitioning [20, 21] are not allowed, a circuit is partitioned into disjointed subcircuits without overlapping. (2) The assignment of each node is based on parent nodes. This is different from other schemes that are based on children nodes [19, 24]. The advantage is in the higher concurrency that can be preserved in this scheme. (3) A node is assigned only after all its parent nodes are assigned and the *greedy* assignment is adopted. In some previous works, a node is assigned right after its first parent (child) is assigned and the node is assigned to the parent (child) without considering other parent (child) nodes [17, 19, 20, 21]. Such methods may lead to unbalanced global structures, namely, the first subgraph is significantly larger than the others.

In [24], primary input gates are evenly assigned to processors. Since subgraphs may be of very different sizes, such a scheme may produce an unbalanced assignment. With the second and third phases, our algorithm resolves the problem by allowing a subgraph to be assigned to multiple processors without compromising much about the concurrency and the interprocessor communication issues.

3.2. Phase 2: Assignment

In the previous phase an undirected graph with $|D| + |I|$ nodes, each representing a subgraph, is constructed from the original directed graph. In this phase, those $|D| + |I|$ nodes are assigned to N processors according to a connectivity matrix obtained in the first, phase and the size of each subgraph. Here a connectivity matrix is a $p \times p$ matrix, where $p(= |D| + |I|)$ is the number of graph nodes, and the matrix element at location (i,j), C_{ij}, denotes the sum of edge weights between subgraphs i and j.

This phase consists of two steps. In each step, a processor workload upper bound is set to be 105% of the average number of graph nodes assigned to a processor.

Step 1 The number of subgraphs is repeatedly cut into half by merging pairs of subgraphs until the number of subgraphs left is less than $5 * N$. During the merging process the HEM (Heavy Edge Matching) technique in [11] is used to group subgraphs into pairs where subgraphs with high edge weights are grouped together. The technique is greedy since, for each subgraph, it simply chooses the neighboring subgraph with the maximal edge weights to the subgraph. Therefore for a graph of $|E|$ edges between subgraphs, the technique's complexity is $O(|E|)$. Note that during the matching process the sum of the sizes of subgraphs in a pair is not allowed to be larger than the processor workload upper bound. The first partition when the number of subgraphs is below 40_*N is recorded and used in Phase 3.

Step 2 The remaining less than $5 * N$ subgraphs are assigned to N processors with considerations of both interprocessor communication and load balancing. First, the largest N subgraphs are each assigned to a processor. Then each of the remaining subgraphs is assigned one by one to individual processors. From among those processors whose workloads would not exceed the upper bound with the assignment, a subgraph is assigned to the one that has the heaviest edge to the target subgraph. If no processor can satisfy the upper bound with the assignment, a subgraph is assigned to the processor with the smallest workload.

3.3. Phase 3: Refinement

Iterative improvement is applied in this phase to fine tune the partitions and reduce interprocessor

communication. Iterative improvement is applied to two different levels of graph granularity. It is first applied to the graph that contains about $40 * N$ subgraphs, where N is the number of processors, and then to the graph where each node is a logic gate or a flip-flop.

Coarse Granularity Refinement

At this level of refinement, there are about 40_*N subgraphs. The objective is to reduce interprocessor communication and to improve load balancing by moving a subgraph as a whole from one processor to another or to exchange two sets of subgraphs. To achieve this objective, the well-known concept of *cut gain* is employed [7, 13]. That is, candidate subgraphs are selected based on the gain in edge cut by moving them or exchanging them. In addition, the moving or exchanging of subgraphs must satisfy a lower bound (= 95% of averaged load) and an upper bound (= 105% of averaged load) of processor workload.

Previous algorithms [7, 13, 23] that are based on *move-based* methods only allow one-way move or two-node exchange scheme. They do not allow *many to many* exchanges that are used in the iCPP algorithm. As a result, when two subgraphs cannot be exchanged without violating workload constraints, the subgraph with larger workload may probably be exchanged with two subgraphs with smaller workload. The algorithm to perform movements or exchanges is as follows:

1. *One-Way Movement*: For each node in the graph (with $40 * N$ nodes), check if it is possible to move the node to a different processor to get gain and satisfy the workload constraint. If the moving does not satisfy the workload constraint, put this node into a candidate list for the following many to many exchange.
2. *Exchange*: For each processor pair, say, processors i and j, there is a candidate list. Each candidate list is sorted according to the gain values.
   2.1. Exchanging two sets of subgraphs is considered. Initially two subgraphs, the first elements from each list, are compared and, if this exchange does not satisfy the workload constraint, additional subgraph is added from the processor with larger workload to satisfy the workload constraint.
   2.2. If it is impossible to find two sets of subgraphs, from among the two first candidates from the two lists, delete the one with larger workload and start Step 2.1 again. If there is no additional element, go to Step 3.
3. Repeat Steps 1 to 2 for a given amount of iterations or until no more gain is achieved.

Fine Granularity Refinement

At this level, the iterative improvement is applied to the graph where each node is a logic gate or a flip-flop so that gates with heavy-weighted edge are assigned to the same processor and the workload of processors are balanced within some prespecified bound.

Even though the iterative improvement is employed in Phase 3 to get better load balancing and to reduce the interprocessor communication, it usually produces better concurrency. This effect, not by design, is detected by using the concurrency metric as described in the next section.

4. CONCURRENCY METRIC

Since the iCPP algorithm is to preserve concurrency, how to measure concurrency is discussed in this section and a concurrency metric is developed. In [25], concurrency was defined as the number of active gates at a given simulation cycle for parallel discrete event logic simulation under the assumption that there is an infinite number of processors. In the same paper, the average parallelism is defined as the average concurrency during the entire simulation. A different definition of concurrency was used by Smith *et al.*, so to take into account the number of processors [24]. However,

both definitions can be used only for a synchronous simulation that performs at lock steps. In an asynchronous parallel simulation as in a Time Warp simulation, it is difficult to measure the concurrency because the "simulation cycle" is not clearly defined.

