# Recursive Feedforward Network 상에서의 효율적인 병렬 시뮬레이션 알고리즘

## An Efficient Parallel Simulation Algorithm on Recursive Feedforward Network

옥 시 건*, 정 창 성**
S. K. Ok, C. S. Jeong

**Abstract**

In this paper we present an efficient parallel simulation algorithm in recursive feedforward network (RFN) which can reduce the simulation delay while decreasing the number of null messages compared to the previous result. As a preprocessing step, we first determine the group and type of each output channel for the nodes using DFS(Depth First Search) algorithm, and show that the number of null messages as well as the simulation delay that may arises in the inner nodes of RFN can be decreased by making use of the new simulation scheme. By the new scheme we decide if null messages are sent to the output channels or not according to the group to which it belongs.

## 1. 서 론

시스템의 규모나 복잡성이 증가하면 수학적, 분석적(analytical) 방법으로는 시스템의 상태(state)를 효율적으로 분석하기는 거의 불가능하기 때문에 시뮬레이션은 아주 유용한 해결책으로 사용되고 있다[1]. 그러나 하나의 프로세서를 사용한 순차적(sequential) 시뮬레이션의 경우 global clock을 이용해서 단계적으로 시뮬레이션 시간을 전진시켜야 하기 때문에 시뮬레이션 해야 할 사건의 수효가 많아지면 사건 목록(event list)에 대기해야하는 사건들이 엄청나게 증가하므로 시뮬레이션을 수행하는데 많은 계산 시간이 필요하게 된다[1]. 이와 같은 문제점들을 해결하기 위하여 최근에는 여러 개의 프로세서들로 구성된 병렬 컴퓨터를 이용한 병렬 시뮬레이션(parallel simulation)에 관한 연구가 진행되고 있으며, 특히 이산 사건의 병렬 처리에 관한 많은 논문이 발표되고 있다[2].

병렬 이산 사건 시뮬레이션은 크게 낙관적(optimistic) 알고리즘과 보수적(conservative)알고리즘으로 구분된다. 낙관적 알고리즘은 프로세서(processor)들이 다른 프로세서들의 시뮬레이션 진행 상황은 전혀 고려하지 않고 자신

* 고려대학교 공과대학 산업공학과
** 고려대학교 공과대학 전자공학과

의 사건 목록(event list)에 쌓여있는 정보만을 처리하는 방법으로 통신(communication)이 이루어지는 프로세서간에 인과성(causality)의 오류가 발생할 수 있다[2]. 이 경우 프로세서들은 이미 처리한 메시지 보다 더 작은 타임 스탬프(timestamp)를 갖고 있는 메시지가 도착을 하면 롤백(rollback)을 통하여 인과성 오류가 발생한 시점으로 되돌아가서 다시 시뮬레이션을 수행하게 된다[2]. 그리고 인과의 오류를 해결하기 위하여 이미 사건들을 처리하고 있는 다른 프로세서로 anti-message를 보내게 되고 anti-message를 받게 된 프로세서들은 상쇄(annihilation)작용으로 그 시점 이전의 메시지들을 모두 무효 처리한다. 이같은 낙관적(optimistic) 병렬 시뮬레이션의 대표적 알고리즘으로는 Time-Warp 알고리즘이 있다[2].

보수적(conservative) 알고리즘은 각 프로세서의 인과의 오류가 발생하지 않음을 확신할 때까지 사건 처리를 유보하게 된다[3]. 이같은 보수적(conservative) 병렬 시뮬레이션의 대표적인 Chandy-Misra 알고리즘에서는 프로세서에 대기하고 있는 메시지의 타임 스탬프 값이 다음에 도착하는 메시지의 타임 스탬프 값 보다 작다는 확신을 가질 때만 메시지를 처리함으로써 해당 프로세서의 시뮬레이션 시간(simulation time)이 롤백 되거나 인과의 오류가 발생하는 것을 방지하게 된다[4]. 그러나 Chandy-Misra 알고리즘을 사용할 경우 교착상태(deadlock)가 발생할 위험이 있기 때문에 이를 해결하기 위해 Misra는 널(null) 메시지를 사용하는 방법을 제안하였다[4]. 널 메시지는 단순히 시뮬레이션을 진행시키위한 메시지로 시스템 상태에 관한 아무런 내용도 포함하고 있지 않고 다만 프로세서의 현재 타임 스탬프에 관한 정보만을 포함하고 있는 메시지로 리얼(real) 메시지가 보내지지 않는 경로에는 널 메시지를 보냄으로써 프로세서들이 교착상태에 빠지거나 인과의 오류가 생기지 않도록 한다[5]. 그러나 시뮬레이션 시스템의 프로세서간의 연결 채널이 증가하면 널 메시지의 개수가 많아지므로 많은 계산 시간이 걸리게 된다[6].

De Vries는 널 메시지를 보내는 대신 메시지가 노드에 도착을 하게 되면 타임 스탬프 정보를 이용해서 메시지가 도착하지 않은 경로에 대한 예측을 함으로써 교착상태를 해결하고 널 메시지의 숫자를 줄이는 방법을 제안했다[7]. 그러나, De Vries 알고리즘에서는 여러 개의 경로가 2개의 노드 사이에 연결된 feedforward network가 recursive하게 나타나는 recursive feedforward network(RFN)의 경우 리얼 메시지가 보내지지 않는 경로내의 feedforward network에 널 메시지가 보내지지 않기 때문에 예측을 할 수가 없으므로 시뮬레이션이 지연되는 단점이 있다[7]. 본 논문에서는 recursive feedforward network상에서 이전의 병렬 알고리즘에 비해 널 메시지의 숫자를 줄이는 동시에 널 메시지를 사용하지 않음으로 인하여 발생할 수 있는 시뮬레이션의 지연시간을 줄일 수 있는 효율적인 병렬 시뮬레이션 알고리즘을 제시한다. DFS 알고리즘을 사용하여 recursive feedforward network상의 각 노드의 출력 채널의 그룹(group) 및 유형을 결정하고, 그룹(group) 및 유형에 관한 정보를 사용하여 널 메시지를 선택적으로 보내줌으로써 내부 노드에서 발생할 수 있는 시뮬레이션 지연시간을 단축시키면서 최대한도로 널 메시지 숫자를 감소시킬 수 있는 새로운 시뮬레이션 기법을 제안한다. 이전의 알고리즘은 Misra의 알고리즘과 같이 모든 채널에 널 메시지를 보내주거나, 또는 De Vries의 알고리즘과 같이 널 메시지를 보내지 않고 예측만 하는 방법을 사용함으로써 과도한 널 메시지의 사용및 시뮬레이션 지연을 피할 수 없었으나, 본 논문에서는 각 그룹별로 채널유형에 근거하여 널 메시지를 보냄으로써 메시지의 숫자를 감소시킬 수 있는 병렬 시뮬레이션 알고리즘을 제시한다.

본 논문의 구성은 다음과 같다. 2장에서는 병렬 시뮬레이션 모델 및 이에 필요한 여러 용어의 정의와 가정들을 설명하고, 3장에서는 Misra및 De Vries의 보수적 알고리즘 및 내재한 문제점에 대해 보다 상세히 기술한다. 4장에서는 병렬 시뮬레이션 알고리즘의 기본 원리에 대해 설명한 후 전 처리(preprocessing) 알고리즘 및 이를 이용한 새로운 병렬 시뮬레이션 알고리즘을 제시한다. 그리고 5장에서는 결론을 내린다.

## 2. 병렬 시뮬레이션 모델

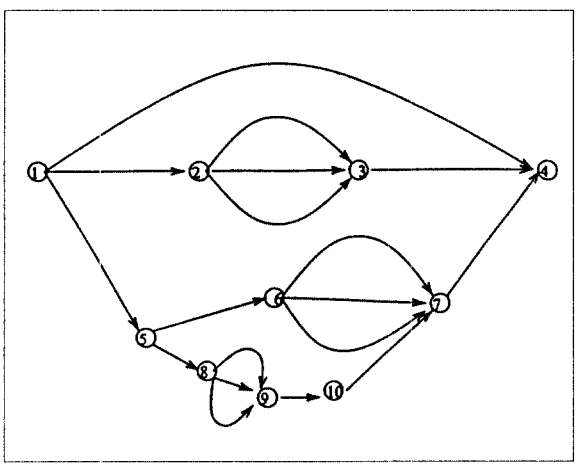
본 절에서는 병렬 시뮬레이션 모델에 사용되는 용어들의 정의 및 가정들에 대해 기술한 후 이를 사용하여 병렬 시뮬레이션모델에 대하여 설명한다.

