UM SIMULADOR DISTRIBUÍDO BASEADO NO PARADIGMA DE EVENTOS CONDICIONAIS

Luis Carlos Alves Pereira Quintela

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS.

Aprovada por:

Prof. Valmir Carneiro Barbosa, Ph.D.
(Presidente)

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Adriano Joaquim de Oliveira Cruz, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
DEZEMBRO DE 1992

QUINTELA, LUIS CARLOS ALVES PEREIRA
Um Simulador Distribuído Baseado no Paradigma de Eventos Condicionais
[Rio de Janeiro] 1992.
x, 67 p., 29.7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas, 1992)
Tese - Universidade Federal do Rio de Janeiro, COPPE
1. Simulação 2. Sistemas Distribuídos
I. COPPE/UFRJ II. Título (série)

À Cláudia Baroni.

Agradecimentos

Ao meu orientador, Valmir, sempre acessível.

Aos meus pais, Luis e Sônia, responsáveis pela minha formação.

Ao meu cunhado, Ruy, um exemplo.

Aos meus colegas de mestrado, Nahri, Felipe, Fernando, Hsing e Delfim, presenças constantes.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

Um Simulador Distribuído Baseado no Paradigma de Eventos Condicionais

Luis Carlos Alves Pereira Quintela

Dezembro de 1992

Orientador: Valmir Carneiro Barbosa
Programa: Engenharia de Sistemas e Computação

A tentativa de estudar o comportamento de sistemas reais complexos analiticamente esbarra nos limites restritos de modelos de representação matematicamente tratáveis. A simulação computacional destes sistemas, através do uso de programas que implementem modelos representativos, surge como alternativa viável de estudo.

O aparecimento de arquiteturas de computadores paralelas, mais particularmente arquiteturas distribuídas, dá novo impulso à área de simulação por permitir, ao oferecer maior poder de computação, a análise de sistemas mais complexos. O uso de paralelismo traz, no entanto, problemas inexistentes no âmbito seqüencial. Se antes a garantia de respeito às relações de causalidade entre os eventos era trivialmente conseguida, paradigmas de simulação devem ser criados para evitar a ocorrência de erros causais quando a simulação progride concorrentemente em espaços distribuídos.

O presente trabalho descreve a implementação do modelo conservador de simulação distribuída por abordagem de eventos condicionais, elaborado por K. M. Chandy e R. Sherman. Um simulador baseado em tal paradigma é apresentado permitindo a construção de sistemas reais quaisquer que possam ser representados por um determinado modelo. Com relação a este simulador, medidas de desempenho foram feitas através da simulação de uma rede de filas de topologia parametrizada. Utilizou-se, para a referida implementação, uma rede hipercúbica de *transputers* e a linguagem paralela *Occam 2*.

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A Distributed Simulator Based on the Conditional Event Paradigm

Luis Carlos Alves Pereira Quintela

December 1992

Thesis Supervisor: Valmir Carneiro Barbosa

Department: Programa de Engenharia de Sistemas e Computação

The attempt to study the behavior of complex real systems analytically comes up against the narrow limits of representation models that can be mathematically treated. The computational simulation of such systems, through the use of programs which implement representative models, appears as a viable study alternative.

The emergence of parallel computer architectures, particularly distributed architectures, stimulates the simulation field by allowing, as it offers computer capacity boosts, the analysis of even more complex systems. Nevertheless, the use of parallelism brings some problems which did not exist in sequential machines. If the guarantee of respect to causality relationships among events was before trivially managed, simulation paradigms must be created in order to avoid the occurrence of causal errors when the simulation progresses concurrently in distributed spaces.

This work describes the implementation of the conditonal event approach to distributed simulation conservative paradigm, elaborated by K. M. Chandy and R. Sherman. A simulator, based in such paradigm, is presented which allows the construction of logical systems for the simulation of real systems which can be represented by a specific model. Concerning this simulator, performance measures were taken through the simulation of a queuing network with parametric topology. A hypercubic transputer network and the parallel language Occam 2 were used in the implementation.

Índice

1 Introdução **1**

1.1 Objetivos 1

1.2 Motivação 2

1.3 Estruturação do Texto 4

2 Simulação **5**

2.1 Conceitos 5

2.2 Simulação Distribuída 11

2.3 Abordagem Conservadora 13

2.4 Abordagem Otimista 17

3 Abordagem por Eventos Condicionais e o Simulador **20**

3.1 Proposta 20

3.2 O Algoritmo Síncrono de Chandy e Sherman 22

3.3 O Algoritmo Assíncrono 26

3.4 Implementação do Simulador 30

3.5 Criação de Aplicações 38

4 Ambiente de Desenvolvimento **42**

4.1 O Multiprocessador NCP I e o Transputer 42

4.2 A Linguagem Occam 2 44

4.3 Ambiente de Programação 49

4.4 O Módulo de Comunicação 50

5 Resultados e Conclusões **52**

5.1 O Modelo de Teste 52

5.2 Resultados Obtidos 55

5.3 Conclusões 62

Lista de Figuras

2.1 Algoritmo Seqüencial de Simulação 8

2.2 Rede de Filas 9

2.3 Ocupação dos Servidores no Exemplo sem Preempção 10

2.4 Diagrama Tempo-Espaço 12

2.5 *Deadlock* 14

3.1 Algoritmo Síncrono 24

3.2 Transições de Estado Global 28

3.3 Estado Global no qual vale a equação 3.1 30

3.4 Estrutura do Simulador para 2 Processadores 31

3.5 Seção de Simulação 36

4.1 Hipercubo de 8 nós 43

4.2 Módulo de Comunicação 50

5.1 Modelo de Teste 53

5.2 Conexão dos *PLs* no Modelo de Teste 54

5.3 Variação do Número de Arestas com S = 2 e F = 16 57

5.4 Variação do Número de Arestas com S = 2 e F = 32 58

5.5 Variação do Número de *Switches* com A = 2 e F = 16 59

5.6 Variação do Número de *Switches* com A = 2 e F = 32 60

5.7 Variação do Número de Filas entre *Switches* com S = 4 e A = 2 61

Capítulo 1

Introdução

Este capítulo descreve, nas duas primeiras seções, o objetivo da tese e a motivação para a sua elaboração. A última seção dedica-se a guiar o leitor, demonstrando a forma como o texto está estruturado.

1.1 Objetivos

O presente estudo tem como fim apresentar a implementação de um simulador geral, baseado no paradigma por abordagem de eventos condicionais proposto por K. M. Chandy e R. Sherman em [CS89a]. O paradigma foi ligeiramente modificado com a intenção de diminuir o tráfego de mensagens necessárias para o controle da simulação. Estas modificações foram possíveis pela estratégia usada na implementação do paradigma, cujos detalhes foram escondidos do programador do sistema lógico.

O simulador permite a criação de aplicações para a simulação de quaisquer sistemas reais que possam ser representados por um modelo a ser descrito. Uma rede de filas parametrizada é usada para a medição do desempenho da implementação. Através da variação destes parâmetros de definição da rede, tenta-se identificar o perfil das aplicações para as quais se pode obter um bom desempenho do simulador. As medições feitas apontam para a alta dependência do modelo por abordagem de eventos condicionais da capacidade da aplicação de explorar o *lookahead* intrínseco ao sistema físico simulado.

A tese discute ainda aspectos relacionados com simulação paralela discreta orientada a eventos, em um ambiente distribuído. Para melhor situar o paradigma de Chandy e Sherman, serão apresentados alguns modelos de simulação distribuída encontrados na literatura específica. Esta compilação não pretende, contudo, ser exaustiva. Tem o intuito de trazer luz ao problema para que o paradigma possa ser claramente analisado, e suas virtudes e fraquezas compreendidas.

1.2 Motivação

Historicamente, o conhecimento humano tem sido representado por modelos. O modo como se percebe, se classifica e se reage a um determinado ambiente é definido pelo modelo concebido para caracterizá-lo. Diversas classes de modelos podem ser identificadas. Interessa-se, dentro do escopo definido para esse trabalho, pela classe de modelos simbólicos.

Objetos, relações e ações podem ser representados por símbolos, bastando que se faça uma associação destes com os conceitos que devem denotar. Os modelos simbólicos englobam modelos analíticos e descritivos.

Por modelos analíticos, entende-se aqueles fortemente baseados em alguma teoria com o intuito de obter resultados desejados dedutivamente, requerendo um profundo conhecimento do sistema em estudo e grande base matemática. Na elaboração de um modelo analítico, simplificações com relação ao sistema real a modelar devem ser feitas para que o modelo seja matematicamente tratável, o que limita consideravelmente o número de sistemas passíveis de estudo.

Modelos descritivos provêem uma técnica experimental para a exploração de sistemas através de processos computacionais. A esta exploração de estados e padrões de comportamento de um sistema dá-se o nome de simulação. Recorre-se a este recurso quando da necessidade de avaliação do comportamento de um determinado problema, para o qual não foi possível estabelecer um modelo analítico. Muitas foram as linguagens desenvolvidas e implementadas especificamente para a criação de programas de simulação, como GASP, SIMSCRIPT, GPSS e SIMULA.

Uma dificuldade, porém, aparece na tentativa de simulação de sistemas grandes, cuja demanda por computação consome porções excessivas de tempo de processamento em máquinas seqüenciais. Face a isto, busca-se uma exploração do paralelismo intrínseco a muitos sistemas reais através do particionamento destes em unidades que possam ser simuladas concorrentemente. Tome-se, como exemplo, a simulação do comportamento de bolas de sinuca sobre uma mesa em um intervalo definido de tempo. Dadas as coordenadas e velocidades vetoriais iniciais

das bolas, deseja-se observar as movimentações e choques das bolas com outras bolas e com as bordas que delimitam a mesa. É extremamente natural uma abordagem do problema na qual a mesa é subdividida em polígonos contíguos, mutuamente exclusivos, talvez quadrados de mesmo tamanho. Para cada partição poderia ser alocada uma unidade computacional que simularia o comportamento do sistema naquele espaço restrito.

A paralelização mostrada é tão natural quanto simplista, na medida em que não considera os problemas de causalidade que advêm do controle distribuído da simulação. De acordo com o exemplo proposto, cada unidade teria conhecimento somente do que ocorre em sua região. Esta visão limitada pode conduzir a erros, como o de se estimar que uma bola cruza uma fronteira entre duas regiões sem o conhecimento de que uma outra bola, vinda de uma região qualquer, pode chocar-se com ela, desviando-a do curso anteriormente previsto. Abstraindo-se, no entanto, desta dificuldade, tem-se uma noção do potencial de paralelismo que pode ser aproveitado em uma simulação.

Alguns modelos[1] têm sido propostos com a intenção de permitir a elaboração de simulações em ambientes paralelos, particularmente em sistemas distribuídos. Em tais ambientes, a falta de uma base de tempo global gera uma situação na qual o respeito às relações de causalidade entre os eventos da simulação não pode ser trivialmente mantido, como o seria em uma máquina seqüencial.

Pode-se dividir os modelos de simulação distribuída em dois grupos, diferentes exatamente pelo modo como mantêm as relações causais entre os eventos. O primeiro engloba os modelos conservadores, que implementam um método para garantir que erros de causalidade nunca ocorram. Os modelos otimistas, que constituem o segundo grupo, deixam a simulação fluir livremente para, quando da detecção de uma inconsistência causal, estornar a computação errada, dando prosseguimento à simulação a partir do ponto anterior ao erro.

O paradigma de Chandy e Sherman retoma a discussão sobre abordagens conservadoras, de certa forma ofuscadas pelos bons resultados obtidos com a proposta otimista de *Timewarp*, por Jefferson ([Jeffe85]). Ele se utiliza de conceitos de simulação seqüencial para criar um protocolo de seleção de eventos seguros a serem processados. Este protocolo garante que a simulação progride sempre sem a necessidade do envio de mensagens nulas, estratégia da qual lança mão o modelo conservador clássico ([CM79]) para a prevenção de deadlock, e que tende a sobrecarregar o sistema.

[1]Note que o termo "modelo" é usado agora no sentido de um protocolo para garantir o andamento correto de uma simulação distribuída, e não como modo de representação de um sistema real.

1.3 Estruturação do Texto

Este trabalho está dividido em cinco capítulos, sendo o primeiro a introdução já vista. O segundo capítulo atem-se à apresentação de conceitos de simulação. Discute os problemas enfrentados quando da paralelização de uma simulação e mostra os principais modelos, otimistas e conservadores, já propostos como forma de resolvê-los.

O capítulo seguinte é reservado para o paradigma de Chandy e Sherman e para o simulador que nele se baseia. Detalhes da implementação deste e o modo de utilizá-lo são abordados.

O ambiente de desenvolvimento do simulador é apresentado no capítulo 4, que comenta as características do multiprocessador utilizado no desenvolvimento, bem como do *transputer* T800. O capítulo discute ainda aspectos concernentes à linguagem de programção escolhida, *Occam 2*, e seu ambiente de programação TDS2 (*Transputer Development System*). Refere-se também ao *processador* de comunicação, uma camada de *software* na qual se apóia o simulador para a entrega de mensagens entre processadores.

Finalmente, o último capítulo descreve o modelo de teste usado para a medição do desempenho do simulador, listando os resultados obtidos. A conclusão da tese finaliza esse capítulo e, conseqüentemente, o trabalho.

Capítulo 2

Simulação

Quatro seções compõem este capítulo. A primeira procura introduzir os conceitos da área de simulação, servindo de base para a seção seguinte, que os discute para um ambiente distribuído.

As duas últimas seções listam alguns modelos de simulação distribuída, a primeira tratando das abordagens conservadoras e a seguinte das abordagens otimistas.

2.1 Conceitos

Alguns aspectos devem ser considerados na caracterização de métodos de simulação. Esta seção se ocupa com a apresentação de conceitos fundamentais que serão usados no decorrer do trabalho. Começa-se por introduzir o conceito de tempo de simulação.

> **Definição.** *Tempo de Simulação.* Em [PWM79], define-se tempo de simulação como a abstração do tempo real, no sentido em que um estado do sistema real em um dado tempo real corresponderá a um estado abstrato do sistema de simulação no tempo de simulação correspondente.

Identificam-se duas formas principais na modelagem de sistemas de simulação. Em ambas, o progresso da análise do comportamento do sistema é ditado pelo tempo de simulação. No caso em que o tempo é avançado em fatias pequenas, iterativamente, levando o sistema a um novo estado a cada fatia, tem-se a simulação contínua orientada por tempo. Neste método, relações

entre objetos que compõem o sistema real são refletidas em taxas de mudança das variáveis que descrevem o estado do sistema. Equações diferenciais são usadas para representar estas taxas e poderiam, a princípio, ser resolvidas analiticamente. Técnicas computacionais, porém, são preferíveis dado o esforço envolvido na solução de equações diferenciais.

No segundo método — simulação discreta orientada a eventos — o tempo de simulação avança, em saltos não constantes, para pontos discretos nos quais houve uma transição de estado do sistema, ou seja, a ocorrência de um evento. Neste caso, nada de relevante ocorre entre duas transições de estado. Cada método possui suas aplicações específicas. Para a simulação contínua, cita-se como aplicação típica a análise do fluxo de calor em um corpo tridimensional. Uma rede de filas computacionais é um bom exemplo de sistema a ser modelado pelo método de simulação discreta. Mais detalhes a respeito dos dois métodos podem ser encontrados em [Kreut86].

De interesse desta tese são as simulações discretas orientadas a eventos de sistemas reais que possam ser representados por sistemas físicos, de acordo com a definição a seguir.

Definição. *Sistemas Físicos.* Define-se *sistema físico* como um conjunto de processos físicos (*pf*) independentes que interagem através da troca de mensagens, cada qual representando um componente do sistema real a ser simulado.

As mensagens trocadas pelos processos físicos representam eventos do sistema real que ocorrem nos *pf*'s de destino das mensagens. A cada evento está associado um rótulo de tempo, indicando o momento de sua ocorrência. Esta característica cria uma relação de dependência entre os eventos. Claramente, nenhum evento pode acontecer antes que todos os eventos dos quais ele depende tenham ocorrido, exigindo que a relação de dependência seja uma ordenação parcial.

De acordo com Misra em [Misra86], os sistemas físicos obedecem as seguintes condições:

1. *Realizabilidade* – Toda mensagem enviada por um processo físico no tempo t é uma função do seu estado inicial, t, e das mensagens por ele recebidas até t, inclusive. Esta propriedade estabelece simplesmente que um processo não envia mensagens baseado na expectativa da chegada de mensagens no futuro.

2. *Preditibilidade* – Para todo ciclo de n processos físicos $pf_0, \ldots, pf_{n-1}$, onde pf_i envia mensagens para pf_{i+1} e as recebe de pf_{i-1} (considerando a aritmética dos subscritos em módulo n), existe um pf no ciclo e um número real $\epsilon, \epsilon > 0$, tal que as mensagens enviadas pelo pf ao longo do ciclo podem ser determinadas até o tempo $t+\epsilon$, dado o conjunto de mensagens

que o pf recebe até t, inclusive. A condição é requerida para que seja evitada a situação na qual uma mensagem enviada em t é função de si própria.

Para ilustrar as condições acima, toma-se o exemplo de um circuito lógico digital. Os processos físicos são as portas lógicas e as mensagens trocadas indicam as variações de sinal nos fios que interconectam as portas.