Most asynchronous parallel simulation studies use the performance of real parallel simulations to evaluate partitioning algorithms. Recently, a few works based on the critical path analysis have been performed to estimate the execution time of parallel simulations [9, 12, 18]. However, their approaches require the execution of either a sequential simulation or a parallel simulation. Since the performance of parallel simulation is affected not only by partitioning but also by many other factors such as synchronization algorithms and target architectures, it is desirable to have a concurrency metric for partitioning that is independent of those factors and requires neither parallel simulation nor sequential simulation. Such a metric for concurrency has never been defined previously.

Given a partitioned directed graph, we want to define a concurrency metric that can be evaluated without performing a parallel simulation or a sequential simulation. Since concurrency is related to the number of active gates at a simulation cycle, nodes in a partitioned directed graph need to be classified into linearly ordered levels so that each level can be evaluated at about the same time on all processors. This classification process, called *levelization*, is based on the distances of individual graph nodes from primary input gates or flip-flops and the processor assignment. Once the levelization procedure has been applied to a directed graph, the concurrency can then be defined.

4.1. Concurrency Levelization

A simple levelization procedure may simply rank graph nodes based on their ranks, where the rank of a graph node is defined as the maximum distance from all primary input gates or flip-flops that can reach the node. Since two nodes of different ranks may be executed at the same time during an asynchronous parallel simulation, such a simple levelization technique may not work very well in terms of measuring concurrency.

A better levelization procedure should allow graph nodes of different ranks to be assigned to the same level as long as the workload in each level is (roughly) balanced across different processors. Here we propose such a better *concurrency levelization* procedure which consists of three phases.

- In the first phase, the lower bounds for the levels of individual graph nodes are determined based on data dependency by traversing through the graph.
- In the second phase, the upper bounds for the levels of individual graph nodes are determined by *reversely* traversing through the graph.
- In the last phase, specific level is assigned to each graph node, starting from the first level, so that the workload in each level is as much balanced as possible across different processors. In this way, the concurrency is expected to be more closely related to what happens during an asynchronous parallel simulation.

The three-phase procedure is similar to the commonly used graph scheduling based on critical path. The only difference is that, in our case, when a parent node and its child node are assinged to the same processor, they are allowed to be assigned to the same level.

Phase 1: Determine Lower Bounds

Four principles are adopted in our levelization procedure to determine the lower bounds.

1. All primary input nodes and flip-flops are assigned to the first level. Namely, the lower bound level and upper bound level are zero.
2. All child nodes with the same set of parents are assigned to the same level. This is based on an assumption that all child nodes are executed at about the same time.

3. Two connected nodes are assigned to different levels if their dependency is inter-processor dependency. Otherwise, they are assigned to the same level. (If two nodes are assigned to the same processor, their dependency is called intra-processor dependency. Otherwise, it is called inter-processor dependency.) Inter-processor dependency is the main reason for processors to block and is, therefore, considered to be important in evaluating concurrency.
4. The number of levels should be minimized if possible.

The algorithm to determine lower bounds starts from the primary input gates and ffip-flops by assigning 0 for their bounds and then the lower bound of each node t can be determined from the lower bounds of its parent nodes as

$$\text{low}(t) = \begin{cases} 0 & \text{if } t \text{ has no parent node,} \\ \max_{s \in S} f(s) & \text{otherwise} \end{cases} \tag{1}$$

where S is the set of parent nodes of t and

$$f(s) = \begin{cases} \text{low}(s) & \text{if } s \text{ and all its child nodes are assigned to the same processor} \\ \text{low}(s)+1 & \text{otherwise} \end{cases}$$

The computation of low(t) for all graph nodes adopts the data-dependency traversing (DDT) method that is used in the first phase of the iCPP.

In Figure 3, the algorithm is applied to two different partitions of the same graph. Each node has two numbers: the node number (inside each circle) and its level lower bound (outside each circle). In Figure 3(b), nodes 1 and 5 can be executed at about the same time because there is no interprocessor dependency. (The meaning of $T(1)$, $T(2)$, and $C(2)$ in the figure will become clear later after the concurrency metric is introduced.)

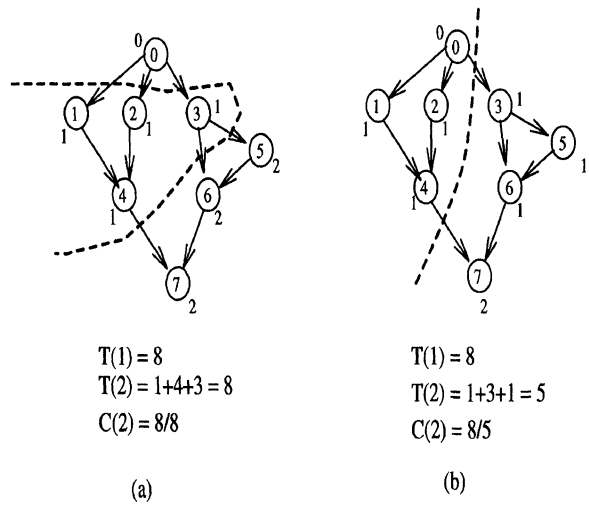

FIGURE 3 Concurrency levelization and concurrency metric.

Phase 2: Determine Upper Bounds

Upper bound levels can be similarly determined by using the following three principles.

1. All primary output nodes and parent nodes of all flip-flops are assigned to be $M (= \max_t \text{low}(t))$.
2. All parent nodes with the same set of child nodes are assigned to the same level.
3. Two connected nodes are assigned to different levels if their dependency is inter-processor dependency. Otherwise, they are assigned to the same level.