**정의 1 : 시뮬레이션 네트워크** $S(N,C)$는 노드(node)들의 집합$N$과 이들을 연결하는 directed 채널들의 집합 $C$로 이루어진 방향성 그래프(directed graph)로 표현한다. $S(N,C)$에서 노드 $p$에서 노드 $q$로 directed된 채널은

$chan(p,q)$로 나타내고, $chan(p,q)$는 노드 $p$에 대해서는 출력 채널(output channel), 노드 $q$에 대해서는 입력 채널(input channel)이라 한다.

**정의 2** : 시뮬레이션 네트워크에서 입력 채널이 한개이고 출력 채널이 2개 이상인 노드를 **disperse 노드**라 하며, 입력 채널이 존재하지 않는 disperse 노드를 **source 노드**라 한다. Input 채널이 2개 이상이고 출력 채널이 하나인 노드를 **merge 노드**라 하며, 출력 채널이 존재하지 않는 merge 노드를 **sink 노드**라 한다. 또한 하나의 입력 채널과 출력 채널만이 연결된 노드를 **single pass 노드**라 한다.

예를 들어 〈그림 1〉에서 노드 2, 5, 6, 8은 disperse 노드, 노드 1은 source 노드, 노드 3, 7, 9 는 merge 노드, 노드 4는 sink 노드, 노드 10은 single pass 노드이다.




〈그림 1〉

**정의 3** : 시뮬레이션 네트워크 $S(N,C)$에서 $N$에 속한 노드 $p$와 노드 $q$ 사이의 **simple directed path** $sp(p,q)$는 노드 $p$와 노드 $q$를 연결하는 $v_0(=p)c_1v_1c_2v_2 \ldots c_kv_k(=q)$ ($k\geq 1$, $c_i=chan(v_{i-1},v_i)$, $v_i$는 unique) sequence로 정의하고, $sp(p,q)$는 $p$에서 $q$로 directed 되었다고 한다.

**정의 4** : **Feedforward network** $FN(p,q)$는 노드 $p$와 $q$를 연결하고, $p$에서 $q$로 directed된 한개 이상의 simple directed path들로 구성이 되는 시뮬레이션 네트워크이다.

**정의 5** : 노드 $p$와 노드 $q$ 사이의 **multiple track** $mt(p,q)$는 노드 $p$와 노드 $q$를 연결하는 여러 개의 simple 또는 complex track 으로 구성된다. 노드 $p$와 노드 $q$ 사이의 **simple track** $st(p,q)$는 노드 $p$와 $q$사이의 simple directed path $sp(p,q)$이다. **Complex track** $ct(p,q)$는 노드 $p$와 노드 $q$ 사이에 하나 이상의 multiple track들과 simple path들이 연결된 $v_0(=p)t_1v_1t_2v_2 \ldots t_kv_k(=q)$ ($t_j\in sp(v_{i-1},v_i)$ or $mt(v_{i-1},v_i)$, $k\geq 1$) sequence로 정의하고, $ct(p,q)$는 $t_1, t_2, t_3, \ldots, t_k$로 구성되었다고 한다. Track $st(p,q)$ 또는 $ct(p,q)$에서 $p$를 시작(start) 노드, $q$를 종료(end) 노드라 하고, $p$와 $q$는 track상에서 서로 **대응**되는 노드라고 한다.

노드 $p$와 노드 $q$ 사이의 recursive feedforward network는 feedforward network내에 recursive하게 다른 feedforward network를 포함하는 시뮬레이션 네트워크로서 다음과 같은 형태로 정의될 수 있다.

**정의 6** : 노드 $p$와 노드 $q$ 사이의 **recursive feedforward network** $RFN(p,q)$는 노드 $p$, $q$및 $mt(p,q)$로 이루어지는 시뮬레이션 네트워크다.

예를 들어 〈그림 1〉에서 RFN(1,4)는 한 개의 simple track $st_1(1,4)$와 2개의 complex track $ct_2(1,4)$, $ct_3(1,4)$로 구성되어 있다. Complex track $ct_2(1,4)$는 $sp_2(1,2)$, $mt_4(2,3)$, $sp(3,4)$로 구성되고, $ct_3(1,4)$는 $sp_3(1,5)$, $mt_5(5,7)$, $sp_4(7,4)$로 구성되어 있다. 다시 $mt_5(5,7)$은 2개의 complex track $ct_4(5,7)$, $ct_5(5,7)$으로 구성되어 있다. Complex track $ct_4(5,7)$과 $ct_5(5,7)$은 내부에 3개의 simple path로 이루어진 multiple track $mt_6(6,7)$과 $mt_7(8,9)$을 포함하고 있다.

**정의 7** : **시뮬레이션 시간**은 시뮬레이션 네트워크에서 노드나 채널 상에서 현재 시뮬레이션이 진행되는 시간이고, 이때 노드의 시뮬레이션 시간을 로컬 시간(**local time**), 채널의 시뮬레이션 시간을 **채널 시간**이라 한다.

**정의 8** : 노드에서 메시지를 처리하기 시작하는 시간을 **initiation time**, 메시지 처리를 완료하고 다른 메시지 처리를 시작할 수 있는 시간을 **next initiation time**이라 한다.

**정의 9** : 노드 $p$에서 $q$로 $chan(p,q)$를 통해 메시지 $m$을 보낼 때 $m$의 **타임 스탬프**는 $q$에 메시지가 도착하는 시간으로 이는 노드 $p$가 메시지 $m$을 보낸 로컬 시간과 같다.

**정의 10** : 노드 $p$와 노드 $q$사이에 track $tr(p,q)$가 존재할 때 track $tr(p,q)$에서의 시뮬레이션 지연$d(tr(p,q))$는 $tr(p,q)$에 있는 모든 노드들에서 소요되는 최소 시뮬레이션 시간의 합이다.

**정의 11** : 노드 $p$와 노드 $q$사이에 track$tr_1(p,q)$와 $tr_2(p,q)$

가 존재하고 노드 $p$에서 next initiation time $T_1$에 $tr_1(p,q)$를 따라 메시지 $m$을 보내면, track $tr_2(p,q)$를 따라 $q$에 들어오는 메시지는 $t_b=d(tr_2(p,q))$일 때 적어도 $T_1+t_b$이후에 도착하게 된다. 따라서 $tr_1(p,q)$를 따라 $q$의 입력 채널$c_1$에 도착하는 메시지 $m$에 노드 $p$의 next initiation time $T_1$정보를 갖고 있다면, $tr_2(p,q)$를 따라 입력 채널$c_2$를 통해 $q$에 들어오는 메시지는 적어도 $T_1+t_b$ 이후에 도착한다는 것을 예측할 수 있다. 이때 $T_1+t_b$를 $c_2$의 채널 시간의 **예측 시간**(predicted time)이라 한다.

본 논문에서 다루는 병렬 시뮬레이션 모델은 recursive feedforward network로서 각 노드는 프로세스에 해당하고, 각 노드간의 통신은 채널을 통해서 이루어진다. 병렬 시뮬레이션 모델에서는 각 노드는 입력 채널을 경유하여 노드에 도착한 메시지를 대기행렬에 저장하고, 메시지와 연관된 사건처리가 끝나면 출력 채널을 통하여 다른 노드로 메시지를 보내는 시뮬레이션 작업을 동시에 수행한다. 인접해 있는 노드 $p$와$q$간에 메시지를 보내는데 소모되는 시간은 없다고 가정한다. 따라서 노드 $p$에서 메시지를 보내는 시간과 노드 $q$에서 메시지를 받는 시간은 일치한다.

각 노드는 입력 채널을 통해 들어오는 메시지를 처리하여 출력 채널을 통해 하나의 리얼 또는 널 메시지를 생성하게 된다. 노드 $p$에서 생성된 메시지는 노드 $p$의 next initiation time에 대한 정보를 저장하여 도착된 노드에서 이를 사용하여 채널 시간의 예측을 할 수 있게 된다. 널 메시지가 노드에 도착하면 실제적인 시뮬레이션을 수행하지 않고 단지 시뮬레이션 시간을 갱신시키는 역할을 한다[2]. 즉 널 메시지가 simple pass인 노드 $p$에 도착하면 실제적인 시뮬레이션을 수행하지 않고 노드 $p$의 로컬 시간에 노드 $p$에서의 시뮬레이션 지연을 더한 값을 타임 스탬프로 갖는 널 메시지를 생성하고, merge 노드인 경우 도착한 널 메시지의 타임 스탬프를 사용하여 채널시간을 갱신하고 다른 채널 시간 값을 예측하게 된다. Merge 노드는 모든 채널에 메시지가 도착했거나, 도착하지 않는 경우 도착시간을 예측할 수 있을 때 비로소 현재 노드에 대기하고 있는 리얼 메시지중 타임 스탬프 값이 최소인 리얼 메시지를 처리하게 되고 만약 타임 스탬프가 최소인 메시지가 리얼이 아니라면 대기하게 된다.