Para cada porta, determina-se um atraso de 5 unidades de tempo para a produção de um sinal de saída, a partir dos sinais de entrada. Como uma porta só produz uma saída após a chegada dos sinais de entrada, então a propriedade de *Realizabilidade* é satisfeita. Além disso, dada a entrada de uma porta no tempo t, sua saída pode ser prevista até $t + 5$. Vale, portanto, a propriedade de *Preditibilidade* para qualquer ciclo de processos físicos do sistema.

O sistema físico conceituado será simulado por um sistema lógico, segundo a definição seguinte.

Definição. *Sistemas Lógicos.* Considere um conjunto de processos lógicos (pl), tal que haja uma relação unívoca entre estes e os processos físicos do sistema físico. Espera-se de um pl que ele produza uma seqüência $\{(t_0, m_0), \ldots, (t_{n-1}, m_{n-1})\}$, onde $m_0, \ldots, m_{n-1}$ são as mensagens enviadas pelos processos físicos nos tempos $t_0, \ldots, t_{n-1}$, com $t_0 \leq \ldots \leq t_i \leq \ldots \leq t_{n-1}$ ([Misra86]). Cada pl deve possuir ainda um grupo de variáveis para armazenar o seu estado na simulação. A este conjunto de pl's dá-se o nome de *sistema lógico.*

Apresenta-se agora o algoritmo seqüencial de simulação, a ser seguido pelo sistema lógico. Antes porém, definem-se duas estruturas de dados essenciais gerenciadas pelo algoritmo:

- *Relógio* – Uma variável do tipo real, monotonicamente não-decrescente, que indica o tempo até onde o sistema lógico progrediu na simulação. Se *Relógio* $= r$, então todas as mensagens associadas a um tempo $t < r$ já foram entregues. A variável implementa o conceito de tempo de simulação, já definido.

- *Lista de Eventos* – Uma seqüência de tuplas (t_i, e_i, o_i, d_i), organizada pela ordem não decrescente dos tempos t_i, onde e_i é um texto que caracteriza o evento, o_i a identificação do processo lógico origem do evento e d_i a identificação do processo lógico destino. Uma tupla pertencente à lista e associada ao tempo t_i é enviada ao seu processo lógico destino se o processo origem da tupla não receber nenhuma mensagem num tempo t qualquer, tal que *Relógio* $\leq t < t_i$.

Relógio := 0;
Lista de Eventos := $\{(t_i, e_i, o_i, d_i) \mid e_i$ caracteriza um evento inicial,
escalonado por o_i para ocorrer em d_i, no tempo $t_i\}$;

while *Lista de Eventos* $\neq \emptyset$ **do**
begin
Selecione a tupla na *Lista de Eventos* com menor rótulo de tempo t_i associado;
Simule o recebimento da tupla no tempo t_i pelo processo lógico destino;
Relógio := t_i;
end;

Figura 2.1: Algoritmo Seqüencial de Simulação

Segue-se a descrição do algoritmo seqüencial, listado na figura 2.1. Inicialmente, atribui-se ao *Relógio* o valor zero. A *Lista de Eventos* começa com um determinado número de tuplas iniciais. No laço de iteração, remove-se a tupla com menor valor de rótulo de tempo associado entregando-a ao processo lógico de destino. A chegada de uma mensagem a um *pl* simula a ocorrência de um evento no sistema físico.

O processamento de um evento por um *pl* ocasiona uma mudança de estado e gera um número maior ou igual a zero de mensagens que são inseridas na *Lista de Eventos.* Este processamento pode determinar, também, o cancelamento de mensagens contidas na lista. Isto ocorre quando uma expectativa de comportamento futuro da simulação, ditada por eventos contidos na *Lista*, é frustrada pela ocorrência de um determinado evento. Note porém, que se t_c é o rótulo de tempo da mensagem m cancelada e t_r o tempo da mensagem recebida, então *Relógio* $\leq t_r < t_c$. Em seguida ao processamento do evento, é atribuído à *Relógio* o valor do rótulo de tempo da mensagem recebida. O algoritmo segue, iterativamente, até que a *Lista de Eventos* esteja vazia.

Deve-se perceber a importância da seleção do evento com menor rótulo de tempo (E_{min}) na *Lista de Eventos* para ser processado a cada passo do algoritmo. Suponha que, em um dado momento, seja escolhido um evento E cujo rótulo de tempo é maior que o tempo de E_{min} e que a simulação da ocorrência deste evento altere o estado do sistema. Ao ser processado, E_{min} encontraria um estado modificado por E, gerando a situação absurda na qual um evento futuro influencia um evento passado.

O funcionamento do algoritmo pode ser exemplificado pela simulação de um sistema físico, representando um sistema real composto por uma rede de filas, conforme mostra a figura 2.2.

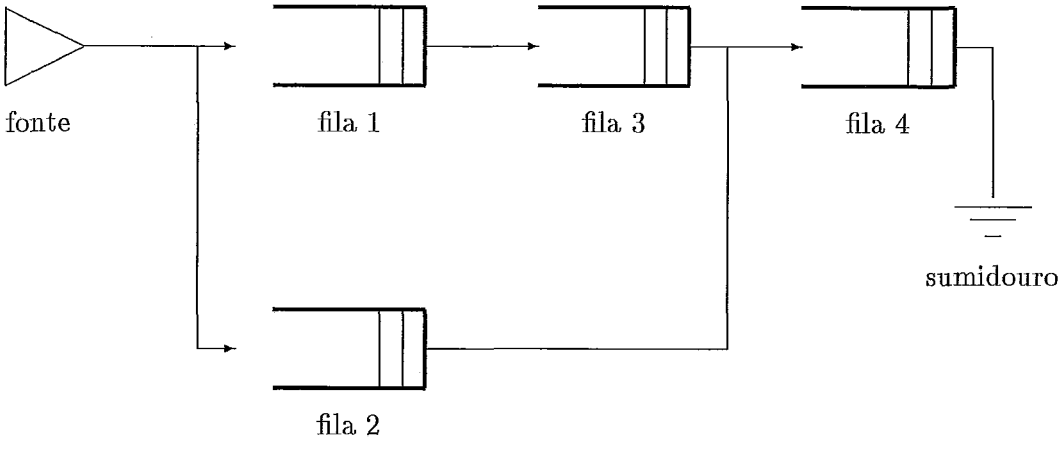

Figura 2.2: Rede de Filas

São processos físicos os elementos destacados no desenho, quais sejam: fonte, que encaminha *jobs*, em tempos aleatórios, às filas 1 ou 2, dependendo de probabilidades associadas a suas duas arestas de saída; as filas 1 a 4, que têm tempos de serviço de 4, 3, 5 e 2 unidades, respectivamente, independentes dos *jobs*; e o sumidouro, que retira do sistema os *jobs* recebidos.

Se a fonte envia à fila1 *jobs* nos tempos 1 e 7, e à fila2 nos tempos 3 e 8, então espera-se que o sistema lógico que simula a rede de filas produza a seqüência de tuplas (t_i, e_i, o_i, d_i) abaixo:

{(1,job1,fonte,fila1), (3,job2,fonte,fila2), (5,job1,fila1,fila3), (6,job2,fila2,fila4),
(7,job3,fonte,fila1), (8,job4,fonte,fila2), (8,job2,fila4,sumi.), (10,job1,fila3,fila4),
(11,job3,fila1,fila3), (11,job4,fila2,fila4), (12,job1,fila4,sumi.), (14,job4,fila4,sumi.),
(16,job3,fila3,fila4), (18,job3,fila4,sumi.)}

A ocupação dos servidores está ilustrada pela figura 2.3. Dada a seqüência de tuplas, é possível afirmar que, em determinado momento da simulação, as estruturas de dados *Relógio* e *Lista de Eventos* têm a seguinte configuração:

Relógio = 11

Lista de Eventos = {(11,job4,fila2,fila4), (12,job1,fila4,sumi.), (16,job3,fila3,fila4)}

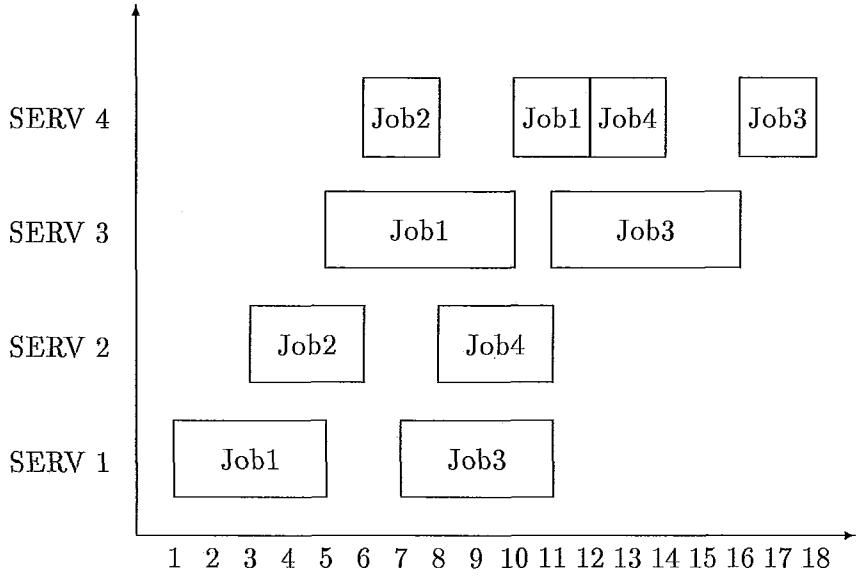

Figura 2.3: Ocupação dos Servidores no Exemplo sem Preempção

Esta situação corresponde ao estado da simulação posterior ao processamento do evento representado pela tupla (11,job3,fila1,fila3), que indica a chegada do job3 na fila3 no tempo 11, originário da fila1.

No próximo passo do algoritmo, a tupla (11,job4,fila2,fila4) seria removida da lista e processada, simulando a ocorrência do evento de chegada do job4 à fila4 no tempo 11. O servidor da fila4 está ocupado do tempo 10 ao tempo 12 servindo o job1 (a saída deste está indicada na *Lista de Eventos* pela tupla (12,job1,fila4,sumi.)) e portanto, só dará início ao serviço do job4 no tempo 12.

Suponha agora, que o job4 tem prioridade maior que o job1. Neste caso, o job1 ocupa o servidor do tempo 10 ao tempo 11, quando então é preemptado pela chegada do job4. Este é servido imediatamente por ser prioritário, saindo no tempo 13. Só então o job1 é reescalonado, ocupando o servidor por mais uma unidade de tempo. Considerando a situação descrita, o processamento do evento representado pela tupla (11,job4,fila2,fila4) implicaria na remoção da tupla (12,job1,fila4,sumi.) e na inserção de duas outras na *Lista de Eventos*, representado as partidas dos *jobs* 1 e 4 da fila4. A configuração das estruturas, posterior ao processamento, seria:

Relógio = 11

Lista de Eventos = {(13,job4,fila4,sumi.), (14,job1,fila4,sumi.), (16,job3,fila3,fila4)}

É importante notar que, quando a mensagem indicando a saída do job1 da fila4 no tempo 12 é removida, vale a relação *Relógio* $= 11 \leq t_r = 11 < t_c = 12$, onde t_r é o tempo do evento que causou a remoção e t_c o tempo da mensagem cancelada.

2.2 Simulação Distribuída

Por sistema distribuído entende-se um conjunto de elementos processadores espacialmente espalhados que não compartilham memória e que estão interconectados por canais segundo uma determinada topologia, através dos quais se comunicam pela troca de mensagens.

Ao considerar-se uma simulação que segue o paradigma anteriormente descrito em um sistema distribuído de acordo com a definição acima, conclui-se que a grande oportunidade de aproveitamento do paralelismo advém do processamento concorrente de eventos nos diversos processadores. É importante notar, no entanto, que as relações de dependência entre os eventos devem ser mantidas. Isto pode ser garantido se a simulação seguir a *Restrição de Causalidade Local* ([Fujim90]), de acordo com a definição abaixo:

> **Definição.** *Restrição de Causalidade Local.* Uma simulação discreta orientada a eventos, consistindo de um conjunto de processos lógicos, obedece à restrição de causalidade local se, e somente se, cada processo lógico processar eventos na ordem não decrescente dos rótulos de tempo.

Num ambiente seqüencial, a restrição é respeitada através da seleção, a cada passo do algoritmo, do evento com menor rótulo de tempo. Logicamente, esta estratégia, quando usada para um sistema distribuído, resultaria em um disperdício computacional, pois seqüencializaria a simulação. Em tal sistema, dois eventos poderiam ser processados em paralelo desde que não houvesse uma relação de dependência entre eles. Esta relação, simbolizada por "$\rightarrow$", é definida como se segue ([Lampo78]):

> **Definição.** *Relação "$\rightarrow$" de Dependência.* Se dois eventos a e b ocorrem num mesmo processo lógico, e tempo de simulação de a é menor que tempo de simulação de b, então $a \rightarrow b$. Se dois eventos a e b ocorrem em processos lógicos distintos, e o processamento de a resulta na geração de b, então $a \rightarrow b$. Além disso, se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$.

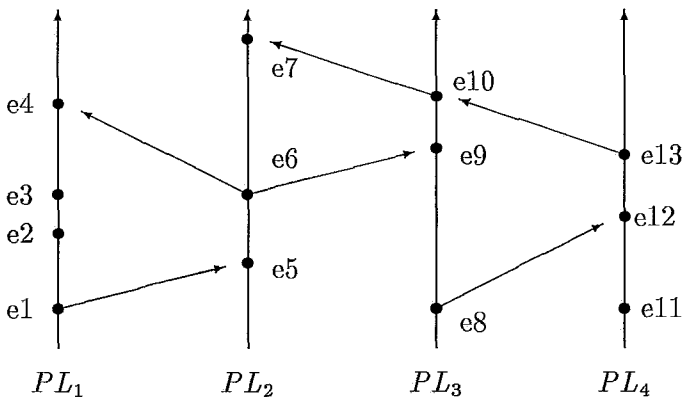

Figura 2.4: Diagrama Tempo-Espaço

Para um evento a, $a \not\to a$, e portanto a relação "$\to$" é uma ordenação parcial no conjunto de eventos da simulação. E se para dois eventos a e b, $a \not\to b$ e $b \not\to a$, então a e b são concorrentes e podem ser processados em paralelo.

A definição pode ser melhor entendida se tomarmos a figura 2.4. Nesta, a direção vertical representa o tempo de simulação e a direção horizontal o espaço. Os pontos denotam eventos e os eixos verticais os processos lógicos. As linhas inclinadas indicam o fluxo de mensagens, no sentido em que o processamento de um evento gera a mensagem que escalonará um segundo evento no processo lógico de destino desta. No diagrama, se $a \to b$, então pode-se caminhar de a para b deslizando-se pelos eixos verticais no sentido do tempo e pelas linhas de mensagens.

Vamos considerar, na figura, que o evento e_1 ocorre no tempo de simulação 5 e e_6 ocorre no tempo 10. Se e_1 ainda não foi processado, então a mensagem que escalona e_5 no processo lógico PL_2 ainda não chegou. Suponha que e_6 já está na lista de eventos de PL_2. Embora e_6 tenha o menor rótulo de tempo na lista, processá-lo implicaria num erro de causalidade, pois pela definição $e_1 \to e_5$ e $e_5 \to e_6$.

A dificuldade maior de uma simulação distribuída surge na identificação da relação de dependência entre eventos. Os termos paradigma e modelo de simulação distribuída serão usados indistintamente neste texto para caracterizar os diferentes modos de se garantir o progresso correto da simulação. Os modelos de simulação distribuída são divididos em dois grupos principais: dos modelos conservadores e dos otimistas.

Os primeiros evitam a ocorrência de erros de causalidade através da determinação, de acordo com alguma estratégia, do *momento seguro* para processar um evento, ou seja, o momento quando todos os eventos dos quais este depende já foram processados. O segundo grupo, dos modelos otimistas, se baseia numa abordagem de detecção e recuperação dos erros causais. Estes dois grupos serão detalhados nas duas seções seguintes.

2.3 Abordagem Conservadora

Os modelos de simulação distribuída classificados como conservadores devem, conforme já dito, processar um evento somente quando este é considerado seguro, ou seja, tem-se a certeza de que nenhum evento chegará posteriormente, cujo rótulo de tempo seja menor que o do evento escolhido para processamento. Suponha que um determinado processo lógico PL_1 possua um evento e_1 em sua fila de eventos cujo rótulo de tempo t_1 é o menor da fila. Se, aliado a este fato, PL_1 conseguir, seguindo alguma estratégia, determinar que nenhum outro evento chegará posteriormente por um de seus canais de entrada com tempo menor que t_1, então e_1 pode ser processado com segurança. A escolha de eventos seguros para processamento garante que a restrição de causalidade local é respeitada.

Alguns modelos conservadores são bloqueantes, no sentido em que, se não há nenhum evento seguro para ser processado, então o processo lógico deve se bloquear. Este fato pode levar a simulação a uma situação de *deadlock*. Os modelos bloqueantes fazem face a este problema através de duas maneiras distintas. A primeira evita que o *deadlock* ocorra pela utilização de um mecanismo de prevenção. O segundo modo consiste em detectar a ocorrência de *deadlock* e usar uma estratégia qualquer de recuperação.