The algorithm to determine upper bounds starts from the primary output gates and parent nodes of flip-flops by assigning M for their bounds and then the upper bound of each node t can be determined from the bounds of its child nodes as

$$\text{high}(t) = \begin{cases} M & \text{if } t \text{ has no child node or has only flip-flops as child nodes,} \\ \min_{s \in S} f(s) & \text{otherwise} \end{cases} \tag{2}$$

where S is the set of child nodes of t and

$$f(s) = \begin{cases} \text{high}(s) \text{ if } s & \text{and all its parent nodes are assigned to the same processor} \\ \text{high}(s) - 1 & \text{otherwise} \end{cases}$$

Phase 3: Assign Specific Levels

After the bounds are set up for each gate, the level of each non-critical node needs to be assigned. Here a node is a *non-critical node* if its level lower bound is different from its upper bound. Otherwise it is *critical* and there is no flexibility in assigning its level.

The assignment starts level by level from level 0, which is the first level, to level M. At each level, the numbers of critical nodes in individual processors are counted and their maximum value is set as the maximum workload at that level. For processors other than the one with the maximum workload at the specific level, some of their non-critical nodes are assigned to the same level so to balance the workload at that level as much as possible. The following is the level assignment procedure.

1. Initialize w_{ijk} to be the number of nodes in processor i that have j as their lower bound and k as their upper bound, where $1 \le i \le N$ and $1 \le j \le k \le M$.
2. $t \leftarrow 0$.
3. For each level t, the maximum workload is calculated among all processors counting only critical nodes. That is, the maximum workload is computed as $\max_t = \max_{i=1}^{N} w_{itt}$. Furthermore, W_{it}, the workload of processor i with level t, is computed as $\min\left(\sum_{k=t}^{M} w_{itk}, \max_t\right)$.
4. For each processor i, critical nodes in the level t are assigned and, if needed, additional non-critical nodes with level t as a lower bound are assigned to the same level until the total workload, $\max_t$, is assigned or no more noncritical nodes are available at this level. During the assignment, each remaining workload, w_{itk}, is updated for $t+1 \le k \le M$. If there are remaining nodes with the level t as a lower bound in each processor, these nodes are assigned at the next level $t+1$ as nodes with level $t+1$ as a lower bound. Namely, the workloads at the next level are updated by $w_{i,t+1,k} = w_{i,t+1,k} + w_{itk}$ by adding the remaining nodes at the level t to the next level for $t < k \le M$.
5. $t \leftarrow t+1$.
6. Steps 3 to 5 are repeated while $t \le M$.

4.2. Concurrency Metric

Based on several assumptions as described below, the concurrency will be defined based on the levelized graph without performing either sequential or parallel simulation. The average concurrency will be derived and used as the concurrency metric in this work.

Assumptions for Concurrency Metric

(1) each gate is active exactly once during the simulation time and the evaluation of each gate takes one time unit,
(2) there is no communication delay between processors,
(3) each scheduler uses a breath-first selection method for the next node, (Note that a real parallel simulation uses the smallest timestamp first scheduling.)
(4) only nodes with the same level can be executed concurrently.

DEFINITIONS The concurrency at time t, denoted as C_t, is defined as the number of active gates at that time and the *average concurrency* with N processors, $C(N)$, is defined as $\left(\sum_{t=1}^{T(N)} C_t\right)/T(N)$, where $T(N)$ is the total amount of simulation time.

The numerator of the average concurrency, $\sum_{t=1}^{T} C_t$, is equal to $|V|$, the total number of gates in a circuit graph. With Assumption (1), it can also be viewed as the total workload or the total execution time in a sequential processor. As a result, it can also be labeled as $T(1)$.

As to the denominator of the average concurrency, $T(N)$, it is the execution time with N processors. With assumptions (2), (3), and (4), $T(N)$ is equal to the sum of maximum execution times at each level. Or

$$T(N) = \sum_{k=0}^{M} \max_{1 \le i \le N} W_{ik}.$$

where M is the maximum level in a levelized directed graph and W_{ik} the number of level-k nodes that are in processor i such that $1 \leq i \leq N$. (If we consider graphs with weighted nodes, *e.g.*, a task graph, W_{ik} should be the sum of weights of level-k nodes in partition i.)

Therefore the concurrency metric $C(N)$ can be computed as

$$C(N) = \frac{T(1)}{T(N)} = \frac{|V|}{\sum_{k=0}^{M} \max_{1 \leq i \leq N} W_{ik}}.$$

Since $|V| = \sum_{k=0}^{M} \sum_{i=1}^{N} W_{ik}$, it can be shown mathematically that $1 \leq C(N) \leq N$. If $C(N) = N$, which occurs when W_{ik} is independent of i, the partitioning is optimal from the point of concurrency. If $C(N) = 1$, which occurs when $\sum_{i=1}^{N} W_{ik} = \max_{1 \leq i \leq N} W_{ik}$ for all k, the partitioning causes sequential processing even though multiple processors are used.

In Figure 3, the levelization algorithm is applied to two different partitions of the same graph. It happens that all the nodes are critical in either partition and the node levels are indicated outside those nodes. For each partition, the concurrency metric is computed and illustrated in the figure. As shown in the figure, the partition in Figure 3(b) has higher concurrency metric value than the other one. This result is consistent with the intuition and is as expected.

5. SIMULATION RESULTS AND COMPARISON

5.1. Simulation Model and Environment

In this paper, the effects of the CPP algorithm on the Time Warp algorithm are studied. In the Time Warp algorithm [8], a processor executes events in a timestamp order and the local simulation time of a processor is called a local virtual time (LVT). Whenever a processor receives an event with a timestamp less than the LVT, it rolls back immediately to the state just before the timestamp of that event. Even though the Time Warp algorithm has a high degree of parallelism, the number of rollbacks must be reduced to get better performance.