모든 노드의 시뮬레이션 시간(simulation time)은 단조증가(monotonous increasing)하기 때문에 노드$p$와 노드$q$ 사이의 $chan(p,q)$를 따라 도착하는 메시지 $m$의 타임 스탬프 값은 순서(order)를 유지하고, 인과(causality)의 오류를 범하지 않는다.

## 3. Previous Result

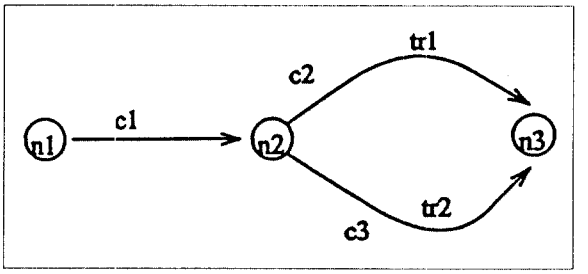
본 논문에서는 앞으로 도착하는 메시지의 도착 시간이 현재 처리하고자 하는 메시지의 도착시간 보다 큰 경우에만 사건을 처리하는 보수적 알고리즘을 다루려고 한다. 보수적 알고리즘으로는 Misra가 인과성 오류를 방지하고 교착상태를 해소하기 위하여 널 메시지를 사용하는 방법을 제안했고[2], De Vries는 널 메시지를 사용하는데 따르는 오버헤드를 줄이기 위하여 아무런 메시지도 대기하지 않고 있는 채널들에 대하여 메시지의 도착시간을 예측하는 방법을 제안하였다[7]. 본 절에서는 보수적 알고리즘에서 대표적인 Misra와 De Vries의 알고리즘 및 이들 알고리즘이 내재한 문제점에 관해 설명한다.

### 3.1 Misra의 널 메시지(null message) 알고리즘

Misra의 알고리즘에서는 disperse 노드 $p$에서 출력 채널을 통해서 메시지를 보낼 때 $p$의 다른 출력 채널에도 널 메시지를 보내게 되고 이러한 널 메시지를 입력 채널$c$를 통해 받는 merge 노드 $q$에서는 널 메시지가 도착한 시간 이전에는 $c$에 다른 메시지가 도착하지 않는다는 정보를 가지고, $q$에서의 교착상태를 해결하고 시뮬레이션을 진행시킨다[3].

예를 들어 〈그림 2〉에서 disperse 노드 $n_2$에서 발생된 첫 번째 리얼 메시지와 두 번째 메시지가 track $tr_1$을 따라 merge 노드 $n_3$의 입력 채널$c_1$에 도착한다고 하자. Merge 노드 $n_3$에서는 모든 입력 채널 시간을 비교한 다음 가장 작은 채널 시간을 갖는 입력 채널에 리얼 메시지가 대기할 경우에만 메시지를 처리해야 하나, $n_3$의 $tr_2$에서의 입력 채널$c_2$에는 메시지가 아직 도착하지 않았기 때문에 노드 $n_3$에서는 시뮬레이션이 지연된다. 만약 disperse 노드 $n_2$에서 두 번째의 리얼 메시지가 track $tr_1$을 따라 보내짐과 동시에 track $tr_2$로 널 메시지를 보낸다면 널 메시지를 받은 $n_3$에서 $c_2$에 널 메시지의 타임 스탬프이후에 메시지가 도착한다는 사실을 이용하여 교착상태 없이 $n_3$에서 시뮬레이션 지연을 방지해 줄 수있다.

이와 같이 Misra 알고리즘은 널 메시지를 사용하여 교



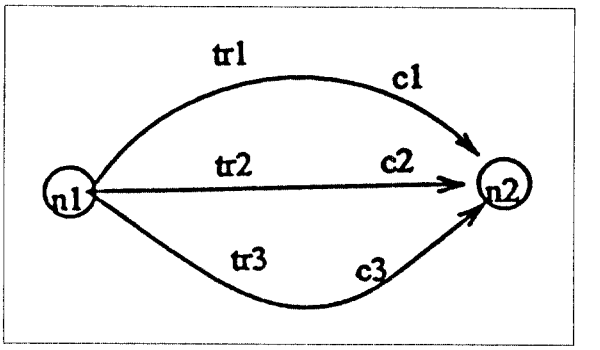

〈그림 2〉

착상태(deadlock) 없이 시뮬레이션을 원활하게 진행시킬 수는 있지만 시뮬레이션 시스템의 구조에 따라 disperse 노드의 수가 증가하면 생성되는 널 메시지의 양이 상당히 증가하기 때문에 프로세서간의 과도한 통신량의 증가로 시뮬레이션의 성능이 크게 감소하는 단점을 가지고 있다.

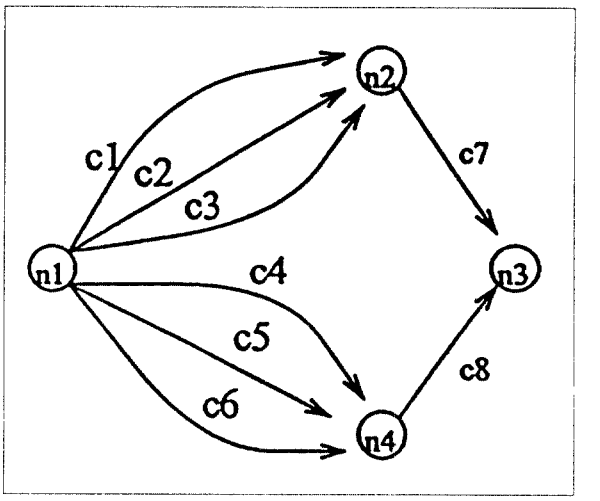
### 3.2 De Vries의 알고리즘

De Vries는 Misra 알고리즘에서 널 메시지의 숫자를 감소시키기 위하여 여러 개의 입력 채널들이 연결되어 있는 merge 노드에 리얼 메시지가 도착하면 아직 메시지가 도착하지 않은 나머지 채널에 대해서 예측을 하는 방법을 제안했다[7].

〈그림 3〉에서 disperse 노드 $n_1$을 출발한 리얼 메시지가 track $tr_1$을 경유하여 merge 노드 $n_2$로 보내지면 노드 $n_1$의 next initiation time값과 $tr_1$의 시뮬레이션 지연이 각각 $T_0$, $d_1$이라면 노드 $n_2$에 도착한 메시지의 타임 스탬프 값은 $T_0+d_1$이 되고, track$tr_2$와 track $tr_3$의 시뮬레이션 지연 시간을 $d_2$, $d_3$라 하면 track$tr_2$와 track $tr_3$에 대한 $n_2$의 각 채널$c_2$와 $c_3$에서의 예측시간은 $T_0+d_2$, $T_0+d_3$가 된다. 따라서 $c_2$와 $c_3$의 예측시간과 리얼 메시지가 대기하고 있는 $c_1$의 채널 시간을 비교하여 $c_1$의 채널 시간이 $c_2$, $c_3$의 예측시간 보다 작다면 $c_1$의 리얼 메시지를 처리할 수 있다. 이와 같이 De Vries 알고리즘에서는 널 메시지를 보내지 않고 채널 시간을 예측함으로써 시뮬레이션 지연을 줄이게 된다. 그러나 〈그림 4〉에서 노드 $n_1$에서 $c_4$를 통해 $n_4$에 메시지를 보낼 때 De Vries 알고리즘에서는 다른 채널에 널 메시지를 보내지 않으므로 $n_4$와는 다른 track에 속해있는 merge 노드 $n_2$에서 예측을 할 수 없기 때문에 $n_2$에서는 시뮬레이션 지연이 발생할 수 있다.




〈그림 3〉




〈그림 4〉

## 4. 병렬 시뮬레이션

본 논문에서는 널 메시지의 개수를 감소시키면서 시뮬레이션의 지연을 방지하기 위하여 채널들을 그룹화하고 simple type과 complex type으로 분류한 후 이에 대한 정보를 기초로 시뮬레이션을 진행시키는 새로운 시뮬레이션 기법을 제시하려고 한다. 본 절에서는 먼저 채널들의 그룹화 및 유형을 이용한 시뮬레이션의 기본 원리에 대해 설명한 후 RFN이 주어 졌을 때 그룹화 및 유형 설정을 수행하는 전 처리 알고리즘(preprocessing) 및 이를 사용한 병렬 시뮬레이션 알고리즘을 제시한다.