Um dos primeiros modelos desenvolvidos ([CM79]) baseava-se num mecanismo de prevenção de *deadlock*. Neste, é exigido que os processos lógicos que compõem o sistema conheçam, estaticamente, os canais pelos quais eles podem receber ou enviar mensagens para outros processos lógicos. No modelo, a seqüência de eventos que flui em um determinado canal está na ordem não-decrescente dos rótulos de tempo. Desta forma, o tempo de um evento que chega a um PL por um canal é um limite inferior para os tempos dos próximos eventos que chegarão futuramente pelo mesmo canal.

A cada canal de entrada está associada uma fila constituída dos eventos recebidos por aquele canal, dispostos pela ordem de chegada (FIFO). Atribui-se, ainda, um relógio a cada canal de entrada, cujo valor será o tempo do evento na frente da fila correspondente ao canal, ou o tempo do último evento por este recebido caso a fila esteja vazia. O algoritmo usado pelo modelo segue selecionando o canal cujo relógio tem o menor valor entre os relógios de todos

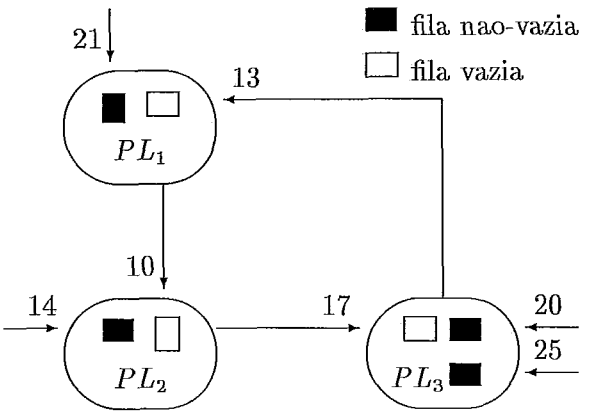

Figura 2.5: *Deadlock*

os canais de entrada de um PL e processando o evento na frente da fila correspondente a este canal. Se esta fila está vazia, então o processo bloqueia-se, à espera da chegada de mensagens que o retirem deste estado. Se um ciclo de canais existir, tal que seus relógios tenham valor mínimo nos PLs correspondentes e tal que suas filas estejam vazias, então ocorre uma situação de *deadlock*, conforme mostra a figura 2.5.

Nesta nota-se que, para todos os processos lógicos, a fila associada ao canal de entrada com menor valor de relógio — indicado pelo número que rotula a seta representando o canal — está vazia. Todos os processos contidos na figura bloqueiam-se, portanto. Como os ditos processos configuram um ciclo, tem-se uma situação de *deadlock*.

Para evitar este caso, o modelo se utiliza de mensagens nulas, assim chamadas porque não escalonam nenhum evento, funcionando apenas como mensagens de sincronização. As mensagens nulas são rotuladas com um valor de tempo e têm o seguinte significado. Uma mensagem nula com tempo t que chega a um processo lógico PL_2 enviada por PL_1 é a garantia de que PL_1 **não** mandará mais eventos para PL_2 cujos tempos sejam menores que t.

O rótulo de tempo de uma mensagem nula a ser enviada por um processo é função dos relógios associados aos canais de entrada e do incremento mínimo de tempo ϵ[1] dos eventos processados pelo PL. Sempre que terminar o processamento de um evento, um PL computa o

[1] Este incremento ϵ significa que os eventos gerados a partir do processamento de um evento associado ao tempo t terão rótulos de tempo maiores ou iguais a $t + \epsilon$.

mínimo entre os relógios dos canais de entrada e soma ϵ a este valor. O tempo resultante da soma será o rótulo da mensagem nula a ser enviada por cada canal de saída.

Este esquema evita a ocorrência de *deadlock* conquanto não exista um ciclo de PLs tal que a soma dos incrementos mínimos ϵ neste ciclo seja zero. Esta condição limita o número de sistema físicos que podem ser simulados pelo modelo.

As mensagens nulas provaram ser um fator de sobrecarga do sistema. Face a isto, surgiram outros modelos que as utilizavam somente sob demanda. Sempre que um processo lógico chega a uma situação de bloqueamento, ele envia pedidos de limites inferiores de tempo de futuros eventos aos PLs vizinhos. Esta estratégia diminui o tráfego de mensagens, que é comprovadamente um fator crítico em sistemas distribuídos.

Em [CM81], Chandy e Misra descrevem um modelo similar ao modelo acima. A estratégia para determinação do evento seguro é a mesma, porém o modelo não implementa um mecanismo para prevenção de *deadlock*, deixando que este ocorra. Um processo especial, o controlador, é usado para detectar o *deadlock* e quebrá-lo. Este processo computa, quando da detecção do problema, uma valor t_{kr} que corresponde ao mínimo dos tempos dos próximos eventos a serem enviados pelos PLs que compõem o sistema lógico. O evento cujo rótulo de tempo é igual a t_{kr} é um limite inferior para todo o sistema, sendo considerado, obviamente, seguro. Desta forma, o PL_k é informado pelo controlador que pode enviá-lo para o PL_r, quebrando a situação de *deadlock.*

Pode ser encontrada, em [Lubac89], a descrição de um modelo conservador que se utiliza de janelas móveis de tempo de simulação para reduzir a computação necessária para determinar quando é seguro processar um evento. O limite inferior da janela é definido como sendo o menor rótulo de tempo de todos os eventos não processados do sistema. Suponha, por exemplo, que seja definida uma janela do tempo de simulação t_1 ao tempo t_2. Seja t o mínimo incremento possível no sistema, tal que $2t \leq t_2 - t_1$. Então não é possível que um evento não processado em um determinado PL_i cause o escalonamento de um evento em um PL_j dentro da janela de tempo definida, para qualquer PL_j que não seja vizinho de PL_i, onde vizinhos de PL_i são os processos para os quais ele pode escalonar um evento. Desta forma, um processo lógico limita o número de eventos não processados a investigar para a determinação de um evento seguro aos eventos pertencentes às listas dos processos vizinhos.

Um ponto importante a considerar é o modo de determinação da largura da janela. Se esta é pequena, poucos eventos estarão disponíveis para execução em paralelo. Além disso, se a janela for grande, de pouco adiantará para limitar o espaço de análise dos eventos não processados. Mais ainda, se a largura da janela é menor que o menor intervalo entre os tempos

de quaisquer dois eventos da simulação, então todos os eventos cujos tempos caiam nos limites da janela podem ser processados concorrentemente, conforme afirmado em [SBW88]. É óbvio que o tamanho apropriado da janela móvel de tempo só pode ser estabelecido com conhecimento específico do sistema físico a ser simulado.

Ghosh et al. propõem, em [GDM91], uma outra abordagem, baseada na construção de uma rede acíclica de fluxo de dados a partir do grafo original formado pelos processos físicos e seus canais de interconexão. Esta rede é usada para computar mudanças incrementais de *lookahead*[2] ou informações referentes a dependências dinâmicas de dados. As computações executadas pela rede de fluxo de dados e pela simulação em si progridem concorrentemente. Valores calculados na rede são passados aos processos da simulação para que estes decidam quais eventos são seguros para serem processados.

Um modelo baseado na distância entre processos lógicos está descrito em [Ayani89]. O conceito de distância entre dois processos, PL_1 e PL_2, é definido como sendo a diferença entre o tempo de ocorrência de qualquer evento possível em PL_1 e o tempo do efeito causal (novo evento escalonado) em PL_2, resultante do processamento deste evento. Dois eventos e_1 e e_2, ocorrendo respectivamente em PL_1 e PL_2, são considerados concorrentes se: (1) o processamento de e_1 não afeta e_2; ou (2) $t(e_2) > t(e_1)$ e $0 < t(e_2) - t(e_1) < d_{12}$, onde $t(e)$ é o tempo de simulação do evento e, e d_{ij} a distância entre dois processos lógicos PL_i e PL_j, como definido. A segunda condição estabelece que qualquer evento gerado pelo processamento de e_1 para ser escalonado em PL_2, deve ter tempo de simulação maior que $t(e_2)$.

O algoritmo seguido por este modelo é síncrono, dividido em três fases. Na primeira, selecionam-se os eventos na frente das filas correspondentes aos PLs. Na segunda fase, identifica-se o paralelismo existente para o processamento dos eventos selecionados, usando-se as duas condições estabelecidas acima. Uma vez identificados os eventos concorrentes, passa-se à terceira fase do algoritmo, na qual os eventos escolhidos são processados.

Pelos modelos descritos nesta seção, pode-se ter uma noção da importância do *lookahead* para a determinação dos eventos seguros para processamento em estratégias conservadoras. Se nenhum *lookahead* pode ser determinado, através da análise de características específicas do sistema físico simulado, então o processamento do evento com menor rótulo de tempo no sistema lógico pode afetar qualquer outro evento pendente, obrigando a uma simulação seqüencial.

A crítica mais contundente que se faz aos protocolos conservadores relaciona-se com o

[2] A capacidade de previsão do que poderá ocorrer no futuro da simulação é conhecida como *lookahead*. Se o relógio de um PL tem valor t e o mínimo incremento de tempo no processamento de um evento é ϵ para o mesmo PL, então $t + \epsilon$ é um limite inferior para o tempo do próximo evento a ser enviado por este PL. Esta situação resulta em *lookahead* de tamanho ϵ.

fato de que muito do paralelismo possível no processamento de eventos é desperdiçado. A próxima seção descreve uma classe diferente de modelos que tenta explorar mais profundamente este paralelismo, mas com a desvantagem de ser obrigada a estornar computações errôneas em decorrência do excesso de otimismo.

2.4 Abordagem Otimista

Em oposição às estratégias conservadoras, modelos de simulação distribuída otimistas não selecionam eventos seguros para serem processados. Sua atenção é voltada para a detecção e correção de erros causais gerados pelo processamento de eventos na ordem incorreta dos rótulos de tempo. Como conseqüência do processamento de eventos ainda não identificados como seguros, os modelos otimistas tendem a aproveitar melhor o paralelismo em situações onde um erro de causalidade poderia ocorrer, porém freqüentemente não acontece.

O modelo de simulação otimista mais conhecido e mais pesquisado foi proposto por Jefferson em [Jeffe85]. Algumas variações sobre o modelo original de *Timewarp* foram apresentadas, conforme se discutirá posteriormente. Por esta estratégia, duas filas de eventos são mantidas em cada processo lógico. Uma, de entrada, contendo os eventos já recebidos e não processados e uma segunda, de saída, com uma cópia das mensagens enviadas pelo processo escalonando eventos em outros PLs. Um processo lógico segue retirando eventos da fila de entrada, processando-os e enviando os eventos gerados para os processos de destino, armazenado uma cópia das mensagens mandadas. Um erro causal é detectado quando o próximo evento a ser processado tem rótulo de tempo menor que o relógio local do PL. Após a constatação do erro, o estado do processo no momento imediatamente posterior ao processamento do evento de maior tempo anterior ao tempo do evento que causou o erro é restaurado, e as mensagens enviadas durante a computação indevida são canceladas. A este procedimento de estorno dá-se o nome de *Rollback*.

Para que um estado possa ser restaurado durante um *rollback*, é necessário que um processo lógico continuamente salve o seu estado. O cancelamento de mensagens já enviadas é feito mandando-se as cópias destas, armazenadas na fila de saída, com sinal negativo. Estas *anti-mensagens* têm o efeito de aniquilar suas correspondentes positivas.

Pode ser provado que a simulação por *Timewarp* progride sempre. Vale lembrar que o evento com menor valor de rótulo de tempo é seguro para ser processado. No modelo de *Timewarp*, este valor é denominado GVT (*Global Virtual Time*), sendo utilizado para a consumação de operações irreversíveis e para a liberação de memória. Nenhum *rollback* será feito para um tempo de simulação anterior ao valor do GVT, e portanto estados de processos lógicos gravados em tempos menores podem ser descartados, assim como as cópias de mensagens que não mais

serão canceladas. Da mesma forma, operações não estornáveis como gravação em disco podem ser realizadas se estão associadas a tempos de simulação anteriores ao GVT.

Como variações à estatégia clássica de *Timewarp*, cita-se o cancelamento preguiçoso (*lazy cancelation*) e a reavaliação preguiçosa (*lazy reevaluation*). Estas propostas partem do princípio de que nem sempre uma computação realizada na ordem contrária dos rótulos de tempo é errônea. Suponha que após um *rollback* o reprocessamento de eventos na ordem correta gere as mesmas mensagens anteriormente enviadas e cujas cópias estão armazenadas na fila de saída. Neste caso, estas mensagens não precisariam ser canceladas. Considerando esta situação, a estratégia de cancelamento preguiçoso compara as mensagens geradas pelo reprocessamento posterior a um *rollback* com as cópias das mensagens já enviadas, evitando o cancelamento de eventos corretos. A reavaliação preguiçosa baseia-se nas mesma constatação, porém com relação ao estado do processo lógico. Críticos das duas variações apontam para o tempo perdido com a comparação de mensagens e estados, permitindo que a computação otimista errônea progrida pelo sistema.

No entanto, as estratégias preguiçosas podem permitir que a simulação execute por um tempo inferior ao tempo de execução do caminho crítico ([SCS89]), no caso da computação prematuramente realizada estar correta. Outra vantagem dessas estratégias é a de possiblitar o aproveitamento do *lookahead* inerente a um sistema físico, mesmo quando a programação dos processos lógicos não o faz explicitamente. Para a visualização desta característica, considere uma rede de filas na qual os *jobs* são servidos pela ordem de chegada e possuem todos as mesmas prioridades. Neste sistema, se um *job* chega a uma fila no tempo t e o servidor se ocupa deste por s unidades de tempo, então o relógio local pode ser avançado para $t + s$, pois um *job* que chega num tempo t_2, $t < t_2 < t + s$, não interferirá na partida do anterior, dada a ordem FIFO dos serviços. Considere ainda que o processo lógico que simula a fila foi programado de forma a considerar a chegada e partida de um *job* como dois eventos distintos a serem processados e portanto não se aproveita do *lookahead* intrínseco ao sistema descrito. Tal processo lógico, em conseqüência da chegada de um *job* em t, processaria otimistamente o evento de partida deste em $t + s$ e escalonaria a sua chegada ao processo lógico destino através do envio de uma mensagem para este. A chegada posterior de um *job* no tempo t_2, de acordo com a relação acima, ocasionaria um *rollback*. A estratégia preguiçosa, no entanto, não cancelaria a mensagem enviada pois o reprocessamento exigido pelo *rollback* a geraria novamente.

Modelos mistos, englobando abordagens conservadoras e otimistas, têm sido propostos com a intenção de aglutinar as vantagens de ambas. Dentro desta classe pode ser apontado o recente modelo Espaço-Tempo de Chandy e Sherman ([CS89b]). O modelo considera a simulação como um gráfico bidimensional, onde uma dimensão representa o tempo de simulação e a outra, as

variáveis de estado do sistema. Ao simulador cabe preencher o gráfico através da determinação dos valores das variáveis de estado ao longo do tempo.

Os processos lógicos particionam o gráfico espaço-tempo em regiões contíguas de forma a cada processo ser responsável pelo preenchimento na região a ele atribuída. O modelo segue um algoritmo de relaxação no qual os processos computam o comportamento de suas regiões usando estimativas do comportamento de regiões vizinhas. Toda vez que um processo computar um novo comportamento, ele o informa aos processos responsáveis pelas regiões vizinhas. Sempre que um processo receber novas informações de um vizinho, ele atualiza a estimativa relativa a este vizinho e recalcula o seu comportamento baseado na nova estimativa. O algoritmo termina quando um ponto fixo é alcançado, tal que o comportamento das regiões permaneça inalterado conforme a computação progride. Note que, pelo particionamento do gráfico espaço-tempo, o paralelismo na simulação é também temporal e não apenas espacial como nos modelos até então discutidos.

Modelos parcial ou totalmente otimistas baseiam-se em mecanismos como o *rollback* para o estorno de computações erradas. A avaliação do desempenho de tais protocolos está intimamente ligada portanto, ao tempo gasto em execução incorreta e no estorno desta execução por algum mecanismo. A experiência obtida em implementações do *Timewarp* tem mostrado que o custo de operações de *rollback* pode ser mantido suficientemente baixo.

Capítulo 3

Abordagem por Eventos Condicionais e o Simulador

O presente capítulo procura apresentar a implementação de um simulador geral, independente de aplicações, baseado no paradigma por abordagem de eventos condicionais de Chandy e Sherman ([CS89a]), sendo estruturado em cinco seções.

Na primeira, faz-se uma introdução ao modelo e coloca-se a função do simulador. As duas seções seguintes descrevem os algoritmos síncrono e assíncrono seguidos pelo paradigma. Detalhes da implementação do simulador estão presentes na seção 3.4 e o modo de programação de aplicações neste ambiente está apresentado na seção 3.5.

3.1 Proposta

Em [CS89a] foi descrito por Chandy e Sherman um novo paradigma conservador de simulação distribuída, diferente dos modelos apresentados na seção 2.3 por basear-se no conceito de eventos condicionais. Conforme colocado na seção 2.1, um evento com tempo t_1 na *Lista de Eventos*, numa simulação seqüencial, é condicional na medida em que somente será processado se o processo lógico que o escalonou não receber nenhum evento com rótulo de tempo t_2, tal que $Relógio \leq t_2 < t_1$.