The experiments were performed on an Intel Paragon XP/S machine and an IBM SP2 machine. The Paragon machine is a massively parallel processor that has a large number of computing nodes interconnected by a high-speed mesh network. Each node has two identical 50 MHz Intel i-860XP processors and 32 Mbytes of memory. One processor executes user and operating system codes while the other is dedicated to message passing. Each node runs the XP/S operation system using MPI communication environment. The IBM SP2 is also a massively parallel processor that has a large number of computing nodes interconnected by a high-speed multistage network. Each node of SP2 has a 75 MHz processor and 64 Mbytes of memory. Several of the largest sequential ISCAS benchmark circuits [2] were used as test cases. Their characteristics are summarized in Table I. One hundred random input vectors were used for the simulation of each circuit and the first ten of them were used in the pre-simulation to get estimates of node activity levels and edge weights. *Note that in this section the pre-simulation is used only where it is explicitly specified.* The gate-level unit delay model was adopted for the simulations. For each of those ISCAS circuits, the CPP algorithm finished the partitioning part of simulation within 5 seconds on a sequential DEC alpha 3000/400 workstation.

TABLE I ISCAS benchmark circuits

No. of Circuit	No. of gates	No. of input gates	No. of output gates	No. of flip-flops
c6288	3376	32	32	0
c7552	4951	206	108	0
s9234	5808	36	39	211
s13207.1	8651	62	152	638
s15850.1	10383	77	150	534
s38417	23843	28	106	1636
s38584.1	20717	38	304	1426

5.2. Performance Comparison

To facilitate comparison, three other partitioning algorithms, including the *random* partitioning, the *depth first search* (DFS) partitioning, and the *string* partitioning, were implemented and the MeTis package that supported the *multilevel* partitioning [11] was used.

1. The random partitioning scheme randomly assigns circuit gates into processors with the constraint that each processor is allocated roughly the same number of circuit gates. Even though the random partitioning scheme introduces a lot of communication overhead, it is expected to provide pretty good load balancing [16, 24].
2. The DFS partitioning scheme uses a basic depth first search to traverse a circuit graph starting from a pseudo root that has one directed edge to each primary input gate. The traversing is therefore guaranteed to visit each gate once. The sequence of gates is then assigned to the processors, with each processor getting roughly equal number of gates [10, 15]. This scheme generates partitions with relatively low interprocessor communication and achieves balanced processor workloads. However, the DFS scheme does not consider concurrency issue, unlike the CPP algorithm. Note that both node activity levels and edge weights were not considered in either random or DFS scheme.
3. The *string* partitioning algorithm [17] is the first algorithm to consider concurrency and it produces solutions of high degree of concurrency. However it generates a lot of interprocessor communication because communication between strings is not considered.
4. MeTis is a software package that implements a multilevel partitioning algorithm for irregular graphs. It is known to produce high quality solutions for different applications, especially in reducing interprocessor communication [11].

The CPP algorithm was originally compared to the random and the DFS algorithms on an Intel Paragon machine. Because that machine was phased out right before the development of the iCPP algorithm, the simulation results of the iCPP algorithm, along with the string algorithm and the MeTis were from an IBM SP2 machine. That is why in this section not all algorithms are compared together for parallel simulation results. When parallel simulations are not required for a performance metric, such as the concurrency metric, all those algorithms are compared at the same time.

The algorithms were evaluated by comparing the following performance measures: concurrency, load balancing, the ratio of edge cut, the ratio of external events, the ratio of erroneous events, the execution time, and the speedup. Here the ratio of edge cut is defined as the number of cut-edges (*i.e.*, edges between nodes on different processors) divided by the total number of edges in a circuit. The ratio of external events is defined as the number of external events divided by the total number of external and internal events. These two performance measures are related to interprocessor communication. Note that a cut-edge does not necessarily produce an external event while an external event does imply the existence of a cut-edge.

Concurrency

The inherent parallelism of a circuit depends on its structure, such as the number of primary input gates, the number of flip-flops, and the connectivity of circuit gates. In general, a loosely connected circuit has high concurrency while a tightly connected circuit has relatively low concurrency. Table II summarizes the concurrency of several circuits after the application of those partitioning algorithms. Figure 4 displays those results graphically for the circuit s38417 with three partitioning algorithms that produce a high degree of concurrency. It shows that the iCPP algorithm produces better concurrency than MeTis does while the string algorithm is the best in terms of concurrency.

TABLE II Concurrency metric

Circuit	Experiment Partitioning	No. of Processors 2	4	5	10	16	20	32	40	52	64
	String	2.0	3.9	5.0	9.7	15.2	19.0	28.8	35.2	41.9	48.4
s38417	DFS	2.0	4.0	4.8	7.7	10.5	11.3	15.5	16.3	20.1	22.2
	MeTis	2.0	4.0	5.0	9.9	15.0	17.8	21.8	20.6	26.0	27.3
	iCPP	1.9	3.9	5.0	9.8	15.0	16.6	22.2	22.0	28.8	28.3
s38584.1	iCPP	1.9	3.9	4.9	6.8	8.0	9.5	12.1	15.1	17.1	19.5
s15850.1	iCPP	1.9	3.9	4.9	7.1	8.6	9.1	12.1	12.8	12.8	13.4

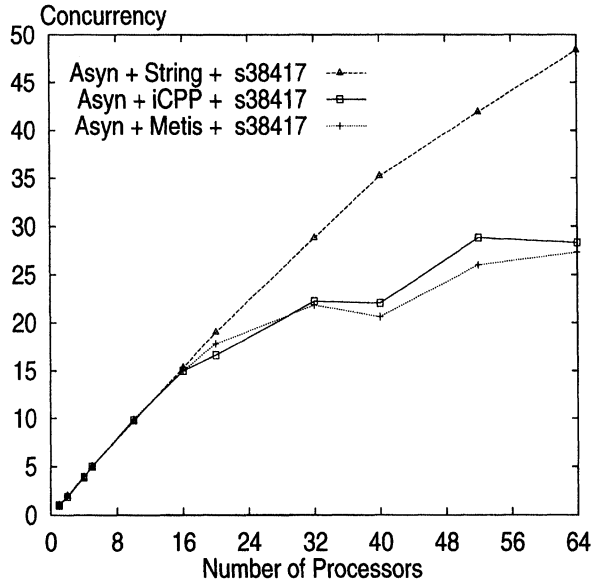

FIGURE 4 Concurrency.