### 1) 기본 원리

병렬 시뮬레이션의 기본 원리를 설명하기 위해 다음과 같이 recursive feedforward network상에서의 채널들의 그룹 및 유형에 대해 정의한다.

**정의 12** : 노드 $m$과 $n$ 사이의 채널$c(=chan(m,n))$에 대하여 2개 이상의 track을 갖는 $mt(p,q)$의 한 track $t(p,q)$가 $c$를 포함하는 $v_0(=p)t_1v_1t_2v_2 \ldots v_i(=m)t_{i+1}v_{i+1}(=n) \ldots v_{k-1}t_kv_k(=q),(i=0,1,\ldots,k\text{-}1,\ t_{i+1}=c,\ t_j \in sp(v_j,v_{j+1})$ or $mt(v_j,v_{j+1})$ for $j \neq i+1)$의 sequence로 나타낼 수 있으면, $t(p,q)$는 $c$의 father track $ft(c)$, $p$를 $c$의 nearest disperse node $nd(c)$, $q$를 nearest merge node $nm(c)$라 정의한다.

**정의 13** : Disperse 노드 $p$의 output channel들의 집합 $C_{out}(p)$는 같은 nearest merge 노드 $v_i$를 갖는 출력 channel들의 집합 $GR_i(p,v_i)$들로 다음과 같이 disjoint하게 나누어 질 수 있고,

$$C_{out}(p) = \bigcup_i GR_i(p,v_i), \text{ where } \bigcap_i GR_i(p,v_i) = \phi$$

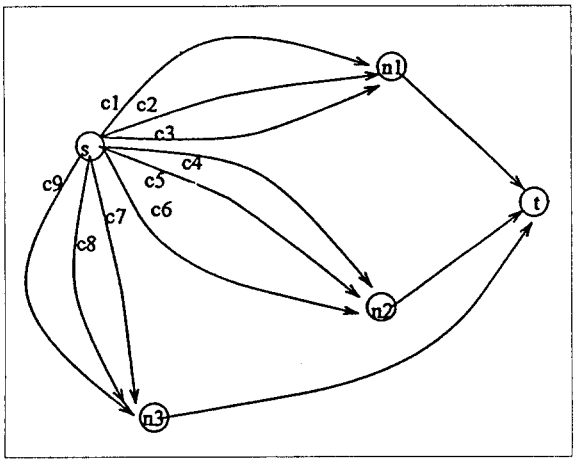
$$GR_i(p,v_i) = \{c \in c \in C_{out}(p),\ nm(c) = v_i\}$$

이때 $GR_i(p,v_i)$를 같은 nearest merge 노드 $v_i$를 갖는 노드 $p$의 그룹이라 정의하고, $C_{out}(p)$를 그룹으로 나누는 것을 $p$의 출력 채널들을 그룹화 한다고 한다.

**정의 14** : Recursive feedforward network에서 노드 $m$에서 $n$으로 directed된 채널$c$가 있을 때 $c$의 유형 $tch(c)$는 $ft(c)$가 complex track이면 complex type, simple track이면 simple type으로 정의한다.

예를 들어 〈그림 5〉와 같이 RFN(s,t)가 주어 졌다고 하자. RFN(s,t)는 3개의 track $tr_1$(s,t), $tr_2(s,t)$, $tr_3(s,t)$로 구성되어 있고 $tr_1$(s,t)는 $mt(s,n_1)$, $sp(n_1,t)$, $tr_2(s,t)$는 $mt(s,n_2)$, $sp(n_2,t)$, $tr_3(s,t)$는 $mt(s,n_3)$, $sp(n_3,t)$로 각각 구성되어 있다. 다시 $mt(s,n_1)$는 3개의 track $tr_4(s,n_1)$, $tr_5(s,n_1)$, $tr_6(s,n_1)$, $mt(s,n_2)$는$tr_7(s,n_2)$,$tr_8(s,n_2)$,$tr_9(s,n_3)$,$mt(s,n_3)$는$tr_{10}(s,n_3)$,$tr_{11}(s,n_3)$, $tr_{12}(s,n_3)$으로 각각 구성되어 있다. 노드s에서 채널$c_1$을 경유하여 track $tr_4$를 따라 메시지가 보내진다면 Misra 알고리즘에서는 $c_2 \sim c_9$의 나머지 모든 채널들을 통해 널 메시지를 보내야 하므로 널 메시지의 숫자가 증가하게 된다.

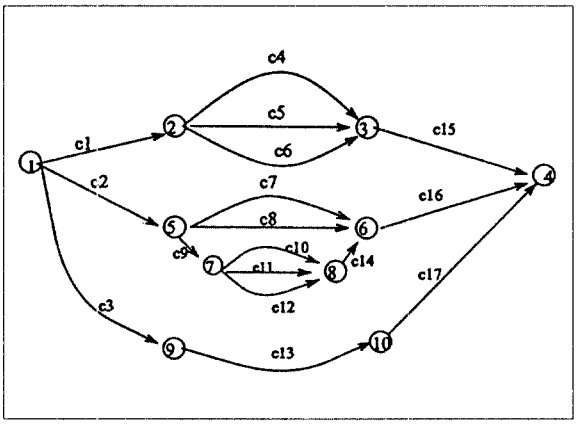
그러나, $c_1$에 리얼 메시지 $m_1$을 보내면 merge 노드 $n_1$에서는 $m_1$을 이용하여 $tr_5$, $tr_6$를 통해 들어오는 $n_1$의 입력 채




〈그림 5〉

널 시간을 예측할 수 있으므로 $c_2$와 $c_3$를 통하여 널 메시지를 보낼 필요가 없다. 마찬가지로 $c_4$에 널 메시지를 보내면 merge 노드 $n_2$에서는 나머지 track $tr_8$,$tr_9$에서 들어오는 $n_2$의 입력 채널 시간을 예측할 수가 있고, $c_7$에 널 메시지를 보내면 $n_3$에서는 나머지 track $tr_{11}$과 $tr_{12}$에서 들어오는 $n_3$의 입력 채널 시간을 예측할 수 있다. 이와 같이 source 노드 $s$에서 출발하는 출력 채널들을 같은 nearest merge node $n_1$, $n_2$, $n_3$를 갖는 3개의 그룹 $GR_1(s,n_1) = \{c_1,c_2,c_3\}$, $GR_2(s,n_2) = \{c_4,c_5,c_6\}$, $GR_3(s,n_3) = \{c_7,c_8,c_9\}$로 나누고 각 그룹의 하나의 채널에 리얼 또는 널 메시지 한 개만을 보내주더라도 모든 merge 노드 $n_1$, $n_2$, $n_3$에서 채널 시간의 예측이 가능하게 되어 Misra의 알고리즘에 비해 널 메시지의 숫자를 감소시킬 수가 있다.

또한 〈그림 6〉과 같은 RFN(1,4)가 주어졌다고 가정을 할 때 RFN(1,4)는 3개의 track $tr_1$(1,4), $tr_2$(1,4), $tr_3$(1,4)로 구성이 되어있고, $tr_1$(1,4)와 $tr_2$(1,4)는 다시 내부에 multiple track인 $mt(2,3)$, $mt(5,6)$을 포함하고 있으며, $tr_3$(1,4)는 simple track으로 구성되어 있다. 출력 채널$c_2$를 통해 track $tr_2$(1,4)를 따라 리얼메시지 $m_1$이 보내진다고 하자. De Vries의 알고리즘에서는 $m_1$이 $tr_2$(1,4)에서 처리되어 노드 4에 도착을 하면 입력 채널$c_{17}$로 들어오는 메시지의 도착시간은 예측할 수 있지만, $c_1$으로는 널 메시지가 보내지지 않았기 때문에 $mt(2,3)$의 merge 노드 3에서는 입력 채널 시간을 수정할 수가 없고, 따라서 $c_{15}$의 채널 시간을 예측을 할 수가 없으므로 시뮬레이션이 지연된다. 따라서 그




〈그림 6〉

룹내에 recursive하게 multiple track을 포함하고 있는 complex track들이 존재한다면 해당 track에는 널 메시지를 보내주어서 내부 multiple track의 merge 노드에서 대기하고 있는 메시지들을 처리함으로써 시뮬레이션을 빨리 진행시키는 것이 필요하다. 이를 위해 정의 14와 같이 disperse 노드 $m$의 출력 채널$c_i$가 속한 father track $ft(c_i)$의 내부에 multiple track을 포함하고 있으면, $ft(c_i)$는 complex track이 되고 $c_i$가 complex type이 되므로, complex type의 출력 채널$c_i$에는 반드시 널 메시지를 보내 주어 complex track내의 merge 노드에서의 입력 채널 시간의 예측을 가능하게 함으로써 시뮬레이션의 지연을 방지하게 된다.