Tomando por base este preceito, o paradigma por abordagem de eventos condicionais separa os eventos escalonados para ocorrer no futuro em condicionais e definitivos. Os últimos

são seguros para processamento enquanto que os primeiros devem aguardar que determinadas condições sejam satisfeitas para tornarem-se definitivos, e portanto elegíveis ao processamento. Vale dizer que o evento com menor rótulo de tempo na *Lista de Eventos* de uma simulação seqüencial é o próximo evento a ser processado, embora seja condicional. A *Lista de Eventos* é, portanto um mecanismo para a determinação de eventos definitivos a partir de eventos condicionais.

No modelo apresentado em [CS89a], os processos lógicos que compõem o sistema implementam o algoritmo para transformação de eventos condicionais em definitivos, além da simulação do sistema físico. A proposta desta tese consiste em criar um simulador, baseado no paradigma por abordagem de eventos condicionais, que se atenha ao algoritmo de transformação, isolando os processos lógicos desta tarefa. Desta forma, o programador do sistema lógico somente necessita de conhecimentos relacionados com o sistema real que se deseja simular. O modelo distribuído necessário para garantir que a simulação progrida livre de erros causais e *deadlock* fica totalmente encapsulado no simulador.

Todas as mensagens trocadas entre processos lógicos fluem através do simulador. Este recebe os eventos, condicionais ou definitivos, dos PLs de origem e os entrega aos PLs de destino no momento oportuno. É tarefa do processo lógico definir, pelo uso de características específicas ao sistema físico, o tipo de um evento escalonado, se condicional ou definitivo. Cabe a este ainda, informar quando um evento condicinal não mais ocorrerá.

Para melhor esclarecer o comportamento de um PL com relação ao simulador, toma-se uma rede de filas segundo a topologia apresentada na figura 2.2. Considerando a existência de *jobs* com dois níveis de prioridade, se um *job* (job1) que chega, digamos no tempo 10, à fila3 é de baixa prioridade, então o processo lógico que simula esta fila (PL_3) deve encaminhar ao simulador um evento condicional para ser escalonado no tempo de simulação 15, no PL que simula a fila4 (PL_4). Se o PL_3 recebe agora um evento no tempo 12, representado a chegada de um job de alta prioridade (job2), então o *job* de baixa prioridade, que seria servido do tempo 10 ao tempo 15, deve ser preemptado para ceder o servidor ao *job* de alta prioridade, de acordo com as regras do sistema físico em questão . Diante disso, cabe ao PL_3 informar ao simulador que o evento condicional de chegada do job1 na fila4 ao tempo 15 não mais ocorrerá. Além disso, o PL_3 deve enviar um evento definitivo de chegada do job2 no tempo 17 à fila4 (como o job2 é de alta prioridade, não poderá ser preemptado) e reenviar o evento condicional de chegada do job1 à fila4, desta feita no tempo 20. É importante notar que um evento condicional tornado definitivo não poderá mais ser eliminado. O simulador deve escolher o momento oportuno para esta transformação de maneira a respeitar este preceito.

Analogamente aos processos lógicos, que se abstêm de detalhes do modelo, ao simulador

são totalmente transparentes as características concernentes ao sistema físico a ser simulado. A separação das tarefas é bastante clara, cabendo ao simulador transformar eventos condicionais em definitivos e entregar estes aos PLs de destino, e aos processos lógicos o processamento e geração de novos eventos segundo as regras ditadas pelo sistema físico que simulam.

A capacidade de determinar quais eventos são definitivos, independentemente do fato de que o evento condicional com menor tempo do sistema é seguro e, conseqüentemente definitivo, consiste no fator crítico para o bom desempenho do paradigma. Passa-se à descrição do modelo por abordagem de eventos condicionais. Este pode ser implementado síncrona ou assincronamente, conforme coloca-se nas duas seções seguintes.

3.2 O Algoritmo Síncrono de Chandy e Sherman

O modelo de Chandy e Sherman está apresentado em duas versões : síncrona e assíncrona. A versão síncrona será discutida primeiramente, por sua simplicidade. Para a sua descrição, considere um sistema lógico [1] conforme definido na seção 2.1 e as seguintes variáveis locais para cada PL_x pertencente ao sistema:

- u_r – Um tempo para cada porta r de entrada, indicando que PL_x recebeu mensagens de outro PL, através da porta r, no intervalo $[0, u_r]$.
- u_s – Um tempo para cada porta s de saída, indicando que PL_x enviou mensagens para outro PL, através da porta s, no intervalo $[0, u_s]$.
- $cond_s$ – O tempo no qual é enviada a próxima mensagem por PL_x através da porta s de saída e depois do tempo u_s, dado que nenhuma mensagem seja recebida através de r e depois de u_r, para toda porta r de entrada de PL_x.
- $next_x$ – o mínimo de $cond_s$, para toda porta s de saída de PL_x.
- E_x – Um conjunto de pares (ps, msg), onde ps é uma porta de saída e msg uma mensagem enviada por esta porta no tempo $next_x$, se nenhuma mensagem for recebida ao longo de r e depois de u_r, para toda porta r de entrada de PL_x.
- $C_x[y]$ – Um vetor para armazenar $next_y$, para todo PL_y do sistema lógico.

[1] Para este modelo, tome um sistema lógico totalmente conectado através de *portas*. Este termo foi usado por compatibilidade com o artigo que descreve o algoritmo.

Neste modelo, uma mensagem representa mais do que apenas um evento para ser escalonado no processo lógico destino. A composição de uma mensagem enviada por um PL_x a um PL_y é mostrada a seguir:

- (s, D_s) – Um par, onde D_s é uma seqüência descrevendo a mensagem na forma de pares $(t[i], e[i])$, com $0 \leq i \leq K$ e $K \geq 0$. $t[i]$ é um tempo, tal que $t[i] > u_s$ e $e[i]$ é a representação de um evento ou o símbolo especial *null*. A seqüência D_s está organizada pela ordem crescente de $t[i]$ e representa tudo a ser enviado ao longo da porta s de saída no intervalo de tempo de simulação $(u_s, t[K]]$.
- $next_x$ – O valor anteriormente definido.

Ao enviar uma mensagem contendo D_s, um PL_x deve atribuir $t[K]$ a u_s e quando a mensagem é recebida por um PL_y, este deve atribuir $t[K]$ a u_r, onde r é a porta de entrada conectada à porta de saída s. Considere, por exemplo, uma simulação na qual um PL_x escalona os eventos definitivos A, B e C para ocorrerem nos tempos 5, 10 e 20, respectivamente, num PL_y. Além disso, $u_s = 3$ e nada mais será enviado no intervalo $(3, 30]$. Os eventos citados estariam caracterizados por uma seqüência $D_s = \{(5, A), (10, B), (20, C), (30, null)\}$. Com o envio da mensagem, a u_s seria atribuído o valor 30 e, quando da chegada da mensagem a PL_y, u_r receberia o valor 30, onde s é a porta de saída de PL_x conectada à posta r de entrada de PL_y.

O algoritmo síncrono é composto de três fases, quais sejam a obtenção de eventos definitivos a partir de eventos condicionais, o envio de mensagens e o recebimento de mensagens. Inicialmente $u_r = 0$ e $u_s = 0$, para toda porta r de entrada e para toda porta s de saída. Além disso, $next_x = 0$, $C_x[y] = 0$ e $E_x = (s, null)$, para todo PL_x do sistema. Cada PL_x executa repetidamente o laço listado na figura 3.1.

Na primeira fase, a equação 3.1 é utilizada para transformar eventos condicionais em definitivos. Aliado ao uso da equação, é possível ainda determinar outros eventos definitivos considerando características específicas do sistema físico, como a certeza, por exemplo, de que um *job* de alta prioridade não será preemptado. Durante a segunda fase, cada PL_x enviará uma mensagem por cada porta s de saída, contendo o valor $next_x$ e um par (s, D_s). A seqüência D_s terá os eventos defintivos a serem escalonados no PL_y conectado à porta s. Com o envio das mensagens, u_s deve ser atualizado. As mensagens enviadas serão recebidas na terceira fase na qual, de posse dos valores nelas contidos, cada PL_x atualizará o vetor $C_x[y]$ e os relógios u_r.

Duas propriedades devem ser provadas para este algoritmo para garantir sua corretude. A primeira é a sua seguraça, ou seja, que os eventos escalonados pelos processos lógicos são eventos

begin

1. **Obtenção de eventos definitivos a partir de eventos condicionais**
   Se a equação abaixo vale:

$$next_x = mínimo\ de\ C_x[y],\ para\ todo\ y \tag{3.1}$$

   então msg será enviada por PL_x ao longo da porta ps no tempo $next_x$, para todos os pares (ps, msg) em E_x. Usando as condições iniciais, as mensagens recebidas e os eventos definitivos obtidos a partir de eventos condicionais, compute a saída de PL_x tão longe no futuro quanto se possa.

2. **Envio de Mensagens**
   Atualize $next_x$ e E_x, envie uma mensagem para cada PL_y e atualize u_s. A mensagem enviada contém o valor (atualizado) de $next_x$ e o par (s, D_s), para cada porta s de saída de PL_x conectada a uma porta de entrada de um PL_y.

3. **Recepção de Mensagens**
   Espere para receber uma mensagem de cada PL. Após o recebimento de uma mensagem de um PL_y, atribua o campo $next_y$ da mensagem a $C_x[y]$ e atualize o valor de u_r, onde r é a porta de entrada conectada à PL_y.

end.

Figura 3.1: Algoritmo Síncrono

que os processos físicos escalonariam. Se T é o menor valor $next_y$, para todo y do sistema, provar-se-á por contradição que para todo t, tal que $t < T$, nenhum evento será enviado, no sistema físico, por uma porta s depois de u_s e antes de T. Suponha, por absurdo, que haja tal envio por alguma porta s de algum PF_y e seja t o menor tempo de todos os eventos enviados. Para que o PF_y mande um evento depois de u_s e antes de $next_y$, é preciso que tenha havido a chegada de um evento por alguma porta r de PF_y num tempo t' anterior a t e posterior a u_r. Isto só poderia ocorrer se houvesse o envio deste evento no tempo t' por uma porta v de algum PF conectada à porta r. Este envio, no entanto, contradiz a afirmação de que t é o menor tempo de um envio anterior a T.

A segunda propriedade a ser provada é a que garante a progressão do algoritmo. Pode ser afirmado que, para uma dada iteração k do laço, existe pelo menos um PL_x para o qual o valor $next_x$ é o mínimo de todo o sistema. Isto determina a existência de pelo menos um

evento definitivo, que poderá ser enviado na próxima iteração. Desta forma, fica provado que o algoritmo progride, livre de *deadlock*. Vale notar que um evento determinado como definitivo sem o uso da equação 3.1 é enviado na própria iteração do seu escalonamento. Um evento condicional transformado em definitivo através da equação é enviado na iteração seguinte ao seu escalonamento, quando o processo lógico já terá recebido os valores $next_y$ de cada PL_y do sistema, informação necessária para considerá-lo definitivo.

Voltemos à figura 2.2 para exemplificar o funcionamento do algoritmo. Partamos de um ponto hipotético da simulação no qual a fila1 possui um *job* de alta prioridade (job1) e as filas 2 e 3 um *job* de baixa prioridade cada uma (job2 e job3, respectivamente). Considere ainda, que os *Relógios* das filas 1 a 3 valem, respectivamente, 5, 3 e 6. Neste ponto:

$next_1 = 9$	$next_2 = 6$	$next_3 = 11$
$E_1 = (s_{13}, job1)$	$E_2 = (s_{24}, job2)$	$E_3 = (s_{34}, job3)$
$D_{13} = \{(9, job1)\}$	$D_{24} = \{\ \}$	$D_{34} = \{\ \}$

Na iteração corrente, somente PL_1 pode enviar um evento (chegada do job1 à fila3 no tempo 9), pois este é o único definitivo. Os outros PLs enviam mensagens contendo $next_x$ e o conjunto D_s vazio. Ao final desta iteração, cada PL_x teria seu vetor $C_x[y]$ com a seguinte configuração: $C_x[1] = 9$, $C_x[2] = 6$ e $C_x[3] = 11$. Na iteração seguinte, pela aplicação da equação 3.1 PL_2 tornaria o evento por ele escalonado em definitivo. Mais ainda, a chegada do job1 à fila3 preemptaria o *job* de baixa prioridade em serviço. PL_3 escalonaria, então um evento definitivo correspondendo à chegada do job1 no tempo 14 à fila4. Seria a seguinte a configuração das variáveis locais:

$next_1 = \infty$	$next_2 = 6$	$next_3 = 14$
$E_1 = (s_{13}, null)$	$E_2 = (s_{24}, job2)$	$E_3 = (s_{34}, job1)$
$D_{13} = \{\ \}$	$D_{24} = \{(6, job2)\}$	$D_{34} = \{(14, job1)\}$

Com isso, PL_2 e PL_3 enviariam definitivamente seus eventos escalonados, com o evento de PL_3 partindo na mesma iteração no qual foi escalonado e o evento de PL_2 na iteração seguinte ao escalonamento.

3.3 O Algoritmo Assíncrono

O algoritmo anterior progride, iterativamente, por passos síncronos. Em cada passo, um PL envia uma mensagem por cada porta de saída e recebe uma mensagem por cada porta de entrada. Há ainda uma versão assíncrona do algoritmo seguido pelo modelo, na qual os PLs progridem na simulação com velocidades diferentes. Quando um processo lógico recebe uma mensagem vinda de outro PL, ele computa sua saída, com base na mensagem recebida, tão longe no futuro quanto possa. Desta forma, um PL não precisa esperar pela chegada de uma mensagem por cada porta para produzir uma saída.

O algoritmo assíncrono tem como objetivo permitir que um PL_x possa utilizar a equação 3.1, a despeito da falta de passos síncronos na progressão da simulação. Conforme será visto a seguir, na apresentação informal do algoritmo, uma condição é definida com o intuito de determinar o momento correto para a aplicação da equação.

Cada PL_y mantém um contador n_r para cada porta r de entrada e um contador n_s para cada porta s de saída. Estes valores informam o número de mensagens recebidas ou enviadas pela porta correspondente. O valor $next_y$ corresponde ao tempo do próximo evento condicional a ser enviado por PL_y. Em *tempos arbitrários*, um PL_y grava $next_y$ e os contadores n_r e n_s — com a restrição de que a gravação seja feita num tempo finito após a mudança dos valores — atomicamente, no sentido em que os valores não podem mudar durante a gravação. Também em tempos arbitrários, um PL_y irradia, para os outros processos lógicos do sistema, os valores gravados — com a restrição de que os valores sejam irradiados num tempo finito após a gravação. Em acréscimo ao vetor $C_x[y]$, que armazena os valores $next_y$ para todo PL_y do sistema, um PL_x possui ainda os vetores $D_x[r]$ e $D_x[s]$, que conterão, respectivamente, os valores n_r e n_s recebidos, para todas as portas r e s do sistema. É da responsabilidade de um PL_x garantir que $C_x[x] = next_x$, $D_x[r] = n_r$, para toda porta r de entrada de PL_x e $D_x[s] = n_s$, para toda porta s de saída de PL_x, de forma que um processo lógico não envie mensagens para si mesmo.

Dado um evento condicional com rótulo de tempo $next_x$, a ser enviado caso PL_x não receba nenhum evento por uma porta r de entrada após u_r, para todo r, este será convertido num evento definitivo caso valha a equação 3.1 e a equação 3.2, conforme definida abaixo:

$$D_x[r] = D_x[s], \textit{ para todas as portas } r \textit{ e } s \textit{ conectadas.} \tag{3.2}$$

A prova de corretude do algoritmo assíncrono está relacionada com a determinação de *Estados Globais* em sistemas distribuídos, conforme discutido em [CL85]. No artigo, Chandy e

Lamport propõem um modelo de sistema distribuído e definem o conceito de *Estado Global* para tal modelo.

De acordo com os autores, um sistema distribuído é composto por um conjunto de processos e um conjunto de canais direcionados que os interconectam. Assume-se que os canais tenham capacidade infinita — ou seja, um processo não se bloqueia na tentativa de enviar uma mensagem — e entreguem as mensagens na ordem de envio, num tempo arbitrário porém finito. O estado de um canal é a seqüência de mensagens através dele enviadas, excluido-se as mensagens através dele recebidas, isto é , o conjunto de mensagens em trânsito.

Um processo é definido como um conjunto de estados, um estado inicial e um conjunto de eventos, onde evento em um processo é uma ação atômica que pode mudar o seu estado e o estado de no máximo um canal incidente ao processo. Um evento e é caracterizado pela tupla $< p, s, s', M, c >$, onde p é o processo onde e ocorre, s é o estado de p antes da ocorrência de e, s' é o estado de p após a ocorrência de e e M uma mensagem (caso exista) enviada ou recebida através do canal c (dependendo de sua orientação).

Definição. *Estado Global.* Um *Estado Global* de um sistema distribuído é um conjunto *consistente* de estados dos processos e canais que compõem o sistema. O *Estado Global Inicial* é composto pelo estado inicial de cada processo e o estado de cada canal como sendo a seqüência vazia.