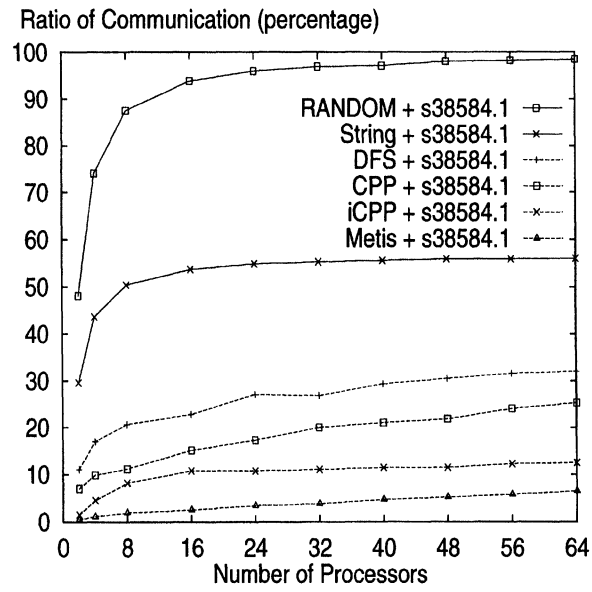

FIGURE 5 Edge cut ratio.

Interprocessor Communication

In Figure 5, partitioning results of the s38584.1 circuit are summarized in terms of the ratio of edge cut. As can be seen from the diagram, curves for different partitioning algorithms exhibit the same trend as the number of processors increases. Another observation is that the iCPP and the MeTis algorithms consistently produce lower ratio of edge cuts than those other four algorithms. As a matter of fact, when bi-partitioning (two-processor partitioning) is performed, the ratio of external events for the iCPP algorithm is 1.5%. This compares favorably with several other partitioning algorithms such as Flip-flop-Clustering (10%), Min-Cut (15%), and Levelizing (20%). These results were reported in [26]. Even though the Corolla-clustering algorithm [26] with its 0.6% ratio of external events for bi-partitioning definitely outperforms the iCPP algorithm in this regard, it is relatively slow and does not take concurrency into consideration. The figure also shows that the ratio of edge cuts with the iCPP algorithm is only 12.5% even when 64 processors are used. It is therefore interesting to check out the other limiting factor in Time Warp simulation, *i.e.*, the rollback overhead.

Event Rollback

The ratios of erroneous events versus the number of processors are summarized in Figure 6 for the

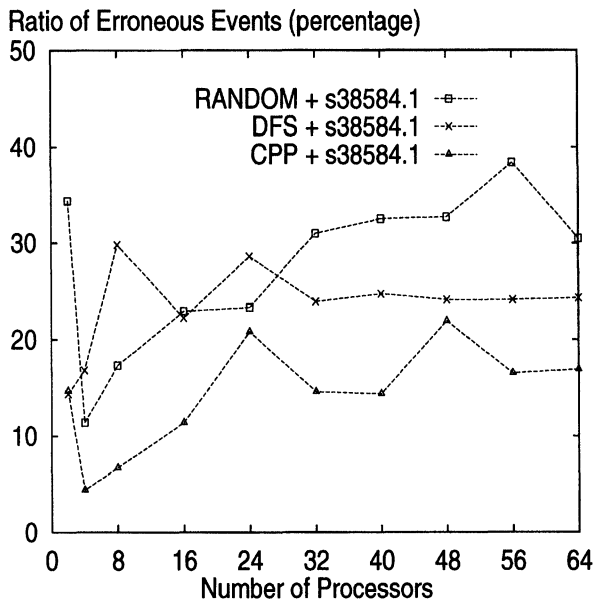

FIGURE 6 Ratio of erroneous events.

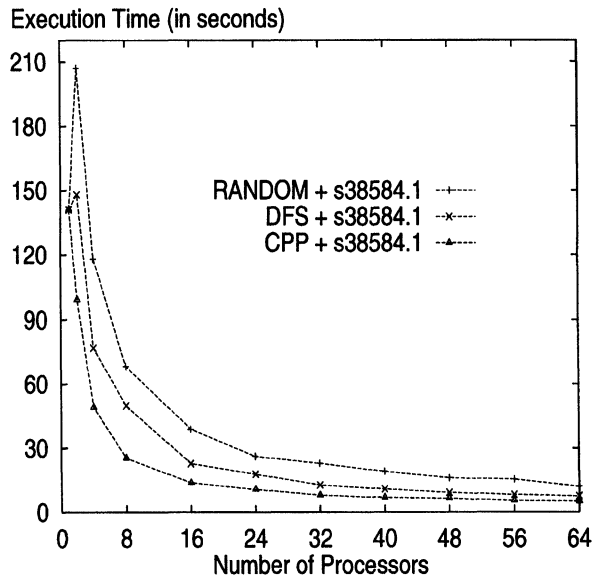

FIGURE 7 Execution time.

three partitioning schemes. The circuit is the s38584.1 circuit that has 20717 gates. To read the diagram, see for example, less than 18 percent of events were canceled when the CPP algorithm was used for 64 processors. Apparently the CPP algorithm introduces lower ratios of erroneous events than those of the DFS and the RANDOM schemes.

Simulation Execution Time

Figure 7 shows the execution times for simulating the s38584.1 circuit with different algorithms. Here the execution time excludes the time for partitioning, reading input vectors, and printing output vectors. Note that, when the number of processors is fixed, experiments show that the execution time increases linearly with the number of randomly generated input vectors. For real circuit simulations, the circuit is expected to be larger and much higher number of input vectors are required. It therefore makes sense to exclude the time to perform partitioning since it is performed only once as an initialization step.