### 2) 전 처리 알고리즘 PR_RFN

PR_RFN(Preprocessing of Recursive Feedforward Network) 알고리즘은 RFN($p$,$q$)가 주어졌을 때 DFS(depth first search) 알고리즘을 사용하여 disperse 노드의 출력 채널들을 그룹화 하고 각 채널의 유형을 결정한다. 다음 절에 기술한 병렬 시뮬레이션 알고리즘에서는 널 메시지 숫자를 줄이면서 시뮬레이션 지연을 감소하는데 PR_RFN에서 생성한 채널들의 그룹화 및 유형에 관한 정보를 사용하게 된다. PR_RFN 알고리즘의 보다 구체적인 내용은 아래에 설명되어 있다. PR_RFN 알고리즘에서 disperse 노드가 아닌 노드 $p$에서의 출력 채널$c$의 유형은 $p$와 $nm(c)$ 사이에 multiple track을 포함하지 않으면 일시적으로 simple type 으로 지정하고, 포함하면 complex type으로 지정한다. 따라서 PR_RFN 알고리즘에서는 모든 disperse 노드$p$에 대해 노드 $p$와 nearest merge node $nm(p)$ 사이의 multiple track 포함 여부를 결정하므로 정확한 출력 채널 유형을 생성하게 된다.

**ALGORITHM PR_RFN**

Input : recursive feedforward network RFN($p$,$q$)

Output : type of each channel in RFN($p$,$q$) and $GR_1(p,v_1)$,$GR_2(p,v_2)$ $\cdots$ $GR_k(p,v_k)$ for each disperse node $p$

1). Mark all the channels "unvisited". For every node $v \in V$, let k($v$)←0. Let $i$←0 and $v$←$p$.($p$ is the node we choose to start the search from.)

2). $i$ ←$i$ + 1, k($v$)←$i$.

3). If there is no unvisited output channel from $v$
then go to step 5

4). If there is any unvisited channel from $v$, do the following:

4.1) Choose an unvisited $chan(v, u)$, and mark $chan(v,u)$ "visited".

4.2) If $v$ is a disperse node
then PUSH($chan(v,u)$,$S_1$)

4.3) If k($u$) ≠ 0
then begin
PUSH($u$,$S_2$)
$tch(v,u)$ ← simple type
go to Step 3

end

4.4) else f($u$)←$v$, $v$←$u$ and go to step 2

5). Execute the proper action according to the node type of $v$.

5.1) If node type of $v$ is a sink or a merge node
then begin
PUSH($v$,$S_2$)
*tch*(f($v$),$v$) ← simple type
end

5.2) If node type of $v$ is a disperse node
then begin
1) for $i$=1 to *out*($v$)
insert(pop($S_1$),pop($S_2$)) into list L
2) store a set of channel with the same merge node $m$ into a group $GR(v,m)$ by searching L
3) *tch*(f($v$),$v$) ← complex type
end

5.3) If node type of $v$ is a simple pass
then if *chan*($v$,$u$) is complex type
then *tch*(f($v$),$v$) ← complex type
else if *chan*($v$,$u$) is simple type
then *tch*(f($v$),$v$) ← simple type

5.4) If $v$ is a root $p$
then stop

5.5) Set $v$← f($v$) and go to step 3

PR_RFN 알고리즘의 step 1)에서는 recursive feedforward network의 모든 채널은 "미방문(unvisited)", 모든 노드 번호 k($v$)는 0, 탐색 시작 노드는 $p$가 된다. Step 2)에서는 현재 위치하고 있는 노드$v$에 대하여 번호를 부여한다. 초기에는 모든 노드 번호가 0 으로 주어졌기 때문에 노드 번호가 0이 아니라면 이미 다른 경로를 이용하여 방문 되었음을 알 수가 있다. Step 3)에서는 현재 위치하고 있는 노드에서 아직 방문하지 않은 채널이 있으면 step 4)를 수행하고, 방문하지 않은 채널이 없다면 step 5)를 수행한다. Step 4)에서는 현재의 노드$v$에서 다른 노드로 연결된 채널중 방문하지 않은 채널을 선택하고 *chan*($v$,$u$)를 "방문(visited)"이라고 표시한다. Step 4.2)에서 만약 노드$v$가 disperse node 라면 step 5.2)에서의 그룹화를 위하여 stack $S_1$에 *chan*($v$,$u$)를 저장한다. Step 4.3)에서 노드 번호 k($u$)가 0이 아니면 $u$는 이미 다른 채널을 통하여 방문한 노드이기때문에 $u$는 merge 노드가 되고, *chan*($v$,$u$)가 속한 *ft* (*chan*($v$,$u$))상에서 $u$에 대한 대응 disperse 노드에서의 grouping을 위하여 stack $S_2$에 노드 $u$를 저장한다. 한편 노드 $v$와 노드 $u$ 사이의 채널$c$(=*chan*($v$,$u$))는 $nm(c)$가 $u$이고 노드 $v$와 노드 $u$ 사이가 multiple track을 포함하지 않기 때문에 $c$는 simple type이 되고, 다른 방문하지 않은 노드를 찾기 위해 step 3)을 수행한다. 만약 step 4.4)에서 노드번호 k($u$)가 0 이면 $u$는 새로 방문하는 노드 이므로 $v$는 $u$의 전노드 f($u$)가 되고 노드 $u$는 새로운 출발 노드 $v$가 된후 번호를 부여받기 위해 step 2)를 수행한다. Step 5)에서는 현재 노드 $v$에서 다른 노드로 연결된 미방문 채널이 존재하지 않기 때문에 이전의 노드 $f(v)$로 회귀하여 다시 미방문 채널을 찾게 되며, 회귀 할 때 $v$의 노드 유형에 따라 채널 유형을 정의하고 그룹화를 수행하게 된다. Step 5.1)에서 만약 노드 $v$가 merge 노드이면 step 4.3)과 마찬가지로 $u$에 대한 대응 disperse 노드에서의 그룹화를 위하여 stack $S_2$에 $v$를 저장하고 *chan*($f(v)$,$v$)의 유형은

simple type이 된다. Step 5.2)에서 만약 노드 $v$가 disperse 노드라면 $chan(f(v),v)$의 father track이 $u$에서 시작되는 multiple track을 포함하는 complex track이므로 $chan(f(v), v)$는 complex type로 정의된다. Disperse 노드 $v$에서 각 출력 채널$c_i$는 과거 step 4.2)에서 stack $S_1$에 저장하고, $c_i$의 father track $ft(c_i)$를 따라 $v$에 대응하는 merge 노드 $nm(c_i)$를 방문한 후 step 4.3) 또는 5.1)에서 $nm(c_i)$가 stack $S_2$에 저장된다. 따라서, stack $S_1$에 저장되어 있는 $v$의 출력 채널 $c_i$와 $nm(c_i)$는 일대일 대응관계가 있게 되므로 $v$의 출력 채널의 개수만큼 stack $S_1$과 $S_2$를 발췌하여서 같은 nearest merge node를 갖는 채널들을 그룹화 하게 된다. Step 5.3)에서 만약 노드 $v$가 simple pass이면 step 5.3)과 같이 $tch(f(v),v)$는 $tch(v,u)$와 동일하게 됨을 쉽게 알 수있다. Step 5)에서 현재의 노드로 부터 과거의 노드로 회귀한 후 step 3)를 수행하여 다시 위와 같은 과정을 반복하게 되고 "미방문" 채널이 존재하지 않고 현재의 노드가 root이면 step 5.4)에서 알고리즘이 종료된다. PR_RFN 알고리즘의 correctness는 앞의 설명에 의해 유추될 수 있고, 각 채널은 두번이상 방문되지 않으므로 채널의 갯수를 $m$이라할 때 PR_RFN 알고리즘은 모든 disperse 노드 $p$의 채널에 대하여 유형과 그룹화를 O(m)에 생성한다.