Seja k o número de mensagens enviadas por um processo p através de um canal c antes que o estado de p fosse gravado. Seja k' o número de mensagens enviadas por um processo p através de um canal c antes que o estado de c fosse gravado. Um conjunto de estados locais de processos e estados de canais é um conjunto *consistente* quando $k = k'$, para todo processo p do sistema e para todo canal c de p orientado para fora. Analogamente, seja l é o número de mensagens recebidas por um processo q através de um canal c antes que o estado de q fosse gravado e l' o número de mensagens recebidas por um processo q através de um canal c antes que o estado de c fosse gravado. Um conjunto de estados locais de processos e estados de canais é *consistente* quando $l = l'$, para todo processo q do sistema e para todo canal c de q orientado para dentro. Obviamente, o número de mensagens recebidas por um canal não pode exceder o número de mensagens enviadas por este canal, ou seja $k' \geq l'$ e $k \geq l$.

Para que se possa compreender a definição de estado global, apresenta-se um exemplo simples, extraído de [CL85]. Considere um sistema formado por dois processos p e q, e dois canais c e c' que os conectam. O canal c parte de p para q e o canal c' de q para p. Um *token* transita entre os processos p e q, que trocam de estado conforme possuam ou não o *token* (s_1 e

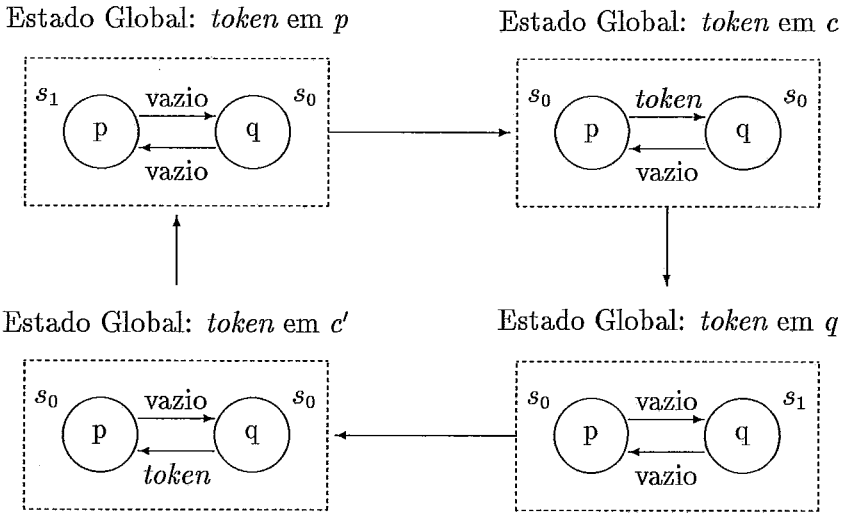

Figura 3.2: Transições de Estado Global

s_0, respectivamente). Dois eventos podem ocorrer no sistema, quais sejam: (1) o envio do *token* (transição de s_1 para s_0) e (2) a recepção do *token* (transição de s_0 para s_1). As transições de estados globais estão mostradas na figura 3.2.

Assuma que, inicialmente, p possua o *token* e grave o seu estado s_p como sendo s_1. Em seguida, p envia o *token* através de c. Suponha agora, que os estados de q, c e c' sejam gravados. Estes seriam, respectivamente, $s_q = s_0$, $s_c = token$ e $s_{c'} = vazio$. O conjunto de estados locais $S = \{s_p, s_q, s_c, s_{c'}\}$ é inconsistente, pois mostra dois *tokens* no sistema, em p e em c. Este fato ocorre porque o estado de p foi gravado antes do envio de uma mensagem através de c e o estado de c, após o envio desta mensagem, e portanto $k < k'$, de acordo com as definições dadas acima para k e k'. É facil ver que se $k > k'$, o conjunto também é inconsistente.

No algoritmo assíncrono por abordagem de eventos condicionais, o estado gravado por um PL_x é composto pelo seu valor $next_x$. Se considerarmos um canal como sendo a união entre uma porta s de saída e uma porta r de entrada, então o seu estado é dado pelos contadores n_r e n_s, mantidos, respectivamente, pelos processos de origem e destino do canal. Desta forma, os processos lógicos interconectados por um canal contribuem para a gravação do seu estado. Note, no entanto, que o algoritmo preocupa-se tão somente com o número de mensagens em trânsito pelo canal (dado pela diferença $n_r - n_s$) e não com as mensagens em si.

O conjunto de estados locais e estados de canais, contido nos vetores $C_x[y]$, $D_x[r]$ e $D_x[s]$ de um PL_x, constitui um estado global, pois a gravação dos valores $next_x$, n_r e n_s é feita atomicamente, conforme já dito, e portanto $k = k'$ e $l = l'$. O algoritmo determina ainda, para a correta aplicação da equação 3.1, que o estado de todo canal do sistema no estado global gravado seja uma seqüência vazia. Esta exigência está representada na equação 3.2, sendo necessária para que um PL não transforme um evento condicional em definitivo sem o conhecimento dos rótulos de tempo dos eventos que estão em trânsito pelos canais. A desobediência desta condição pode resultar em erros causais, como o apontado no exemplo a seguir.

Tomemos um sistema lógico composto por três processos, PL_1, PL_2 e PL_3. Vamos assumir que $next_1 = 3$, $next_2 = 9$, $next_3 = 8$ e que os canais entre os processos estão vazios. Considerando que cada processo tenha gravado o seu estado e o irradiado pelo sistema, junto com os valores dos contadores de mensagens enviadas e recebidas, tem-se que:

$C_x[1] = 3$, $C_x[2] = 9$ *e* $C_x[3] = 8$ *para todo* PL_x *do sistema.*

Desta forma, PL_1 torna o seu evento definitivo e o envia para PL_2, por exemplo. Feito isto, PL_1 grava o seu novo estado e irradia pelo sistema uma mensagem contendo $next_1 = \infty$ e $D_1[1,2] = 1$. Considerando-se que PL_3 recebeu a informação propagada, a configuração de $C_3[y]$ seria:

$C_3[1] = \infty$, $C_3[2] = 9$ *e* $C_3[3] = 8$.

PL_3 poderia então, considerar o seu evento como tendo o menor rótulo de tempo do sistema e torná-lo definitivo. Note, no entanto, que o evento enviado de PL_1 para PL_2 ocorre no tempo 3 e pode cancelar o evento já escalonado por PL_2 para ocorrer no tempo 9. Se, ao ser processado, o evento recebido por PL_2 em 3 gera um evento para ocorrer, digamos, no tempo 6 em PL_3, então um erro causal poderá se configurar, pois o evento condicional que ocorreria no tempo de simulação 8 já foi transformado em definitivo sem, no entanto ter o menor tempo de todo o sistema. O erro foi gerado porque havia uma mensagem em trânsito de PL_1 para PL_2 não considerada quando da aplicação da equação 3.1. Este fato era indicado pela desigualdade $D_1[1,2] = 1 \neq D_2[1,2] = 0$.

Na figura 3.3, as retas horizontais representam processos lógicos e as esferas numeradas, estados locais destes processos. Uma seta inclinada mostra o envio de um evento de um PL para outro. No diagrama, se uma seta une dois estados locais, então o estado origem e o estado destino da seta foram gravados levando-se em conta a partida e a chegada do evento, respectivamente. Os números nas esferas representam os valores $next_x$ dos estados, de acordo com o exemplo anterior. Isto afirmado, resulta que os conjuntos $S_1 = \{3, 9, 8\}$ e $S_2 = \{\infty, 6, 8\}$ são estados globais nos quais é possível a aplicação da equação 3.1 e o conjunto $S_3 = \{\infty, 9, 8\}$ não o é.

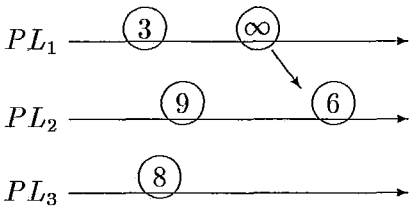

Figura 3.3: Estado Global no qual vale a equação 3.1

A determinação de estados globais — para os quais não há eventos em trânsito — no algoritmo assíncrono tem o mesmo objetivo que o sincronismo por envio e recepção de mensagens no algoritmo síncrono: aplicar a equação 3.1 usando dados consistentes. Desta maneira, se, como mostrado anteriormente, um estado global sem eventos em trânsito é uma *fotografia instantânea consistente* da simulação, então o menor $next_x$ do estado corresponde ao tempo de um evento seguro para processamento. Além disso, como existe pelo menos um PL_x para o qual o valor $next_x$ é mínimo no estado global, então o algoritmo progride, livre de *deadlock*.

3.4 Implementação do Simulador

Conforme visto na seção 3.1, a tese tem como fim a implementação de um simulador que encapsule o modelo por abordagem de eventos condicionais, permitindo aos processos lógicos que se atenham exclusivamente à simulação do sistema físico. A presente seção descreve detalhes de implementação de tal simulador. A estrutura de programação de um processo lógico exigida pelo simulador será abordada na seção seguinte.

O simulador foi implementado na linguagem de programação *Occam 2* para uma máquina paralela distribuída de oito nós com topologia de hipercubo. Cada nó é composto por um processador T800 e um módulo de memória local, não acessível por outros nós. A comunicação entre os processadores é feita exclusivamente por canais físicos que os interconectam de acordo com a topologia citada. No capítulo 4, o ambiente de desenvolvimento — neste incluídos o processador, a linguagem *Occam 2* e a máquina distribuída — será apresentado detalhadamente.

O simulador tem uma estrutura padrão, composta de dois módulos, que é replicada em todos os processadores da máquina distribuída. A figura 3.4 mostra esta estrutura para dois

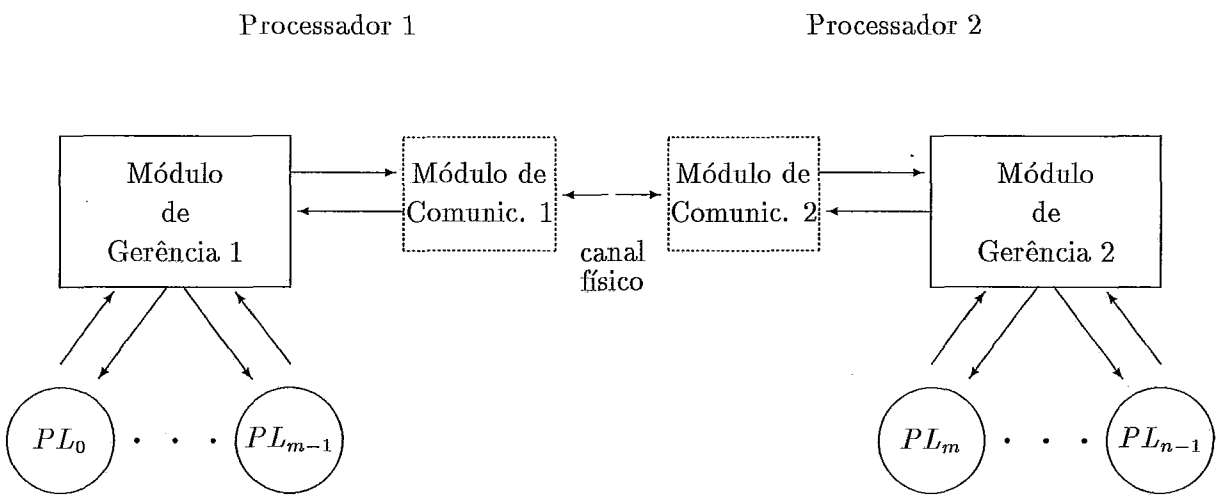

Figura 3.4: Estrutura do Simulador para 2 Processadores

processadores e sua conexão com um sistema lógico de n processos, dos quais m executam num processador e $n - m$ em outro, com $n > m$. O balanceamento de carga é estático e de responsabilidade única do programador do sistema lógico.

O módulo de comunicação faz a ligação entre os módulos de gerência da simulação, que executam em diferentes processadores. Todas as mensagens que fluem de um módulo de gerência (ou gerente, mais sucintamente) para outro módulo externo — sejam estas mensagens eventos para serem processados em processos lógicos externos ou mensagens de controle da simulação que implementam o modelo por abordagem de eventos condicionais — passam pelos módulos de comunicação correspondentes aos gerentes envolvidos. Estes cuidarão para que as mensagens cheguem ao seu destino. Caso não haja um canal físico direto, conectando dois processadores cujos módulos de gerência trocam mensagens, os módulos de comunicação se encarregarão do roteamento destas mensagens por nós intermediários. No escopo definido para esta seção, somente o módulo de gerência será abordado detalhadamente. A descrição do módulo de comunicação fica a cargo da seção 4.4.

A tarefa de um gerente da simulação é receber eventos escalonados pelos processos lógicos sob sua responsabilidade (internos) e encaminhá-los, no momento que considerar oportuno, para os processos lógicos onde estes eventos ocorrerão. Se o PL destino do evento for interno, a entrega do evento é feita pelo próprio gerente. Caso seja externo, o gerente envia o evento, através do

seu módulo de comunicação, para o gerente externo responsável pelo processo lógico destino que se encarregará da entrega. Um evento segue, portanto dois caminhos distintos, caso os *PLs* de origem e destino executem no mesmo processador ou em processadores diferentes, conforme mostrado abaixo:

$PL_{origem} \rightarrow G_1 \rightarrow PL_{destino}$, ou

$PL_{origem} \rightarrow G_1 \rightarrow C_1 \rightarrow C_2 \rightarrow G_2 \rightarrow PL_{destino}$,

onde G_1, G_2 são módulos de gerência e C_1, C_2 são módulos de comunicação.

O algoritmo implementado pelo gerente baseia-se no algoritmo assíncrono do modelo de Chandy e Sherman. Introduz, porém, algumas modificações com a intenção de diminuir o trânsito de mensagens entre processadores e, conseqüentemente, melhorar o desempenho da simulação. As otimizações são possíveis em virtude da concentração das tarefas inerentes ao modelo nas mãos dos gerentes. Os valores $next_x$, por exemplo, para um conjunto de *PLs* que executa em um mesmo processador, são controlados pelo gerente deste processador. Este pode, portanto, saber qual dos processos lógicos possui o menor valor $next_x$ local ao processador, sem que seja necessário a troca de mensagens entre os *PLs*. O menor valor $next_x$ do sistema lógico pode ser descoberto pela troca de mensagens entre os gerentes, cujo número (igual ao número de processadores que, para a máquina utilizada na implementação, totaliza 8) é significativamente menor que o número de processos lógicos para uma aplicação padrão (no exemplo usado na medição do desempenho do simulador, este número chegou a 1.024 *PLs*). São as seguintes as modificações feitas no modelo original:

1. *Semântica de n_r e n_s.* Os contadores n_r e n_s são usados para o controle do fluxo de eventos entre gerentes e não mais entre processos lógicos. A idéia por trás desta modificação é que não é importante saber quais canais não estão vazios, mas sim se **algum** canal não está, já que o modelo exige que não haja mensagens em trânsito em **nenhum** canal no aproveitamento de um estado global gravado. Desta forma, as várias conexões entre os *PLs* que executam em dois processadores diferentes são vistas como dois grandes canais (nos dois sentidos) interligando os dois gerentes que controlam estes *PLs*. Os canais entre os processos lógicos sob a responsabilidade de um mesmo gerente são tratados como um único canal que parte deste gerente com destino ao próprio. Esta medida diminui substancialmente o número de contadores a serem mantidos.

2. *Atualização de n_r e n_s.* Um gerente possui uma estrutura de dados para armazenar eventos, condicionais ou definitivos, já recebidos de *PLs* internos ou externos, para serem entregues aos *PLs* de destino. Em linhas gerais, o algoritmo seguido por um gerente é formado por um laço englobando três operações básicas. Primeiramente, é verificada a

existência de alguma mensagem externa, vinda de outro processador, que, se existir, será devidamente tratada. Em seguida, o gerente tenta transformar eventos condicionais em definitivos, seguindo as regras do modelo original. Na última fase, o evento definitivo com menor rótulo de tempo na estrutura de armazenamento de eventos é selecionado para ser entregue ao PL destino. Vários eventos (definitivos e condicionais) podem estar esperando nesta estrutura que tenham como origem, por exemplo, um PL_x e como destino um determinado PL_y. Dado isto, será a seguinte a forma de atualização de n_r e n_s. Quando o gerente que controla PL_y entregar definitivamente a este um evento escalonado por PL_x, o contador n_r correspondente ao canal que conecta os gerentes responsáveis por PL_x e PL_y é incrementado. A estrutura que armazena os eventos em um gerente está organizada pelos rótulos de tempo dos eventos. Um eventos definitivo nesta estrutura, escalonado por PL_x para ocorrer em PL_y no tempo t, é considerado em trânsito se não há nenhum evento condicional de PL_x para PL_y na estrutura com rótulo de tempo menor que t. O contador n_s, relativo ao canal que une os gerentes responsáveis por PL_x e PL_y, deve ser acrescido de uma unidade para cada evento em trânsito de PL_x para PL_y na estrutura. Deste modo, se um canal parte de um gerente G_1 para outro gerente G_2, então G_1 atualiza o contador n_s do canal para cada novo evento em trânsito escalonado por um PL sob sua tutela para ocorrer em um PL sob a responsabilidade de G_2. Da mesma forma, G_2 atualiza o contador n_r do canal entre G_1 e G_2 para cada evento entregue a um PL sob sua responsabilidade, escalonado por um PL controlado por G_1.