Table III lists the values used in Figure 7 as well as the execution times for simulating two other circuits. It shows that the parallel simulation of s38584.1 can be executed in 5 seconds using 64 processors (with node activities and edge weights obtained through pre-simulation) while the sequential simulation takes 140 seconds. It also shows that, when the same number of processors is used, the parallel simulation with the CPP algorithm is usually faster than with the other two algorithms for the s38584.1 circuit. In particular, when two processors are used, the RANDOM and DFS algorithms do not get any speedup because the RANDOM algorithm suffers from a lot of interprocessor communication while the DFS algorithm does not provide a high degree of load balancing over time, *i.e.*, it does not provide enough concurrency and therefore causes a lot of rollbacks.

Note that the pre-simulation speeds up parallel simulations in two ways. Edge weights representing event frequencies over edges can be used to reduce the interprocessor communication and the node activity level representing event frequencies

TABLE III The execution time (in seconds); CPP (= CPP without node activity), CP^3(= CPP with pre-simulation)

Circuit	Experiment Partitioning	No. of Processors 1	2	4	8	16	24	32	40	48	56	64
	RAN DOM	141.0	209.9	118.0	68.2	38.8	26.1	22.9	19.1	16.1	15.4	12.0
s38584.1	DFS	141.0	148.4	77.0	49.8	22.9	17.9	12.8	10.8	9.2	8.1	7.5
	CPP	141.0	117.9	55.9	27.6	14.5	10.7	9.4	10.9	9.5	7.6	6.6
	CP^3	141.0	99.2	49.1	25.5	13.8	9.7	7.9	6.8	6.4	5.3	5.0
s38417	CP^3	103.0	67.9	32.4	15.2	8.2	6.3	4.8	4.7	4.2	3.2	3.3
s15850.1	CP^3	42.5	28.4	14.1	8.3	5.8	5.0	4.3	4.0	3.9	3.2	3.4

inside a node can be used to get better load balancing.

Load Balancing

Figure 8 shows the processor workloads obtained with parallel simulations when the CPP algorithm is used with or without the pre-simulation. Each point in the figure denotes the total number of events evaluated for a specific processor. Apparently better load balancing is achieved when the node activity levels and edge weights are available in Phase 3 of the CPP algorithm. Note that unbalanced workload increases the number of erroneous events, increases the amount of rollbacks, and prolongs a parallel simulation.

Speedup

Speedup is defined as the ratio of the execution time of the best sequential simulator to that of the parallel simulator. For most circuits, reasonable speedups may be obtained when the number of processors is small and there are enough events to be executed concurrently. In Figure 9, speedups obtained with different algorithms are shown. The figure shows that the performance of a Time Warp simulation is very sensitive to partitioning when large number of processors are employed and the CPP algorithm achieves better speedup than those of the RANDOM and the DFS partitioning algorithms. Note that when the number of processors is near 40, the CPP algorithm without pre-simulation degrades and performs roughly the

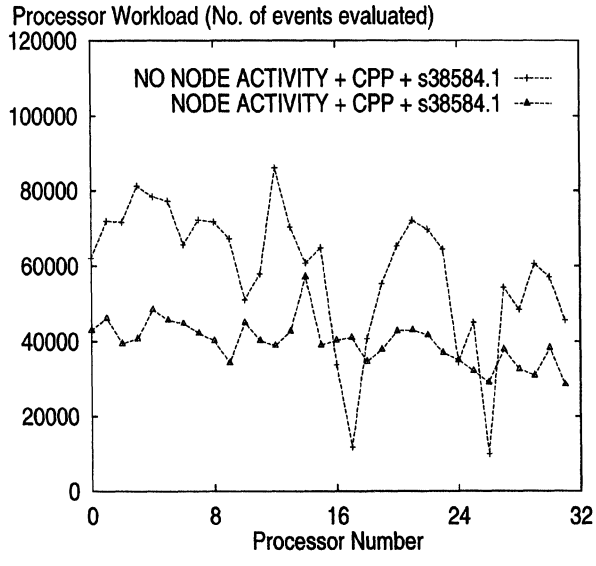

FIGURE 8 Processor workload with 32 processors.

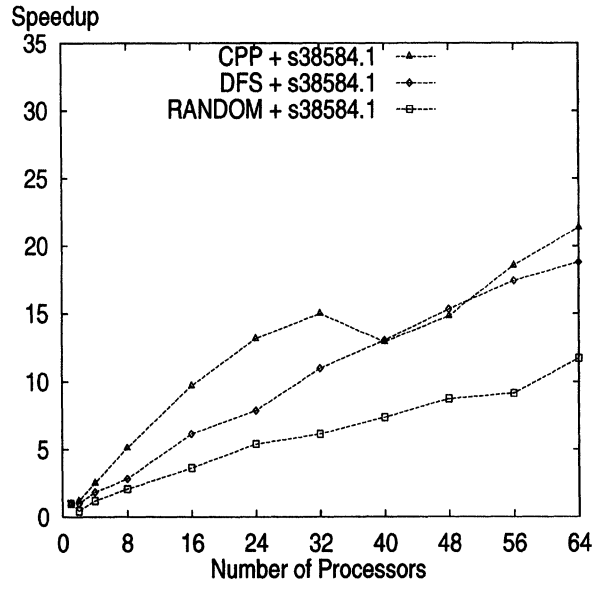

FIGURE 9 Speedups without pre-simulation.

same as DFS does. By comparing this phenomenon to the 538584.1 curve in Figure 10 where the speedups of three large ISCAS89 circuits are displayed for the CPP with pre-simulation, another observation is that, when the number of processors is below 32, the performance of the CPP algorithm is almost not influenced by the decision to consider node activity levels and edge weights. However, when more processors are added, the advantage of having extra knowledge from the pre-simulation becomes critical. Also can be observed from Figure 10 is that a significant speedup of 31 with 64 processors was obtained for the s38417 circuit.