PR_RFN 알고리즘을 〈그림 6〉의 RFN(1,4)에 대해 예를 들어보도록 하자. RFN(1,4)에서 PR_RFN 알고리즘을 수행하기 위한 초기화 작업으로 모든 채널은 "미방문"이라고 정의되며, 노드 번호(k($v$))는 "0"가 된다. Disperse 노드 1에서 채널$c_1$을 따라 탐색을 시작하면 stack $S_1$에 $c_1$을 저장한 후 $c_1$은 "미방문"에서 "방문"으로 바뀌게 되고, 현재 위치$v$는 노드 2가 된다. 다시 $c_4$를 따라 검색이 진행되면 노드 2가 disperse 노드이기 때문에 stack $S_1$에 $c_4$를 저장하며 $c_4$은 "미방문"에서 "방문"으로 바뀌게 된다. 이와 같은 절차를 반복해서 merge 노드 4를 방문하면 출력 채널이 존재하지 않기 때문에 더이상 미방문 채널을 발견할 수 가 없고 탐색은 다시 노드 3으로 회귀하게 되며 stack $S_2$에 merge 노드 4를 저장한다. $c_{15}$는 simple type으로 표시되고 회귀한 노드 3에서 더이상 미방문 채널을 발견할 수 없고 merge 노드이기 때문에 stack $S_2$에 노드 3을 저장 하고 이때 채널$c_4$는 simple type으로 설정한 후 노드 2로 회귀한다. 노드 2에서 미방문 채널$c_5$를 통해 노드 3을 검색할때 $c_5$를 stack $S_1$에, 그리고 노드 3은 이미 방문이 된 상태이므로 노드 3을 stack $S_2$에 넣고 노드 2로 복귀하면서 $c_5$를 simple type으로 지정한다. 마찬가지로 미방문 채널$c_6$를 따라 노드 3을 검색하며 이때 $c_6$를 $S_1$에 저장을 하고 노드 3이 이미 방문된 노드이므로 stack $S_2$에 노드 3을 넣은 후 노드 2로 복귀하면서 $c_6$를 simple type으로 지정한다. 노드 2에서 더 이상의 미방문 채널을 발견할 수가 없고 출력 채널의 수만큼 각 track에 대한 검색을 완료했으므로 grouping을 수행하고 노드 1으로 회귀하며 이때 $c_1$의 father track에는 multiple track이 존재하고 있기 때문에 $c_1$은 complex type로 지정한다. Grouping은 stack $S_1$과 stack $S_2$의 element들을 노드 2의 출력 채널개수인 $out(v)$ 즉 3번 발췌해서 만들어진 조합 ($c_4$,3), ($c_5$,3), ($c_6$,3)을 리스트 L에 넣고 $c_4c_5c_6$은 같은 merge 노드 3을 가지고 있으므로 같은 그룹 $GR(2,3)$에 속한다. 노드 1에서 미방문 채널인 $c_2$를 따라 노드 5를 검색할 때 $c_2$를 stack $S_1$에 저장을 한다. 노드 5 에서 다시 채널$c_7$을 따라 노드 6를 방문하고 노드 5는 disperse 노드이기 때문에 $S_1$에 $c_7$을 저장한다. 노드 6에서 $c_{16}$을 따라 노드 4를 방문하면 노드 4는 이미 방문이 되었기 때문에 $S_2$에 노드 4를 저장을 하고 다시 노드 6로 회귀를 하며 이때 $c_{16}$을 simple type으로 정의한다. 노드 6에서는 미방문 채널이 없으므로 노드 5로 복귀할때 $S_2$에 노드 6를 저장하고 $c_7$을 simple type으로 정의한다. 노드 5에서 $c_8$을 따라 노드 6를 검색할 때 $c_8$을 stack $S_1$에 넣고 그리고 노드 6는 이미 방문이 된 상태이므로 노드 6를 stack $S_2$에 넣고 노드 5로 복귀하면서 $c_8$을 simple type으로 정의한다. 노드 5에서 $c_9$을 따라 노드 7을 방문하면서 $c_9$을 stack $S_1$에 넣고 노드 7에서 $c_{10}$을 따라 노드 8을 방문하면서 $c_{10}$을 stack $S_1$에 넣는다. 노드 8에서 $c_{14}$를 따라 노드 6를 검색하면 노드 6는 이미 방문했던 노드이므로 stack $S_2$에 노드 6을 저장을 하고 $c_{14}$를 simple type으로 지정한 뒤 다시 노드 8로 복귀한다. 노드 8은 미방문한 채널이 없으므로 노드 7로 회귀하고 노드 8을 stack $S_2$에 저장하며 $c_{10}$을 simple type으로 지정한다. 노드 7에서 미방문 채널$c_{11}$을 검색할때 $c_{11}$을 stack $S_1$에 넣고 노드 8은 이미 방문된 노드이므로 노드 8을 stack $S_2$에 넣는다. 마찬가지로 노드 7에서 $c_{12}$를 검색할 때 $c_{12}$를 stack $S_1$에 넣고, 노드 8은 이미 방문된 노드 이므로 stack $S_2$에 노드 8을 넣고 노드 7로 복귀한다. 노드 7에서는 더 이상의 미방문 채널을 발견할 수 없기 때문에 노드 7의 출력 채널의 갯수 $out(v)$ 즉 3번 pop해서 ($c_{10}$,8), ($c_{11}$,8), ($c_{12}$,8)을 리스트 $L$에 넣고 $c_{10}$, $c_{11}$, $c_{12}$는 같은 merge 노드 8을 가지고 있으므로 같은

그룹 $GR(7,8)$에 속한다. 노드 7에서 노드 5로 회귀하면 더 이상의 미방문 채널이 없으므로 노드 5의 출력 채널개수만큼 pop해서 $c_7$, $c_8$, $c_9$은 같은 그룹 $GR(5,6)$에 속하게 된다. 노드 5에서 노드 1로 회귀한 후 $c_3$를 따라 같은 절차로 검색하면 $c_3$는 $GR(1,4)$에 속하게 되고 $c_3$는 simple type이 된다.

### 3) 병렬 시뮬레이션 알고리즘 PAR_SIMULATION_RFN

본 절에서는 recursive feedforward network(RFN)에서 널 메시지의 숫자를 감소시키면서 노드에서 발생할 수 있는 시뮬레이션 지연을 줄일 수 있는 새로운 병렬 시뮬레이션 알고리즘 PAR_SIMULATION_RFN에 관하여 설명한다.

PAR_SIMULATION_RFN 병렬 알고리즘에서는 PR_RFN을 통하여 RFN를 전 처리함으로써 disperse 노드에서의 출력 채널들에 대한 그룹화 및 유형을 결정하고, 이에 대한 정보를 사용하여 각 disperse 노드에서는 각 그룹의 채널유형에 따라 널 메시지를 선택적으로 보내줌으로써 널 메시지의 숫자를 감소시키면서 내부 merge 노드에서 발생할 수 있는 시뮬레이션 지연시간을 단축 할 수 있게 된다. PAR_SIMULATION_RFN에서 각 메시지는 노드의 next initiation time 값과 타임 스탬프에 대한 정보를 가지고 있고, disperse 노드에서는 message의 생성 순서에 따라 번호를 부여하여 인과의 오류를 방지하게 된다[3]. 이는 잘 알려진 방식이므로 알고리즘에서 자세한 언급이 없어도 자동적으로 수행되는 것으로 간주한다. PAR_SIMULATION_RFN의 자세한 설명은 아래와 같다.

**PARALLEL ALGORITHM PAR_SIMULATION_RFN**

Input : recursive feedforward network RFN(s,t)

1. Execute PR_RFN algorithm for RFN(s,t).
2. Execute step 2.1), 2.2) and 2.3) in parallel.
   - 2.1) Repeat the following for each disperse node $p$.
     - 1) Wait for the next message.
     - 2) Place the received messages into queue, and update, for input channel of the node$q$, its channel time.
     - 3) Process the message, and generate a real message over the proper channel $c_k$.
     - 4) Generate, for each group $GR_i(p,v_i)$ to which $c_k$ belongs, a null message for each complex type channel in $GR_i(p,v_i)$ other than $c_k$ if some channels in $GR_i(p,v_i)$ are complex type.
     - 5) Generate, for each group $GR_i(p,v_i)$ to which $c_k$ does not belong, a set of null message over output channels in $GR_i(p,v_i)$ as follows:
       - a) If all channels in $GR_i(p,v_i)$ are simple types and the number of channels in $GR_i(p,v_i)$ is more than one, generate a null message over one of the channels in $GR_i(p,v_i)$.
       - b) If some channels in $GR_i(p,v_i)$ are complex type, generate a null message for each complex type channel in $GR_i(p,v_i)$.
   - 2.2) Repeat the following for each merge node.
     - 1) Wait for the next message.
     - 2) Place the received messages into proper queue.
     - 3) Update, for each input channel of the node $q$, its channel time as follows:
       - a) If there exist messages in the queue, determine the message with the lowest timestamp and assign that timestamp to the channel time.
       - b) If there exist no message in the queue for the channel $c$, compute the prediction time $t_c$ for $c$, and assign it to the channel time of $c$ if $t_c$ is smaller than the present channel time.
     - 4) If the channel with the earliest channel time has a real message, then process the message, and generate a message

over the proper channel unless node type of $q$ is sink.