3. *Envio de n_r e n_s.* Os contadores não serão irradiados pelo sistema. Um contador n_r dando conta do número de eventos recebidos de um determinado gerente será enviado somente para este mesmo gerente, de forma que o mesmo possa compará-lo com o contador n_s correspondente e verificar se o canal está vazio. Analogamente, um contador n_s, mantido por um gerente, só será enviado para o gerente receptor dos eventos transmitidos pelo primeiro. Além de restringir o envio dos contadores, o novo algoritmo determina momentos precisos nos quais estes devem ser enviados. Um contador n_r será transmitido para o gerente de direito toda vez que incrementado. Quanto a um contador n_s, este somente será transmitido quando os eventos em trânsito já computados forem efetivamente entregues ao gerente destino. Explica-se. Um contador só é enviado para que seja comparado com o contador correspondente da outra ponta do canal e se possa verificar se este está vazio. De nado adiantaria enviar um n_s se o contador n_r correspondente não leva em conta ainda os eventos em trânsito ainda não entregues, porém já computados em n_s.

4. *Gravação do Estado Local.* Com a nova forma de envio dos contadores n_r e n_s, um gerente tem condições de determinar se os canais a ele incidentes, que o conectam com outros gerentes, estão vazios. Quando isto ocorrer, o gerente grava o seu estado local como

sendo o mínimo dos valores $next_x$ dos processos lógicos sob sua responsabilidade. Feita a gravação, o gerente irradia o estado local para todos os outros gerentes do sistema. Desta forma, cada gerente possuirá uma cópia dos estados locais de todos os gerentes, gravados quando estes perceberam que não havia eventos em trânsito nos canais a eles incidentes. Este conjunto de estados locais constitui um estado global no qual o estado de cada um dos canais do sistema lógico é uma seqüência vazia e conseqüentemente, para o qual vale a aplicação da equação 3.1.

As modificações tentam tirar vantagem do controle da simulação centralizado nos módulos de gerência, de maneira a diminuir as mensagens em trânsito no sistema. No entanto, o princípio básico é o mesmo do algoritmo assíncrono do modelo original por abordagem de eventos condicionais: gravar um estado global do sistema no qual seja possível utilizar a equação 3.1 na transformação de eventos condicionais em definitivos.

O conjunto de estados locais determinado pelo algoritmo modificado constitui um estado global de acordo com a definição encontrada em [CL85], pois embora o estado dos canais e o estado local de um gerente não sejam mais gravados atomicamente, o menor dos valores $next_x$ para um gerente só é computado se for constatado que os canais incidentes a este gerente estão vazios. Esta constatação e a gravação do estado local são feitas atomicamente, no sentido em que os valores $next_x$ e os contadores n_r e n_s não mudam entre as operações. Segue a descrição detalhada do algoritmo do módulo de gerência.

O algoritmo do gerente é subdividido em três seções: seção inicial, de simulação e final. Na primeira, os processos lógicos que compõem o sistema devem se identificar para os seus respectivos gerentes e escalonar os eventos iniciais. Durante a segunda seção, os gerentes controlam o andamento da simulação, recebendo eventos condicionais e definitivos e entregando eventos definitivos para os PLs, até que uma condição de terminação seja alcançada. Na seção final, os gerentes enviam mensagens para o seus respectivos módulos de comunicação, dando conta do término da simulação.

A primeira seção está encarregada da iniciação das variáveis de controle da simulação, como os valores $next_x$ de cada PL ligado a um gerente e os contadores n_r e n_s. Após esta fase, cada PL faz a iniciação das suas variáveis locais e em seguida se identifica para o gerente do seu processador. A cada PL estará associado um número único global, através do qual será feita a sua identificação pelos gerentes. Conhecidas as identificações dos PLs sob sua responsabilidade, um gerente deve enviá-las aos demais irradiando uma mensagem, para que cada um saiba quais gerentes administram quais PLs.

Faz-se aqui uma distinção entre os gerentes. Por restrições da máquina distribuída us-

ada na implementação, somente processos computacionais que executam em um processador específico — o chamado processador raiz — podem realizar operações de leitura e escrita em disco. O gerente que executa neste processador será diferenciado dos demais exatamente por ser o único capaz de realizar tais operações, sendo nomeado gerente raiz.

Após a troca de identificadores, cada gerente, exceto o raiz, envia uma mensagem de sincronização para o gerente raiz, informando que já recebeu uma mensagem com identificadores de cada gerente que compõe o sistema. O gerente raiz, por sua vez, espera pela chegada das mensagens com os identificadores e de uma mensagem de sincronização de cada gerente. Este sincronismo é necessário para separar a fase de identificação da próxima fase, a de espalhamento dos dados iniciais lidos pelo gerente raiz de um arquivo gravado no disco. A estrutura deste arquivo será devidamente descrita na seção 3.5. Pelo momento, é suficiente saber que nele estão gravados dados que devem ser enviados para cada processo lógico do sistema. Cabe ao gerente raiz ler os dados e enviá-los para os *PLs* internos ou para outros gerentes, caso os dados se destinem a *PLs* externos. Deve-se lembrar que todos os gerentes já conhecem a localização de todos os processos lógicos. Com estes dados iniciais, os *PLs* podem determinar, por exemplo, para quais outros *PLs* serão enviados os eventos escalonados ou ainda quais são os eventos iniciais da simulação. Estes são encaminhados aos gerentes tão logo um processo lógico receba seus dados. Desta forma, a seção de simulação começa com um número prévio de eventos escalonados.

Uma outra sincronização é necessária agora para marcar o início da seção de simulação, de forma que os gerentes não confundam mensagens desta seção com mensagens da seção inicial. Assim que um gerente receber as mensagens com os dados iniciais para todos os *PLs* sob sua tutela, ele envia uma mensagem para o gerente raiz informando este fato. Quando o gerente raiz receber uma mensagem de cada gerente do sistema, ele envia uma outra mensagem para cada um dos gerentes vizinhos — gerentes que executam em processadores com os quais há um canal físico direto na máquina distribuída — avisando-os para que dêem início à seção de simulação. Estes, ao receberem esta mensagem, avisam os seus gerentes vizinhos e entram na seção de simulação. O mesmo fará cada gerente que receber a mensagem de início. Desta maneira, nenhuma mensagem da seção de simulação chegará a um gerente antes da primeira mensagem avisando-o do início da seção. Note que cada gerente receberá o número de vizinhos de mensagens de início. As mensagens posteriores à primeira serão ignoradas.

A figura 3.5 mostra o laço repetido na seção de simulação por um gerente, até que este detecte a terminação da simulação. O código está escrito informalmente, porém se utiliza de algumas construções em *Occam 2*[2]. O comando WHILE envolve um construtor ALT, no qual

[2]O leitor não familiarizado com a linguagem deve referir-se à seção 4.2, onde poderá obter detalhes.

```
FimSimulação := TRUE

WHILE NOT FimSimulação
    ALT
        CanalExterno ? Mensagem
            ... Trata chegada de mensagem externa
        TRUE & SKIP
            SEQ
                ... Envia contadores n_r e n_s, se necessário
                ... Grava estado local, se possível, e o irradia
                ... Computa next_x mínimo do estado global
                IF
                    next_x mínimo = ∞
                        FimSimulação := TRUE
                    TRUE
                        SEQ
                            ... Transforma eventos condicionais em definitivos.
                            ... Envia evento definitivo com menor rótulo de tempo
                            IF
                                PL destino do evento é interno
                                    SEQ
                                        ... Recebe eventos resultantes do processamento
                                        ... Remove eventos condicionais cancelados
                                TRUE
                                    SKIP
```

Figura 3.5: Seção de Simulação

é verificada a existência de uma mensagem externa, enviada por outro gerente. Se há uma mensgem, um código é executado para classificá-la e tratá-la de acordo com o seu tipo. Uma mensagem externa pode conter um evento escalonado externamente para ocorrer num PL local, contadores n_r e n_s, o estado local de um gerente, ou ainda a sinalização de que um gerente detectou localmente a terminação da simulação.

Se nenhuma mensagem houver, então o código hierarquicamente envolvido pela declaração TRUE & SKIP é executado. Nesta parte, o gerente realiza as tarefas necessárias para controlar a simulação segundo o modelo por abordagem de eventos condicionais: enviar os contadores n_r e n_s para os gerentes de direito; gravar o seu estado local, quando possível; computar o $next_x$ mínimo do estado global do sistema gravado; transformar eventos condicionais em definitivos; enviar eventos definitivos para os PLs destino; e armazenar os eventos resultantes do processamento de eventos entregues a PLs locais.

O armazenamento de eventos condicionais já escalonados é feito em estruturas denominadas *splay trees*, conforme definido em [ST85]. Uma *splay tree* é reservada para cada processo lógico do sistema em todos os gerentes. Na *splay tree* de um PL estarão armazenados os eventos escalonados para ocorrerem neste PL, quer por PLs internos ao gerente que administra a estrutura, quer por PLs externos. Para facilitar o acesso ao evento definitivo com menor rótulo de tempo nas estruturas (o próximo evento a ser entregue por um gerente), um gerente utliza uma lista simplesmente encadeada, ligando os eventos definitivos com menor rótulo de tempo em cada uma das *splay trees*.

Uma *splay tree* é uma árvore binária que se auto-ajusta à medida que operações de inserção e remoção de nós vão sendo realizadas, de maneira a não permitir a criação de caminhos muito longos e desbalanceados na árvore. Isto é conseguido com a aplicação de uma heurística, denominada *splaying*, a cada operação na árvore. Esta consiste na realização de rotações entre os nós, aplicadas iterativamente do nó inserido ou do pai do nó removido até a raiz da árvore. A aplicação da heurística na árvore binária permite uma complexidade de tempo *amortizada* de $O(log\ n)$ para operações de inserção e remoção, onde complexidade de tempo *amortizada* significa a média da complexidade de tempo numa seqüência de operações de pior caso.

Conforme dito, o laço listado na figura 3.5 é repetido até que uma condição de terminação seja satisfeita. Esta condição pode ser verificada localmente por um gerente. A semântica de um valor $next_x$, como já estabelecido, é o tempo de ocorrência do próximo evento condicional escalonado por um PL_x. Se PL_x não escalona nenhum evento condicional, então $next_x = \infty$. A simulação termina quando o $next_x$ mínimo de um estado global é igual a ∞. Quando um gerente alcança este estágio, ele termina o laço da seção de simulação, envia uma mensagem para cada gerente do sistema informando o acontecido e espera pela chegada de uma mensagem de terminação de cada gerente. Após recebidas estas mensagens, o gerente entra na seção final. A sincronização com os demais gerentes é necessária, pois na seção final o gerente avisa ao seu módulo de comunicação o término da simulação e este suspende sua execução. Se ainda há alguma atividade da simulação em outras partes do sistema e este módulo de comunicação suspenso está servindo de caminho no roteamento de uma mensagem entre dois gerentes não-vizinhos, mensagens podem ser perdidas. Portanto, um gerente só entra na seção final quando souber que todos os gerentes do sistema já detectaram a terminação da simulação.

Algumas regras devem ser observadas na programação de um processo lógico para uso no simulador descrito. Estes detalhes estão listados na próxima seção, que tem o objetivo de ser um guia para a criação de sistemas lógicos.

3.5 Criação de Aplicações

Nenhuma restrição é feita a qualquer sistema real para simulação com o uso do simulador distribuído apresentado, contanto que o sistema possa ser modelado por um sistema físico e implementado por um sistema lógico, de acordo com as definições já dadas. No entanto, regras foram etabelecidas para a elaboração dos processos lógicos, de forma a garantir a corretude da simulação. Esta seção tem o objetivo de apresentar estas regras.

Os processos lógicos, embora assim chamados, não são processos no sentido estabelecido pela linguagem *Occam 2*. Na implementação do simulador, optou-se por criar um módulo da aplicação para cada processador, envolvendo os processos lógicos que executam neste processador. Este módulo é composto por um conjunto de rotinas, escritas pelo programador da aplicação, que implementam o sistema lógico. Este esquema foi escolhido, em detrimento do uso de um processo *occam* para implementar cada processo lógico, para diminuir a tempo perdido com troca de contexto, que pode vir a ser bastante significativo. Os conceitos de sistema lógico e processo lógico, porém, continuam a existir. A comunicação entre o módulo de gerência e o módulo da aplicação, no entanto, será feita através de chamadas a seis funções e não pela troca de mensagens em canais *occam*, como o seria se o módulo da aplicação fosse composto por um conjunto de processos *occam*.

Na visão de um gerente, cada PL sob sua responsabilidade possui dois identificadores: um local e outro global. A identificação global é usada para caracterizar univocamente um processo lógico no sistema lógico inteiro, sendo deste tipo as identificações trocadas pelos gerentes durante a seção inicial (ver seção 3.4). A identificação local é usada para o endereçamento de vetores administrados pelo gerente contendo as variáveis de controle do modelo relativas aos PLs internos. Na seção inicial, cada PL deve se identificar ao seu respectivo gerente, informando a este o par (Id_Local, Id_Global). Esta operação é feita através da chamada de uma função a ser descrita.

Para o simulador, um evento é caracterizado por uma tupla (PL_o, PL_d, t, Tp, Tx), onde PL_o é a identificação do processo lógico origem, PL_d é a identificação do processo lógico destino, t é o rótulo de tempo do evento, Tp é o tipo do evento (condicional ou definitivo) e Tx é o texto do evento, uma seqüência de *bytes* para uso da aplicação cujo significado é transparente para o simulador.

Na criação do módulo da aplicação, o programador deve ter em mente as seguintes regras de comportamento a serem seguidas pelo gerente e pela aplicação. Durante a seção inicial, o gerente chamará a função **IniciaAplic**, que realizará a iniciação das variáveis locais aos proces-

sos lógicos e cuidará da identificação dos mesmos fazendo tantas chamadas da função **Identifica** quantos forem os *PLs* que executam no processador. Cabe ao gerente, ainda na seção inicial, passar os dados iniciais aos processos lógicos através de chamadas à função **DadosIniciais**. Será feita uma chamada para cada processo lógico local. Estes dados contêm eventos iniciais que devem ser escalonados no momento do recebimento dos dados. Durante a seção de simulação, o gerente entrega os eventos aos *PLs* destino por chamadas à função **Aplicacao**. Esta função deve processar o evento recebido e encaminhar os eventos escalonados como resultado do processamento ao gerente através da função **EnviaEvento**. Se o processamento de um evento implicar no cancelamento de eventos condicionais já escalonados, os mesmos devem ser eliminados através da função **ApagaEvento**. É importante que o cancelamento seja feito antes do envio dos novos eventos escalonados pelo processamento do evento recebido.

Algumas constantes devem ser definidas pela aplicação para serem usadas pelo simulador. São estas:

- **VAL INT NLPS** – Uma constante inteira informando o número de processos lógicos locais ao processador.
- **VAL INT TOTLPS** – Uma constante inteira informando o número de processos lógicos total do sistema lógico.
- **VAL INT NGERS** – Uma constante inteira dando conta do número de gerentes, ou seja, o número de processadores envolvidos na simulação.
- **VAL INT GRAU.HIPER** – Uma constante inteira informando o grau do hipercubo, cuja definição é $log_2\ (n)$, onde n é o número de processadores. O grau indica o número de vizinhos de um processador na topologia de hipercubo.
- **VAL INT TAM.TX** – Uma constante inteira indicando o tamanho, em *bytes*, do texto de um evento.
- **VAL INT TAM.FILA** – Uma constante inteira indicando o espaço, em número de eventos, a ser alocado estaticamente para a estrutura de armazenamento de eventos.

Das seis funções que realizam a ligação entre a aplicação e o simulador, três já estão codificadas — as utilizadas pelo módulo da aplicação para a identificação dos processos lógicos e para o escalonamento e cancelamento de eventos — e três são de responsabilidade do programador da aplicação — as chamadas pelo gerente para iniciar as variáveis locais aos processos lógicos, enviar dados iniciais lidos em disco e entregar eventos aos processo lógicos. Segue-se uma lista das funções com descrição dos parâmetros.

1. **Funções utilizadas pelo módulo da aplicação:**

   - PROC **Identifica** (VAL INT IdLoc, IdGlob)
     "IdLoc" é a identificação local do processo lógico e "IdGlob", sua identificação global.
   - PROC **EnviaEvento** (VAL INT LpOLoc, LpD, VAL REAL32 t, VAL INT Tp, []BYTE Tx, INT e)
     "LpOLoc" é a identificação local do *PL* origem do evento e "LpD" a identificação global do *PL* destino. "t" é o rótulo de tempo do evento, "Tp" o seu tipo e "Tx" um ponteiro para uma seqüência de *bytes* contendo o texto do evento. A variável "e" é passada por referência. No retorno da função, "e" contém um ponteiro para o evento guardado na estrutura de armazenamento de eventos. Este ponteiro será usado no caso do cancelamento do evento.
   - PROC **ApagaEvento** (VAL INT e)
     "e" é um ponteiro para o evento a ser cancelado.

2. **Funções utilizadas pelo módulo de gerência:**

   - PROC **IniciaAplic** ()
     Usada para iniciação do módulo de aplicação. Não possui parâmetros.
   - PROC **DadosIniciais** (VAL INT LpLoc, []BYTE dados)
     "LpLoc" é a identificação local do *PL* para o qual se destina a seqüência de *bytes* apontada por "dados". O corpo da função **DadosIniciais** deve fazer tantas chamadas à função **EnviaEvento** quantos forem os eventos iniciais a serem escalonados pelo *PL* identificado por "LpLoc".
   - PROC **Aplicacao** (VAL INT LpO, LpDLoc, VAL REAL32 t, []BYTE Tx)
     "LpO" é a identificação global do *PL* origem do evento e "LpDLoc" a identificação local do *PL* destino. "t" é o rótulo de tempo do evento e "Tx" um ponteiro para uma seqüência de *bytes* contendo o texto do evento.