Performance on the IBM SP2

Among the algorithms considered, three of them that are more competitive are tested on an IBM SP2 machine. These include the proposed iCPP algorithm, the string algorithm, and the MeTis. All simulations are performed with 1000 input vectors using the Time Warp parallel simulation model and is globally synchronized every 100 input vectors. Figure 11 shows the resulting speedups

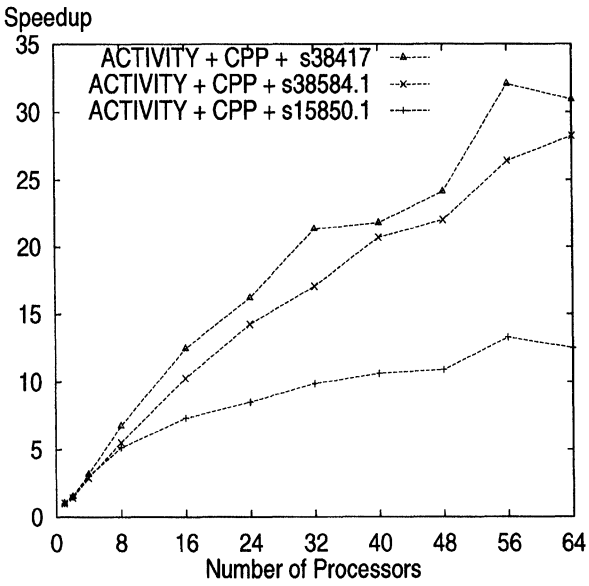

FIGURE 10 Speedups with pre-simulation for different circuits.

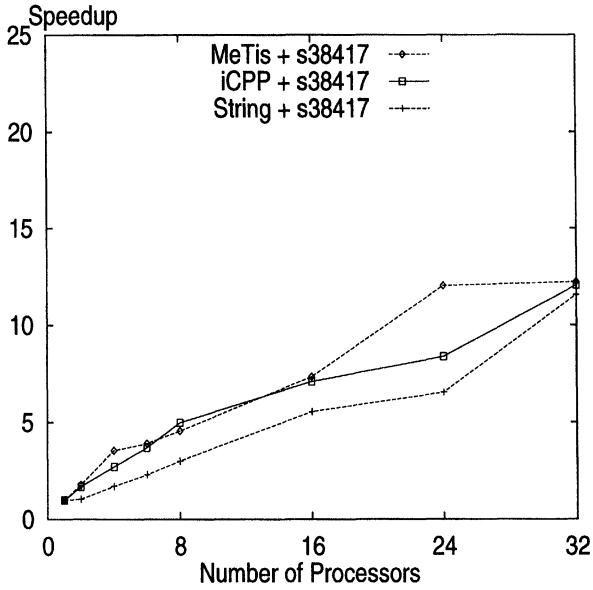

FIGURE 11 Speedups on the IBM SP2 wihout pre-simulation.

for the s38417 circuit without pre-simulation. Among those three partitioning algorithms, the string algorithm is the slowest one while the MeTis outperforms the iCPP algorithm with only two exceptions (when there are 8 processors and 32 processors, respectively). Recall that in Figure 4, in terms of concurrency, the string algorithm performs the best and the MeTis is the worst one. It seems that the amount of interprocessor communication for the iCPP algorithm still needs to be minimized. An analysis of the results shows that the iCPP produces much more rollbacks than the MeTis does. Therefore for the iCPP algorithm to be more competitive, there is a need to reduce the rollback ratio which may be achieved by resynchronizing the simulation more frequently. This in turn requires a more efficient algorithm for the GVT (global virtual time) computation.

6. CONCLUSIONS

In this study, the effect of concurrency, or instantaneous load balancing, is considered in the

development of a partitioning algorithm for parallel logic simulation. The resulting algorithm is called the improved concurrency preserving partitioning (iCPP) algorithm. The algorithm has three phases so that three different goals can be separately considered in order to achieve a good compromise. Unlike most other partitioning algorithms, the iCPP algorithm preserves the concurrency of a circuit by dividing the entire circuit into subcircuits according to the distribution of primary inputs and flip-flops. Preserving concurrency produces better instantaneous load balancing and reduces the gap between local virtual times (LVTs). It in turn reduces the number of erroneous events. Therefore with a good compromise among three conflicting goals, the iCPP algorithm produces reasonably good performance for large circuits.

To measure the concurrency, a concurrency metric that requires neither sequential nor parallel simulation is developed and used to compare six different algorithms, including the random, the depth-first-search (DFS), the string, the MeTis, the CPP, and the iCPP. The results indicate that the iCPP algorithm preserves concurrency pretty well while at the same time it achieves lower interprocessor communication compared to the string algorithm (which also targets concurrency explicitly). In addition to using the concurrency metric, optimistic discrete event simulations are performed on an Intel Paragon and an IBM SP2 to compare the effects of different partitioning algorithms on parallel logic simulations. The large scale comparison effort by itself represents a pretty unique contribution to the research community of parallel logic simulation.

Acknowledgements

This research is supported by an Ohio State Board of Regent Research Investment grant. The authors would like to thank the ASC Major Shared Resource Center of the US Air Force for providing Paragon access and Argonne National Laboratory and Ohio Supercomputer Center for allowing the use of their IBM SP2 machines.

References

[1] Bailey, M. L., Briner, J. V. Jr. and Chamberlain, R. D. (1994). "Parallel Logic Simulation of VLSI Systems", *ACM Computing Surveys*, **26**(3), 255–294.

[2] Brglez, F., Bryan, D. and Kozminski, K. (1989). "Combinational Profiles of Sequential Benchmark Circuits", *Proc. of the IEEE International Symp. on Circuits and Systems*, pp. 1929–1934.

[3] Boukerche, A. and Tropper, C. (1994). "A Static Partitioning and Mapping Algorithm for Conservative Parallel Simulations", *Proc. of 8th Workshop on Parallel and Distributed Simulation*, pp. 164–172.