2.3) Repeat the following for each simple pass node.

1) Wait for next message.

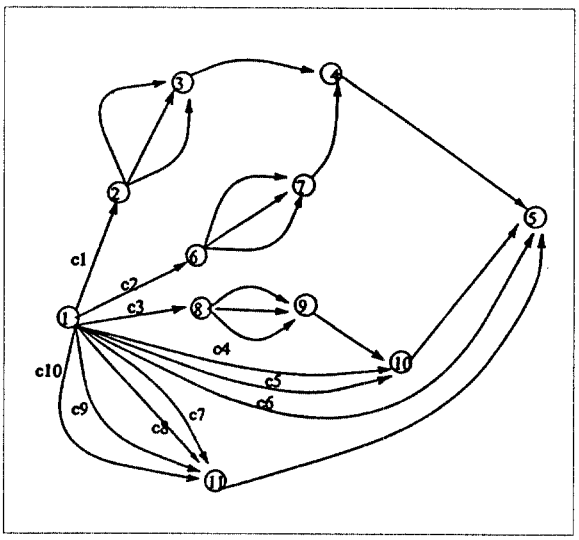
2) Place the received messages into queue.

3) Update, for input channel of the node$q$, its channel time.

4) Process the message, and generate a message over the output channel.

Step 1에서 RFN($s$,$t$)에 대해 PR_RFN 알고리즘을 수행시킨 후 step 2에서 각 노드의 시뮬레이션을 동시에 진행시키게 된다. Step 2.1), 2.2) 및 2.3) 에서는 각각 disperse 노드, merge 노드, single pass 노드에서의 시뮬레이션을 수행한다. Step 2.1)에서 disperse 노드 $p$는 입력 채널에 도착한 메시지를 대기행렬에 넣고 입력 채널 시간을 갱신한 후, 대기행렬에 있는 메시지를 처리하고 적당한 출력 채널 $c_k$를 통해 메시지를 보내는 동시에 각 그룹별로 채널 유형에 따라 널 메시지를 보낸다. $GR_i(p,v_i)$가 $c_k$를 포함하면 $c_k$를 제외한 모든 complex 유형 채널에 널 메시지를 보내고, $GR_i(p,v_i)$가 $c_k$를 포함하지 않는다면 simple type으로만 이루어 진 경우에는 임의의 한 채널에, complex type이 있다면 모든 complex type 채널에 널 메시지를 보내게 된다. Disperse 노드중 source 노드에서는 step 3), 4), 5)만을 수행하고, step 3)에서 메시지만을 생성한다. Step 2.2)에서 merge 노드 $q$는 채널을 통해서 메시지가 도착하면 각 채널의 대기행렬에 저장하고, 각 채널의 시간을 다음과 같이 갱신한다: 각 채널$c$의 대기행렬에 저장된 메시지가 있으면 메시지중 가장 최소 타임 스탬프를 $c$의 채널 시간으로 지정하고, 만약에 저장된 메시지가 없으면 모든 대기행렬에 들어있는 메시지중 최대의 next initiation time $T$와 채널$c$가 속해있는 track의 최소 지연시간 $d$를 더한 예측시간 $T+d$값을 구하여 $c$의 채널 시간으로 지정한다. 만약 최소 채널 시간 값을 가진 채널에 메시지가 도착하면 이를 처리하고 적당한 채널을 통해 메시지를 내보낸다. Message 노드중 sink 노드는 step 4에서 메시지를 생성 시키는 것을 수행하지 않는다. Step 2.3)에서 single pass 노드는 채널시간을 갱신하는 것을 제외하고는 merge 노드와 동일하다.

예를 들어 〈그림 7〉에서 RFN(1,5)에 대해 PAR_SIMULATION_RFN 알고리즘을 설명하면 다음과 같다: PR_RFN 전 처리 알고리즘에 의하여 같은 disperse 노드와 merge 노드를 공유하고 있는 track들로 구성된 그룹을 구별하고 각 채널들은 다시 complex 혹은 simple 유형의 채널들로 구분한다. 그 결과 source 노드 1에는 10개의 출력 채널들이 연




〈그림 7〉

결되어 있고, 채널$c_1$, $c_2$는 $GR_1(1,4)$, $c_3$,$c_4$,$c_5$는 $GR_2(1,10)$, $c_6$는 $GR_4(1,5)$, $c_7$,$c_8$,$c_9$,$c_{10}$은 $GR_3(1,11)$으로 그룹화가 된다. $GR_1(1,4)$에서 $c_1$및 $c_2$의 father track이 complex track이기 때문에 $c_1$, $c_2$는 complex type이고, $GR_2(1,10)$에서 $c_3$의 father track이 complex track이므로 $c_3$는 complex type, $c_4$,$c_5$의 father track이 simple track이므로 $c_4$와 $c_5$는 simple type이 된다. $GR_3(1,11)$에서 각 채널의 father track이 simple track이므로 $c_7$,$c_8$,$c_9$,$c_{10}$은 simple type이 된다. 그리고 $c_6$의 father track이 simple track이므로 $c_6$는 simple type이 되고 $GR(1,5)$에 속한다. 만약 disperse 노드 1에서 첫 번째로 발생한 리얼 메시지가 채널$c_1$을 따라 보내진다면 PAR_SIMULATION_RFN의 step 4), 5)에서 $GR_1(1,4)$은 2개의 complex track 들로 구성되어 있기 때문에 $c_2$에도 널 메시지를 보내 주고,

$GR_2(1,10)$에서는 채널$c_3$, $c_4$, $c_5$중에서 complex 유형인 채널 $c_3$에만 널 메시지를 보내며 $GR_3(1,11)$에서는 4개의 simple type 채널중 하나의 대표 채널에만 널 메시지를 보내면 된다. $GR(1,5)$에는 하나의 simple type 채널 $c_6$로만 구성되어 있기때문에 다른 merge 노드에서의 메시지 지연에는 상관없으므로 널 메시지를 보낼 필요가 없다. 이와 같은 방법으로 각 그룹의 채널들에 대하여 선택적으로 널 메시지를 보내면 해당 disperse 노드의 next initiation time 정보를 사용하여 merge 노드 4, 10, 11에서의 채널 시간의 예측이 가능하게 되어 시뮬레이션 지연을 줄일 수 있고, 모든 채널에 메시지를 보내는 대신 각 그룹별로 대표 채널에 보내줌으로써 널 메시지 사용에 따른 통신 비용(communication overhead)을 줄일 수 있다.