O arquivo que contém os dados iniciais a serem lidos pelo gerente raiz deve ser um arquivo texto, puramente ASCII (*American Standart Code for Information Interchange*), composto por uma seqüência de números inteiros e reais para cada processo lógico do sistema. Os números destinados a um *PL* devem estar separados por espaços em branco e as seqüências para diferentes *PLs* por um conjunto **CR + LF** (*carriage return + line feed*). Um grupo de números a serem enviados a um *PL* deve ter a seguinte estrutura:

$Ni,\ I_0,\ \ldots,\ I_{Ni-1},\ Nr,\ R_0,\ \ldots,\ R_{Nr-1}$,

onde Ni indica o número de inteiros que se seguem, I_0 a I_{Ni-1} são os inteiros, Nr indica o número de reais que se seguem e R_0 a R_{Nr-1} são os números reais. Estes valores serão agrupados num vetor de *bytes* e entregues aos PLs por chamadas à função **DadosIniciais**, conforme dito.

Capítulo 4

Ambiente de Desenvolvimento

O capítulo 4 se atém à apresentação do ambiente de desenvolvimento do simulador. Neste contexto, a máquina distribuída utilizada, bem como o elemento processador de cada nó seu são descritos. Aliado a isto, são apresentados a filosofia de programação da linguagem *Occam 2* e o seu ambiente de programação TDS. A última seção é reservada para o detalhamento dos módulos de comunicação do simulador, já introduzidos.

4.1 O Multiprocessador NCP I e o Transputer

Concebido para permitir o desenvolvimento de aplicações paralelas de dois modelos distintos — os modelos de programação com memória compartilhada e memória distribuída — o sistema NCP I é uma máquina MIMD ([Stone87]) de arquitetura híbrida. O projeto original prevê a construção de 16 nós idênticos, onde cada nó é constituído por dois elementos: *Private Memory Processing Element* (PLE) e *Shared Memory Processing Element* (SME). Cada nó possui ainda dois módulos de memória. Um módulo privado, cujo acesso é restrito ao PME e um módulo compartilhado, acessado por ambos elementos processadores (PME e SME). Mensagens pequenas podem ser trocadas entre o PME e o SME por um *buffer* de mensagens. Os PMEs comunicam-se por uma rede de interconexão formando uma topologia de hipercubo. Os SMEs ligam-se por um barramento através do qual podem acessar a memória centralizada da máquina e o controlador de memória secundária. A descrição completa da arquitetura do NCP I pode ser encontrada em [ACSC91].

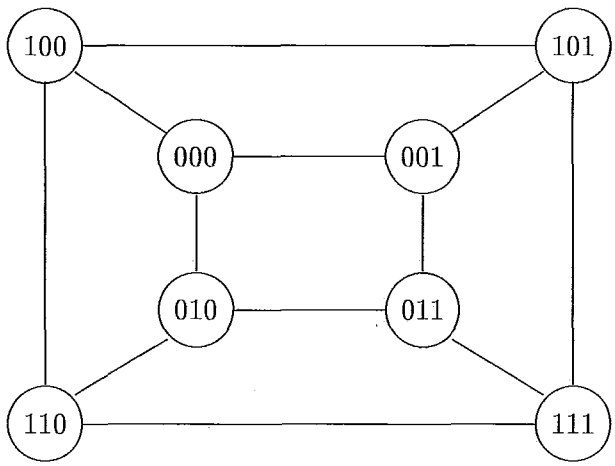

Figura 4.1: Hipercubo de 8 nós

O projeto do NCP I está ainda em fase de desenvolvimento. Atualmente, um protótipo de 8 nós encontra-se em funcionamento. O protótipo acabado não dispõe de uma memória compartilhada, constituindo-se numa arquitetura puramente distribuída. É, no entanto, uma plataforma adequada para a implementação do simulador distribuído. Cada nó possui apenas o PME e seu módulo de memória local e os nós estão conectados segundo a topologia de hipercubo. A comunicação da rede de nós com o computador hospedeiro (um microcomputador PC-AT, compatível com a linha IBM) é feita através de um único PME, doravante designado processador raiz. Este é o único capaz de realizar operações de E/S. A figura 4.1 ilustra a interconexão entre os nós do protótipo implementado, formando um hipercubo de 8 processadores. Um hipercubo de grau n possui 2^n processadores. Cada processador tem n vizinhos que são identificados tomando-se a representação binária do número que rotula o processador e negando cada um dos n *bits* que compõem o número binário.

Um PME é constituído por um *transputer* T800, que é um microprocessador CMOS de 32 *bits*. Em uma única pastilha, o T800 incorpora uma memória SRAM de 4 Kbytes, uma unidade de ponto flutuante de 64 *bits* e 4 *links* bidirecionais de comunicação. O microprocessador opera a 20 MHz, suportando uma capacidade de 1,5 Mflops. Cada *link* serial transfere dados à taxa de 20 Mbits/s. Cada módulo de memória local a um PME possui 2 Mbytes de capaciadade de armazenamento, exceto o PME raiz, cujo módulo local possui 4 Mbytes.

A memória interna à pastilha permite que o *transputer* tenha um número pequeno de registradores. Seis registradores são usados na execução de um processo seqüencial, quais sejam: um ponteiro para o *workspace* do processo, onde estão armazenadas suas variáveis locais; um ponteiro de instruções, indicando a próxima instrução a ser executada; um registrador para a formação de operandos das instruções; e três registradores — A, B e C — que constituem uma pilha de avaliação e são as fontes e destinos da maioria das operações aritméticas e lógicas.

O modelo de concorrência e comunicação da linguagem *Occam 2* (ver seção 4.2) é eficientemente suportado pelo *transputer*. O escalonador de processos está implementado em microcódigo na pastilha, o que elimina a necessidade de um núcleo de sistema operacional em *software*. De acordo com as regras impostas pelo escalonador, um processo pode estar ativo — em execução ou pronto para executar — ou inativo — pronto, à espera de uma mensagem; pronto para enviar uma mensagem ou esperando por um intervalo de tempo. Os processos ativos esperando para serem executados estão mantidos em uma lista e de lá são selecionados para execução. Processos inativos esperando para enviar ou receber uma mensagem são colocados em filas associadas aos canais pelos quais se dará a comunicação.

No projeto de um dispositivo VLSI é essencial a existência de uma descrição comportamental das funções do dispositivo, bem como dos componentes que o formam e o modo como estão interconectados. Para tal, linguagens de descrição de *hardware* têm sido usadas que ofereçam maneiras de expressar paralelismo e comunicação, primordiais na caracterização de um dispositivo. O desenvolvimento do *transputer* T800 está intimamente ligado com a linguagem *Occam*. Esta foi usada como linguagem para descrição do comportamento do dispositivo VLSI que o compõe.

O *transputer* T800, por suas especificações, foi construído para ser usado como *building block* para sistemas concorrentes de processamento. Neste sentido, o NCP I no seu atual estágio de desenvolvimento é uma coleção de *transputers*, conectados por seus *links* seriais de comunicação. Esta configuração o torna ideal para implementação de aplicações paralelas distribuídas especificadas em *Occam*, tal como o é o simulador descrito no capítulo 3.

4.2 A Linguagem Occam 2

O surgimento de máquinas multiprocessadas — decorrente da tentativa de contornar o limite teórico imposto para a velocidade de microprocessadores e do barateamento de pastilhas VLSI — trouxe consigo a necessidade de criação de novas linguagens, próprias para o desenvolvimento de aplicações paralelas. Tais linguagens deveriam ser capazes de expressar paralelismo de forma natural e prover primitivas de comunicação entre processos concorrentes, características estas

não encontradas em linguagens seqüenciais tradicionais como *Fortran* ou *Pascal*. O *Occam* [Burns88] aparece, neste contexto, com o objetivo de propocionar um ambiente de programação paralela.

Conforme aludido na seção 4.1, a linguagem *Occam* e o *transputer* estão associados desde o desenvolvimento de ambos. Mais precisamente, o *transputer* foi projetado com a intenção de constituir uma plataforma de *hardware* eficiente para a execução de processos *Occam*. O modelo de concorrência de *Occam* foi fortemente influenciado pela necessidade de prover as mesmas técnicas de programação para um único *transputer* ou para uma rede destes. O projetista do sistema paralelo pode abster-se completamente do mapeamento dos processos na rede de processadores durante a fase de desenvolvimento.

Embora haja implementações de outras linguagens para o T800 (*C* e *Fortran*), escolheu-se *Occam* para o desenvolvimento do simulador distribuído com base nas colocações feitas acima: a forte relação entre a linguagem e o processador, a naturalidade na expressão de paralelismo e a independência do simulador com as características físicas da rede de *transputers* (topologia e número de processadores).

Processos que interagem para a execução de uma tarefa devem poder se comunicar. Esta comunicação pode ser feita através de variáveis compartilhadas ou por troca de mensagens entre os processos. Se a segunda opção é escolhida, então a linguagem paralela usada na implementação dos processos concorrentes deve prover duas primitivas: SEND e WAIT — a primeira para o envio de uma mensagem e a segunda para a espera da chegada desta. O modelo de comportamento das primitivas classifica-se em síncrono ou assíncrono. No primeiro, o processo que executa um SEND fica bloqueado até que o processo destino da mensagem execute o WAIT correspondente. No modelo assíncrono, um processo não se bloqueia pela execução de um SEND e prossegue sem saber se a mensagem foi recebida. A sintaxe das primitivas de comunicação, ou seja a forma de designar os processos origem e destino das mensagens também depende da linguagem paralela.

Quando há um nome único no sistema para identificar cada processo, o envio e recepção de mensagens pode ser indicado utilizando-se explicitamente os nomes dos processos origem e destino, conforme abaixo:

SEND mensagem PARA processo
WAIT mensagem DE processo

Se não há interesse em saber-se a origem de uma mensagem, mas sim se existe tal mensagem destinada a um processo, a primitiva WAIT poderia ser expressa na forma mais simples:

WAIT mensagem

Uma outra abordagem seria a de definir entidades intermediárias (*mailbox* ou canais, por exemplo) para onde as mensagens fossem mandadas e de onde estas mensagens fossem lidas:

SEND mensagem PARA *mailbox*
WAIT mensagem DE *mailbox*

Com relação a esta estratégia, variações podem ainda existir definindo quantos processos podem escrever e quantos podem ler mensagens de uma determinada entidade intermediária. *Occam* permite a comunicação entre processos concorrentes através da troca de mensagens, segundo o modelo CSP (*Communicating Sequential Processes* [Hoare78]). O modelo de comunicação é síncrono, com a utilização de canais como entidades intermediárias. Um canal conecta dois únicos processos, unidirecionalmente, e possui um tipo (inteiro, real, *booleano*, etc) que está associado ao tipo das mensagens que por ele fluirão. Os comandos de leitura e escrita em canais têm sintaxe simples, definida por dois símbolos: ! e ?. Para enviar uma variável X por um canal ch, um processo executa:

ch ! X

Analogamente, para ler-se uma mensagem de uma canal ch para uma variável Y, deve-se executar:

ch ? Y

Tipicamente, um programa *Occam* é uma coleção de processos concorrentes que trocam mensagens. Cinco são os processos primitivos suportados pela linguagem: SKIP, STOP, atribuição, envio de mensagem e recepção de mensagem. O processo SKIP não tem efeito algum e termina imediatamente. É usado em certas construções para indicar que nenhuma ação deve ser tomada. Em oposição ao SKIP, o processo STOP nunca termina, sendo usado quando do reconhecimento de uma condição de erro. Uma atribuição tem a sintaxe expressa abaixo e a semântica similar a de outras linguagens procedurais.

$V := e$, onde V é uma variável e e uma expressão qualquer válida em *Occam*.

Os processos para envio e recepção de mensagens possuem a sintaxe já descrita para comuicação:

ch ! e
ch ? V

O efeito dos dois processos acima é o mesmo de uma atribuição $V := e$, onde V está em um processo, e e em outro. Uma comunicação é vista em *Occam*, portanto como uma simples atribuição.

Um conjunto de processos primitivos deve estar agrupado por *construtores*. São estes: WHILE, IF, SEQ, PAR e ALT. Os dois primeiros, a despeito de alguns detalhes semânticos e sintáticos, comportam-se como seus semelhantes em linguagens procedurais. De importância são os três seguintes, por implementarem características específicas de *Occam*.

Um grupo de processos envolvidos por um construtor SEQ é executado seqüencialmente.

```
SEQ
  a := 4
  b := a + 10
  ch ! b
```

O trecho acima envia o número inteiro 14 através do canal *ch*, uma vez que as duas atribuições são realizadas uma após a outra e antes do comando de envio.

O construtor PAR indica que um grupo de processos deve ser executado concorrentemente. Os subprocessos envolvidos por um PAR devem prosseguir independentemente dos demais ou interagir com outros subprocessos por troca de mensagens. Um processo PAR só termina quando todos os subprocessos que o constituem terminam. O trecho abaixo implementa um *pipeline* de dois processos, cuja entrada é feita pelo canal *in* e a saída pelo canal *out*. Os processos comunicam-se pelo canal *middle*.

```
CHAN OF INT in, out, middle:
PAR
  INT X:
  WHILE TRUE
    SEQ
      in ? X
      middle ! X
  INT X:
  WHILE TRUE
    SEQ
      middle ? X
      out ! X
```

Uma variação de PAR — PLACED PAR — é usada para mapear os procesos concorrentes em diferentes processadores. Note, no entanto que o efeito de

```
PLACED PAR
  PROCESSOR 1          PAR
    P1          e        P1
  PROCESSOR 2            P2
    P2
```

é o mesmo, com relação a lógica de execução do programa.

O construtor ALT possibilita a um processo escolher, não deterministicamente, entre um número de possíveis fontes de mensagens. É a seguinte a sua forma geral:

```
ALT
  G1
    P1
  G2
    P2
  ⋮
  Gn
    Pn
```

onde Gi é um guarda e Pi um processo, executado quando o guarda correspondente for considerado *pronto*. Um guarda pode ser uma primitiva de recepção de mensagem, uma expressão *booleana* e o processo primitivo SKIP ou uma primitiva de recepção e uma expressão *booleana*. Um guarda será considerado pronto quando houver uma mensagem disponível para ser lida no canal associado ao guarda, ou a expressão *booleana* for verdadeira ou ainda ambas as condições forem satisfeitas, dependendo de como a estrutura do guarda se encaixar, respectivamente, em um dos tipos supra-citados. Se mais de um guarda for considerado pronto, então um deles é escolhido arbitrariamente. Se nenhum guarda estiver pronto, então o construtor termina imediatamente.

Pelas características descritas da linguagem *Occam*, pode-se notar que ela oferece um ambiente natural para a programação de sistemas lógicos de simulação distribuída, conforme a definição do capítulo 2. O esquema de comunicação por canais unidirecionais e a possibilidade de criação de processos concorrentes, igualmente especificados independente de alocação em um mesmo ou em diferentes processadores, encaixam-se perfeitamente ao modelo de programação de simulações descrito.

4.3 Ambiente de Programação

O suporte para a programação em *Occam* numa rede de *transputers* é dado por um ambiente de programação integrado, o *Transputer Development System* (TDS), descrito em [Inmos88]. Este compreende um editor de fontes, um gerenciador de arquivos, um compilador, um *link*-editor, uma ferramenta de depuração e um conjunto de bibliotecas que provê, basicamente, funções matemáticas e operações de E/S.

O TDS executa no *transputer* raiz da rede (o *transputer* conectado com o computador hospedeiro da rede, tipicamente um microcomputador compatível com a linha IBM-PC), permitindo a utilização de teclado, tela e disco do computador hospedeiro. Neste estará ativo um programa servidor de arquivos e operador de terminal para a realização dos serviços requisitados pelo TDS.

Cabe ao TDS carregar a rede de *transputers* com os processos a serem executados em cada processador. Para tal, um arquivo de configuração indicando a localização dos processos na rede deve ser provido ao sistema. A partir do *transputer* raiz, o TDS se encarregará de espalhar os processos devidamente na rede.

A principal ferramenta do TDS é o seu editor de fontes. Dentro do ambiente de edição, pode-se invocar, por exemplo, o compilador ou carregar-se um programa na rede para posterior execução. O editor é baseado no conceito de *folding*, que dá uma estrutura hierárquica ao fonte, refletindo a estrutura do programa em desenvolvimento. Através do uso deste conceito, é possível esconder porções do texto hirarquicamente inferiores. Um *fold* contém um bloco de linhas que pode ser mostrado de duas maneiras. Da primeira forma, aberto, as linhas do bloco são apresentadas entre duas marcas. Da segunda forma, fechado, o bloco é representado por uma única linha, a linha de *fold*. A figura 3.5 no capítulo 3 contém um trecho de código com *folds* apresentados na forma fechada.

O depurador não permite um acompanhamento passo a passo da execução de programas. Em oposição, o depurador aponta a linha do programa fonte onde foi interrompida a execução, quer por uma condição de erro, quer por uma interrupção explícita da parte do programador, e permite a investigação do conteúdo de variáveis do processo interrompido. É possível ainda mudar de processador na rede para investigar-se outros processos, também no ponto de interrupção. A ação do depurador é, portanto, muito limitada, pois dá ao programador apenas uma fotografia instantânea da execução. Este fato tornou a depuração do simulador bastante lenta.