[4] Boukerche, A. and Tropper, C. (1995). "SGTNE: Semi-Global Time of the Next Event Algorithm," *Proc. of 9th Workshop on Parallel and Distributed Simulation*, pp. 68–77.

[5] Briner, J. V. Jr. (1990). "Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time," *Ph.D. Dissertation*, Duke University.

[6] Chamberlain Roger, D. and Henderson Cheryl, D. (1994). "Evaluating the Use of Pre-Simtilation in VLSI Circuit Partitioning," *Proc. of 8th Workshop on Parallel and Distributed Simulation*, pp. 139–146.

[7] Fiduccia, C. M. and Mattheyses, R. M. (1982). "A Linear-Time Heuristic for Improving Network Partitions," *Proc. of the Design Automation Conference*, pp. 175–181.

[8] Jefferson, D. R. (1985). "Virtual Time," *ACM Trans. on Programming Languages and Systems*, **7**(3), 404–425.

[9] Vikas Jha and Rajive Bagrodia (1996). "A Performance Evaluation Methodology for Parallel Simulation Protocols," *Proc. of 10th Workshop on Parallel and Distributed Simulation*, pp. 180–185.

[10] Kapp, K. L., Hartrum, T. C. and Wailes, T. S. (1995). "An Improved Cost Function for Static Partitioning of Parallel Circuit Simulations Using a Conservative Synchronization Protocol," *Proc. of 9th Workshop on Parallel and Distributed Simulation*, pp. 78– 85.

[11] Karypis, G. and Kumar, V. (1995). "Multilevel graph partition and sparse matrix ordering," *Int'l Conf on Parallel Processing*, **3**, 113–122.

[12] Keller, J., Rauber, T. and Rederlechner, B. (1996). "Conservative Circuit Simulation on Shared-Memory Multiprocessors," *Proc. of 10th Workshop on Parallel and Distributed Simulation*, pp. 126–134.

[13] Kernighan, B. W. and Lin, S. (1970). "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, **49**, pp. 291 –307.

[14] Hong, K., Kim and Jack, Jean (1996). "Concurrency Preserving Partitioning (CPP) for Parallel Logic Simulation," *Proc. of 10th Workshop on Parallel and Distributed Simulation*, pp. 98–105.

[15] Pavlos, Konas and Pen-Chung, Yew (1991). "Parallel Discrete Event Simulation on Shared-Memory Multiprocessors," *The 24th Annual Simulation Symposium*, pp. 134– 148.

[16] Kravitz, S. A. and Ackland, B. D. (1988). "Static *vs.* Dynamic Partitioning of Circuits for a MOS Timing Simulator on A Message-Based Multiprocessor," *Proc. of the SCS Multiconference on Distributed Simulation*, pp. 136– 140.

[17] Levendel, V. H., Menon, P. R. and Patel, S. H. (1982). "Special Purpose Computer for Logic Simulation Using Distributed Processing," *Bell System Technical Journal*, **61**(10), 2873–2909.

[18] Yi-Bing, Lin (1992). "Parallelism analyzers for parallel discrete event simulation," *ACM Transactions on Modeling and Computer Simulation*, pp. 239–264.
[19] Malloy, B. A., Lloyd, E. L. and Soffa, M. L. (1994). "Scheduling DAG's for Asynchronous Multiprocessor Execution," *IEEE Transactions on Parallel and Distributed Systems*, **5**(5), 495–508.
[20] Manjikian, N. and Loucks, W. M. (1993). "High Performance Parallel Logic Simulation on a Network of Workstations," *Proc. of 7th Workshop on Parallel and Distributed Simulation*, pp. 76–84.
[21] Mueller-Thuns, R. B., Saab, D. G., Damiano, R. F. and Abraham, J. A. (1993). "VLSI Logic and Fault Simulation on General-Purpose Parallel Computers," *IEEE Trans. on Computer-Aided Design*, **12**(3), 446–460.
[22] Biswajit Nandy and Loucks, W. M. (1992). "An Algorithm for Partitioning and Mapping Conservative Parallel Simulation onto Multicomputers," *Proc. of 6th Workshop on Parallel and Distributed Simulation*, pp. 139– 146.
[23] Laura, A., Sanchis (1989). "Multiple-Way Network Partitioning," *IEEE Transactions on Computers*, **38**(1), 62–81.
[24] Smith, S. P., Underwood, B. and Mercer, M. R. (1987). "An Analysis of Several Approaches to Circuit Partitioning for Parallel Logic Simulation," *IEEE International Conference on Computer Design*, pp. 664–667.
[25] Soule, L. and Gupta, A. (1989). "Characterization of Parallelism and Deadlocks in Distributed Digital Logic Simulation," *Proc. of the 26th Design Automation Conf.*, pp. 81–86.
[26] Sporrer, C. and Bauer, H. (1993). "Corolla Partitioning for Distributed Logic Simulation of VLSI-Circuits," *Proc. of 7th Workshop on Parallel and Distributed Simulation*, pp. 85–92.

Authors' Biographies

Hong K. Kim is currently working towards the Ph.D degree in computer science at Wright State University, Dayton, Ohio. He received the M.S. degree in computer science and the B.S. degree in industrial engineering from New Jersey Institute of Technology, Newark, N.J., and Korea University, Korea, respectively. His research interests include parallel simulation, partitioning algorithms, computer aided design of circuits, and distributed computing. He is a member of ACM and SCS.

Jack Shiann-Ning Jean received the B.S. and M.S. degrees from the National Taiwan University, Taiwan, in 1981 and 1983, respectively, and the Ph.D. degree from the University of Southern California, Los Angeles, C.A., in 1988, all in electrical engineering. In 1989, he received a research initiation award from the National Science Foundation. Currently he is an Associate Professor in the Computer Science and Engineering Department of Wright State University, Dayton, Ohio. His research interests include parallel processing, reconfigurable computing, and machine learning.