〈표 1〉에서는 서로 다른 노드와 채널수를 갖는 5개의 recursive feedforward network에 대하여 PAR_SIMULATION_RFN 알고리즘 및 De Vries 알고리즘을 수행시킨 결과 sink 노드에서의 사건을 처리하면서 증가된 시뮬레이션 시간의 평균 값 t와 그 시간동안 처리된 사건 수의 평균값 n을 보여주고 있다. 이때 하나의 시뮬레이션 프로세스는 하나의 프로세서에 대응한다고 가정하며, source 노드의 시뮬레이션 local time이 25가 될때까지 각각 100회씩 반복 수행 하였고, 각 시뮬레이션 수행시 source 노드에서의 사건 발생 빈도는 주어진 local time 동안 random하게 변화된다. 〈표 1〉에서 보는 바와 같이 노드와 채널의 수가 많지 않고 merge 노드 숫자가 작은 간단한 $RFN_1$에서는 PAR_SIMULATION_RFN 알고리즘이나 De Vries의 알고리즘이 t와 n에서 커다란 차이를 보이지 않는다. 즉 $RFN_1$의 sink 노드에서 사건을 처리하면서 증가되는 시뮬레이션 시간은 PAR_SIMULATION_RFN 알고리즘이 De Vries의 알고리즘에 비하여 1.6%향상되었으며, 이때 처리된 사건 수는 3.5개 증가되었을 뿐이다. 이는 $RFN_1$과 같이 메시지 지연이 발생되는 merge 노드의 숫자가 작은 경우, 각 disperse 노드에서의 메시지가 보내지는 output channel들의 숫자가 작기 때문에 disperse 노드를 출발한 real 메시지가 주어진 시뮬레이션 네트워크의 각 merge 노드에 도착할 빈도수가 높아지고, 따라서 De Vries 알고리즘에서 메시지가 각 merge 노드의 input channel에서 대기하는 시간이 줄어들기 때문이다. 그러나, 〈표 1〉에서 $RFN_2$~$RFN_5$와 같이 노드나 채널수가 크게 증가하여 내부 merge 노드수가 증가하면 PAR_SIMULATION_RFN 알고리즘을 사용했을 경우 De Vries의 알고리즘에 비해 t와 n의 값이 크게 향상된다. 예를 들어 내부 merge 노드 수가 크게 증가한 $RFN_4$를 PAR_SIMULATION_RFN 알고리즘과 De Vries의 알고리즘에 의하여 시뮬레이션을 수행하였을 경우, sink 노드에서의 시뮬레이션 시간은 PAR_SIMULATION_RFN 알고리즘의 경우 18%가 증가하였으며, 처리된 사건의 수는 72개에서 87개로 15개나 증가하였다. 또한 $RFN_5$와 같은 경우는 sink 노드에서의 시뮬레이션 시간은 57%나 증가하였으며 처리된 사건의 수는 39개가 증가하였다. 이와 같이 주어진 시뮬레이션 네트워크의 merge 노드의 숫자가 증가하게 되면 De Vries의 알고리즘의 경우 disperse 노드에서 한 그룹으로 real 메시지가 보내질 때 다른 그룹에는 null 메시지를 보내지 않기 때문에, 각 그룹의 merge 노드에 도착한 메시지가 대기하는 시간이 크게 증가를 하고, 따라서 merge 노드의 input channel의 queue에 저장된 메시지의 숫자가 증가한다. 따라서 〈표 1〉에서 보는 바와 같

〈표 1〉

| RFN | 총 노드수 | 총 채널수 | merge 노드수 | 시뮬레이션수행회수 | De Vries | | PAR_SIMULATION_RFN | |
|---|---|---|---|---|---|---|---|---|
| | | | | | t | n | t | n |
| 1 | 12 | 16 | 3 | 100 | 25.57 | 94.32 | 25.98 | 97.85 |
| 2 | 17 | 27 | 4 | 100 | 23.17 | 83.25 | 25.96 | 97.25 |
| 3 | 24 | 33 | 6 | 100 | 24.11 | 81.05 | 27.93 | 96.10 |
| 4 | 27 | 41 | 7 | 100 | 21.78 | 72.33 | 25.71 | 87.32 |
| 5 | 31 | 44 | 8 | 100 | 17.32 | 56.60 | 27.24 | 95.45 |

t : RFN의 sink 노드에서의 시뮬레이션 시간들의 평균 값

n : RFN의 sink 노드에서의 처리된 사건수의 평균 값

* Local time of the source : 25

이 merge 노드의 개수에 따른 RFN의 구조가 복잡한 네트워크에 대한 시뮬레이션을 수행할 경우 PAR_SIMULATION_RFN은 De Vries 알고리즘에 비해 sink 노드에서 처리된 사건 수 n과 그때까지 진행된 시뮬레이션 시간 t의 값이 크게 증가하여 보다 효율적이라는 것을 알 수 있다.

## 5. 결 론

시뮬레이션은 여러 분야의 실세계 현상을 보여주기 때문에 많은 응용 분야에서 유용하게 이용되고 있으나, 시스템의 규모가 복잡해지고 커짐에 따라 시뮬레이션을 수행하는데 많은 계산시간이 필요하게 된다. 이 같은 문제점을 해결하기 위하여 여러 개의 프로세서를 사용한 병렬 이산 사건 시뮬레이션에 관한 연구가 수행되어 왔다.

본 논문에서는 recursive feedforward network상에서 이전의 병렬 알고리즘에 비해 널 메시지의 숫자를 줄이는 동시에, 널 메시지를 사용하지 않음으로 인하여 발생할 수 있는 있는 시뮬레이션의 지연시간을 감소시키는 새로운 병렬 시뮬레이션 알고리즘을 제시하였다. 병렬 시뮬레이션 알고리즘의 설계를 위해 병렬 시뮬레이션 모델을 정의하고, DFS 알고리즘을 사용하여 recursive feedforward network에서 각 disperse 노드의 출력 채널들의 그룹 및 유형을 결정하는 전 처리 알고리즘 PR_RFN을 제시하였으며, 그룹 및 유형에 관한 정보를 사용하여 각 disperse 노드에서 그룹별로 채널 유형에 따라 널 메시지를 선택적으로 보내줌으로써 내부 노드에서 발생할 수 있는 시뮬레이션 지연 시간을 단축시키면서 최대한도로 널 메시지 숫자를 감소시킬 수 있는 새로운 병렬 시뮬레이션 기법을 제시하였다. 이전의 알고리즘은 Misra 알고리즘 같이 모든 채널에 널 메시지를 보내주거나, 또는 De Vries 알고리즘 같이 널 메시지를 보내지 않고 예측만 하는 방법을 사용함으로써 과도한 널 메시지의 사용및 시뮬레이션 지연을 피할 수 없었으나 본 논문에서는 채널 유형에 근거하여 모든 complex type 채널에 널 메시지를 보내줌으로써 시뮬레이션 지연을 방지하면서, 각 그룹별로 메시지를 보내줌으로써 메시지 숫자를 감소시키는 병렬 시뮬레이션 알고리즘을 설계하였고 실험 시뮬레이션을 통하여 RFN의 내부 merge 노드수가 증가하면 이전의 알고리즘에 비해 시뮬레이션 처리 능력이 향상됨을 보여주었다. 앞으로 RFN 이외에 보다 다양한 형태의 시뮬레이션 네트워크 상에서의 병렬 시뮬레이션 알고리즘의 개발이 필요하고 이에 관한 연구가 수행되어야 할 것이다.

## 참고문헌


[1] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," IEEE Transactions on Software Engineering, Vol. E-5, no. 5, pp. 440-452, Sep. 1970

[2] R. M. Fujimoto, "Time warp on a Shared Memory Multiprocessor," Trans of The Society for Computer Simulation, vol. 6, no. 3, pp. 211-239, July. 1989

[3] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," Communications ACM, Vol. 24, no. 11, pp. 198-206, nov. 1981

[4] J. Misra, "Distributed Discrete-Event Simulation," ACM Computing Surveys, Vol. 18, no. 1, pp. 39-65, Mar. 1986

[5] Chandy, K. M., Holmes, V., and Misra, J. "Distributed Simulation of Networks," Comptrs. Networks 3, 1, Feb. 1979

[6] W. Cai and S. J. Turner, "An Algorithm for Distributed Discrete-Event Simulation - The 'Carrier null Message' Approach," In: Proceedings of the SCS Multiconference on Distributed Simulation Vol. SE-5, no. 5, pp. 440-452, Sep. 1979.

[7] Ronald C. De Vries, "Reducing Null Messages in Misra 's Distributed Discrete Event Simulation Method, " IEEE Transactions on Software Engineering, Vol. 16, no. 1, Jan. 1990

[8] J, K. Peacock, J. W. Wong, and E. G. Manning, "Distributed Simulation Using a Network of Processors," Computer Networks, Vol. 3, no. 1, pp. 44-56, 1979

[9] Abraham Silberschatz, "Communication and Synchronization in Distributed Systems,"IEEE Transactions on Software Engineering, Vol.SE-5, Nov, 1979

## ● 저자소개 ●

**옥시건**
1982년 육군 사관학교 전자과 학사
1986년 국방 대학원 운영분석학과 석사
1989년~1991년 한미 연합사 운영 분석단 워게임 분석장교
1996년 고려대학교 산업공학과 박사
관심분야: 병렬 알고리즘, 병렬 시뮬레이션, 워 게임

**정창성**
1981년 서울대 전기과 학사
1985년 Northwestern Univ. 전산과 석사
1987년 Northwestern Univ. 전산과 박사
1987년~1992년 포항공대 전자계산학과 조교수
1992년~현재 고려대 전자과 부교수
1992년~현재 on Editorial review board of J. Parallel Algorithms and Applications
관심분야: 병렬 처리, 병렬 시뮬레이션