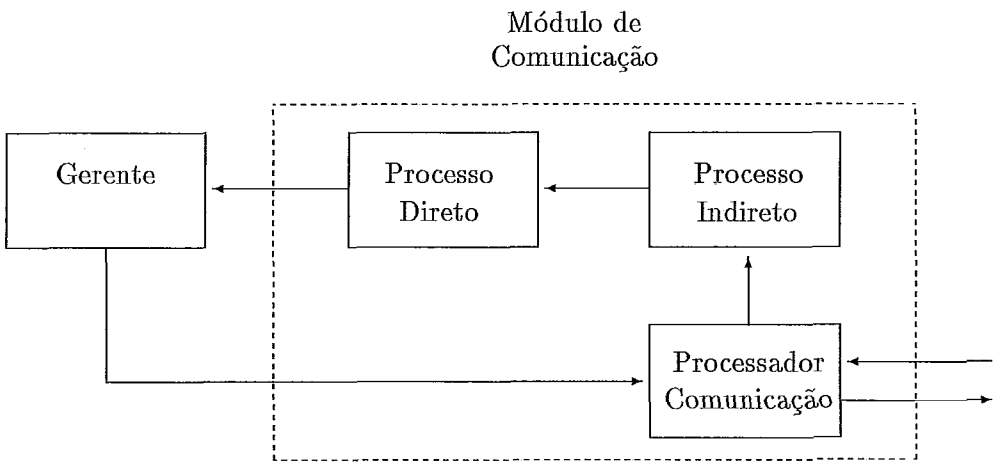

Figura 4.2: Módulo de Comunicação

4.4 O Módulo de Comunicação

Conforme dito na seção 3.4, o simulador está estruturado em três módulos: gerência, comunicação e aplicação. Cada processador da máquina distribuída replica esta estrutura, indistintamente. Os módulos de gerência e aplicação foram descritos no capítulo 3. Esta seção tem como objetivo descrever o módulo de comunicação.

Os processos lógicos enviam eventos para outros processos lógicos através dos gerentes. Estes trocam entre si informações de controle da simulação, pertinentes ao modelo por abordagem de eventos condicionais. Uma camada de *software* que controle este fluxo de mensagens se faz necessária por dois motivos: para evitar *deadlock* e para rotear as mensagens entre os processadores não-vizinhos. Tendo em vista estas duas questões, o módulo de comunicação foi projetado para conter três processos, no sentido conhecido pela linguagem *Occam 2.* A figura 4.2 ilustra o módulo.

O processo central da estrutura é o *processador* de comunicação (CVP), desenvolvido por Lúcia Drummond como tese de mestrado da COPPE em 1990 ([Drumo90]). O CVP funciona como um *processador* para o qual se passam instruções. São várias as instruções providas, porém somente duas são usadas pelo simulador. A primeira — instrução SEND — realiza a troca de mensagens entre dois nós, sejam estes vizinhos ou não. Desta forma, o roteamento de mensagens na rede de *transputers* fica embutido no processador de comunicação, sendo totalmente

transparente para o simulador. Dois operandos são necessários para a realização de SEND: a identificação do processador destino e um vetor de *bytes* formando a mensagem a ser enviada.

A segunda instrução utilizada — BROADT — irradia uma mensagem através de uma árvore geradora da rede de *transputers*, árvore esta previamente definida através de um vetor de *booleanos* que indica quais canais da rede a constituem. No simulador, é definida uma árvore para cada processador para que todos sejam capazes de irradiar mensagens no sistema. São operandos de BROADT a identificação da árvore geradora a ser usada e o vetor de *bytes* que constitui a mensagem.

Com o intuito de evitar *deadlock* no sistema, foram criados dois processos intermediários — processos direto e indireto — para coordenar o trânsito de mensagens recebidas externamente por um CVP a serem encaminhadas ao seu gerente. Estes processos simulam um canal de capacidade infinita entre CVP e gerente. Além disto, os processos intermediários permitem que um CVP não fique bloqueado esperando que o gerente correspondente esteja pronto para receber uma mensagem. Pela descrição feita do processador de comunicação, pode-se determinar a importância deste no esquema de troca de mensagens entre os gerentes. Como conseqüência, é fácil imaginar o impacto no desempenho da simulação causado pelo seu bloqueamento.

O processo indireto está conectado com o CVP e é capaz de armazenar um número grande de mensagens. A idéia é ter um processo simples, que esteja sempre pronto para receber mensagens do CVP, armazenando-as e repassando-as, na ordem de chegada, para o processo direto. Este receberia uma mensagem por vez e esperaria que o gerente estivesse pronto para a comunicação.

O protocolo de comunicação seguido pelos dois processos é simples. Considere um vetor de n posições, onde cada posição é suficiente para conter a maior mensagem que flui entre gerentes. No módulo de comunicação, este vetor é administrado pelos processos direto e indireto. Ao primeiro cabe uma posição do vetor e ao segundo, as $n-1$ posições restantes. Sempre que a posição do vetor administrada pelo processo direto estiver vazia, este faz o pedido de uma mensagem ao processo indireto. Quando o gerente estiver pronto para receber uma mensagem externa, o processo direto entrega a mensagem recebida do processo indireto e faz novo pedido. Este esquema transfere todo o ônus do modelo síncrono de comunicação para o processo direto, uma vez que este é o único a se bloquear, e libera o processador de comunicação para controlar o fluxo de mensagens na rede de *transputers*.

Os processos intermediários são necessários somente em um sentido da comunicação entre os módulos de comunicação e os módulos de gerência. No sentido inverso, os gerentes se conectam diretamente com os CVPs correspondentes.

Capítulo 5

Resultados e Conclusões

O objetivo deste capítulo é apresentar o resultados obtidos com a medição do desempenho do simulador. Os teste foram realizados segundo o modelo apresentado na primeira seção. A última seção do capítulo coloca as conclusões aos quais se chegou pela análise global do trabalho.

5.1 O Modelo de Teste

O modelo de simulação distribuído por abordagem de eventos condicionais — com as modificações descritas e implementado na máquina NCP I — teve o seu desempenho medido através de um sistema lógico que simula o comportamento de uma rede de filas computacionais. Esta rede possui uma topologia parametrizada por quatro fatores, os quais foram variados para que o desempenho do simulador pudesse ser avaliado em diversas situações.

A figura 5.1 ilustra o aspecto da rede utilizada nos testes, ressaltando os parâmetros que a caracterizam. A rede é fechada, com uma população de *jobs* fixa. Esta população — o primeiro parâmetro a ser variado — é dividida pelo número de filas da rede para que se saiba o número de *jobs* iniciais de cada fila. A rede é caracterizada também pelo número de *switches* (S) que a compõem. Um *job*, ao chegar a um *switch*, deve escolher uma de suas arestas de saída, de acordo com probabilidades associadas a estas arestas, para prosseguir (Nos testes, foram consideradas probabilidades iguais para todas as arestas de partida de um *switch*). Feita a escolha, o *job* entra numa seqüência de filas que o levará a um outro *switch*, no qual uma nova aresta será sorteada. O número de arestas (A) na saída de um *switch* e o número de filas (F) que forma a seqüência

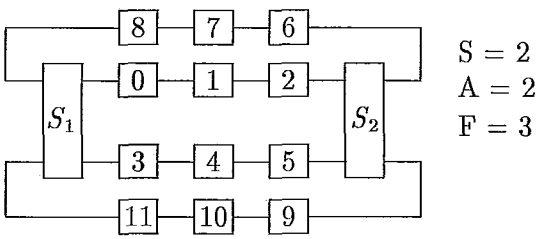

Figura 5.1: Modelo de Teste

de uma aresta são os outros parâmetros de caracterização.

Pode-se abstrair dos *switches* da rede de filas e considerar-se que as filas se conectam diretamente. Desta forma, a última fila de cada seqüência se conecta com cada uma das filas no início das seqüências posteriores a um *switch*. Dito isto, pode-se tomar a rede como um conjunto de filas interconectadas por canais, de forma que a primeira fila de uma seqüência tenha um número A de canais de entrada e um canal de saída, uma fila do meio da seqüência tenha um canal de entrada e um de saída e a última fila da seqüência, um canal de entrada e A canais de saída.

A nova aparência da rede de filas está ilustrada pela figura 5.2. Nela, os PLs foram numerados na ordem das seqüências e dos *switches*. No balanceamento de carga da simulação, a cada processador foram alocados o número de filas da rede dividido pelo número de processadores envolvidos na simulação. Os PLs que executam em um mesmo processador têm numeração contígua, respeitando-se a ordem imposta na ilustração.

Um processo lógico foi alocado para simular cada fila da rede e se comportará como se segue. Ao chegar um *job*, o PL toma o máximo entre seu relógio de simulação e o tempo de chegada do *job* e soma a este valor um tempo de serviço calculado (o modo de cálculo do tempo de serviço será explicado posteriormente). Feito isso, o PL sorteia um canal de saída — que será único caso o PL simule uma fila do início ou do meio de uma seqüência — e envia um evento, com o tempo de ocorrência calculado da forma descrita, para o PL com ele conectado através do canal escolhido[1]. Pelo comportamento descrito, pode-se verificar que um PL envia eventos na ordem crescente de rótulos de tempo. Sendo assim, por cada canal de entrada de um PL chegarão eventos também na ordem crescente de tempo.

Como os *jobs* possuem a mesma prioridade, um PL que tenha somente um canal de entrada

[1]Note que o envio de eventos é sempre feito através dos gerentes.

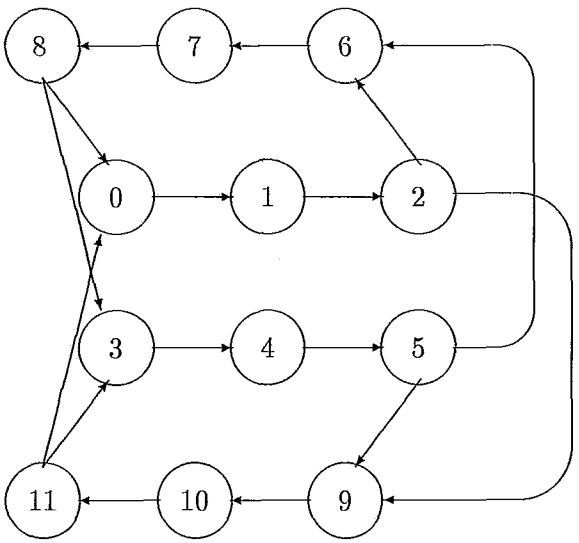

Figura 5.2: Conexão dos PLs no Modelo de Teste

pode escalonar todos os seus eventos como definitivos, já que nenhum *job* será preemptado, pois foi garantida a ordem crescente de tempos dos *jobs* que chegam. No entanto, um PL que possua A canais de entrada deve saber determinar quando um evento escalonado é definitivo ou condicional. Para isso, um relógio é mantido para cada canal de entrada de um PL, indicando o tempo de chegada do último evento por este canal. Se um evento e chega por um canal tal que seu tempo de ocorrência é menor que o relógio de todos os canais de entrada, então nenhum evento chegará cujo rótulo de tempo seja menor. Desta forma, o PL pode processar o evento e marcar o evento resultante do processamento como definitivo.

Eventos que não tenham rótulo de tempo menor que os relógios dos canais de entrada geram eventos condicionais quando processados. Estes eventos condicionais deverão ser cancelados caso seja constatado um erro na ordem de processamento dos eventos. Com o intuito de minorar este cancelamento, os PLs com mais de um canal de entrada foram programados de maneira a acumular alguns eventos e depois processá-los a todos de uma vez. Com isso, eventos que chegam na ordem incorreta na fase de acúmulo poderão ser processados na ordem correta dos rótulo de tempo. Um PL acumula eventos escalonando eventos especiais para si mesmo. Se a lista de eventos retidos está vazia, o próximo evento recebido será posto na lista e causará o escalonamento de um evento especial para um tempo no futuro, enviado ao gerente. Eventos posteriores vão sendo introduzidos na lista. A lista será processada na ordem dos rótulos de

tempo assim que o evento especial for recebido pelo PL.

O tempo de serviço de um *job* foi modelado como uma variável aleatória contínua, exponencialmente distribuída. Segundo Trivedi em [Trive82], o tempo entre duas chegadas sucessivas de *jobs*, o tempo de serviço num servidor de uma rede de filas e o tempo de falha de um componente de um sistema são exemplos de variáveis aleatórias exponenciais. As razões para o uso da distribuição exponencial incluem a sua relação com a distribuição de Poisson e o fato de possuir a propriedade de Markov. Esta reza que o comportamento da variável aleatória independe estatisticamente do seu comportamento passado. A distribuição exponencial é expressa por:

$F(x) = 1 - e^{-\lambda x}$, onde λ é uma taxa constante.

Seja uma variável aleatória R uniformemente distribuída e seja uma variável aleatória X, com distribuição dada pela inversa de $F(x)$ num ponto R.

$$X = F^{-1}(R)$$

$$X = -\frac{ln\ (1-R)}{\lambda}\ .$$

A função de distribuição de probabilidade de X é dada por:

$$\begin{aligned} P(X \leq x) &= P(-\frac{ln\ (1-R)}{\lambda} \leq x) \\ &= P(R \leq 1 - e^{-\lambda x}) \\ &= 1 - e^{-\lambda x}\ . \end{aligned}$$

Portanto, X é exponencialmente distribuída. Além disto, se R é uniformemente distribuída, então $1 - R$ também o é. Desta forma, o tempo de serviço t de um *job* pode ser obtido pela expressão:

$$t = -\frac{ln\ R}{\lambda}\ .$$

5.2 Resultados Obtidos

Para a avaliação do desempenho do simulador, foram medidos os tempos de execução das simulações das redes de filas — obtidas com a variação dos parâmetros que as definem — utilizando-se 2, 4 e 8 processadores do NCP I. No escopo definido para estas variações, limitou-se o número de filas inferiormente em 128 e superiormente em 1.024. Com relação à população (P) de *jobs* no sistema, optou-se por simular cada rede de filas com 16K, 32K e 64K *jobs*[2].

[2] 1K *job* corresponde a 1.024 *jobs*.

Os parâmetros A, S e F foram variados como se segue. Inicialmente, fixou-se $S = 2$ e $F = 32$ e executou-se a simulação para $A = 2$, $A = 4$ e $A = 8$. Em seguida, S e F foram fixados em 2 e 16, respectivamente, e A tomou os valores 4 e 8. No que concerne o parâmetro S, os parâmetros A e F foram mantidos com os valores 2 e 32, respectivamente, e a simulação foi executada para $S = 2$, $S = 4$ e $S = 8$. Novamente fixou-se A e F, porém com $A = 2$ e $F = 16$, e atribuiu-se a S os valores 4 e 8. Finalmente, S e A foram mantidos com os valores 4 e 2, respectivamente, para que se pudesse avaliar a variação do parâmetro F. A simulação foi executada com $F = 16$, $F = 32$, $F = 64$ e $F = 128$. Cada uma das combinações de valores para A, S e F define uma rede de filas diferente. As combinações listadas acima foram simuladas com cada uma das três populações de *jobs* escolhidas. Uma configuração A, S, F e P foi simulada quatro vezes para que se pudesse extrair uma média dos tempos de execução, dado o não-determinismo da simulação.

Os tempos obtidos com as simulações paralelas foram posteriormente comparados com os tempos medidos para as mesmas combinações de parâmetros num simulador seqüencial. Tal simulador implementa o modelo de simulação seqüencial descrito na seção 2.1. A estrutura de dados utilizada para a lista de eventos é a mesma do simulador distribuído, a *splay tree*. Porém, como neste um gerente subdivide a estrutura de *splay tree* e usa uma lista simplesmente encadeada ligando as cabeças das subestruturas (ver seção 3.4), estudou-se a melhor forma de implementação da lista de eventos para o simulador seqüencial: se como uma *splay tree* única ou como um conjunto de *splay trees*, encadeadas por uma lista. Considerando uma única *splay tree*, o tamanho máximo n da estrutura seria a população de *jobs* do sistema. Uma operação em tal estrutura teira custo de tempo $O(log\ n)$. Com várias *splay trees*, a lista de cabeças teria tamanho m igual ao número de filas na rede e cada *splay tree* teria tamanho máximo p, onde $n = mp$. As operações de inserção e remoção teriam custo de tempo $O(m) + O(log\ p)$, caso a operação causasse uma atualização na lista simplesmente encadeada, ou $O(log\ p)$, caso essa atualização não fosse necessária. Comparando o custo de o operações, das quais l demandam uma atualização na lista de cabeças, tem-se que a utilização de várias *splay trees* é vantajosa, ou seja,

$$o\ O(log\ n) > l\ (O(m) + O(log\ p)) + (o - l)O(log\ p)\ ,$$

quando l é pequeno em relação a o e n é grande em relação a m. Experimentalmente, tal situação verificou-se para um rede de 128 filas e uma população de 64K *jobs*. As outras configurações foram simuladas utilizando-se uma única *splay tree* para armazenar a lista de eventos.

Em [Stone87], define-se *speed-up* como a razão entre o tempo para executar o programa seqüencial mais eficiente que realiza uma computação, e o tempo de execução de um programa

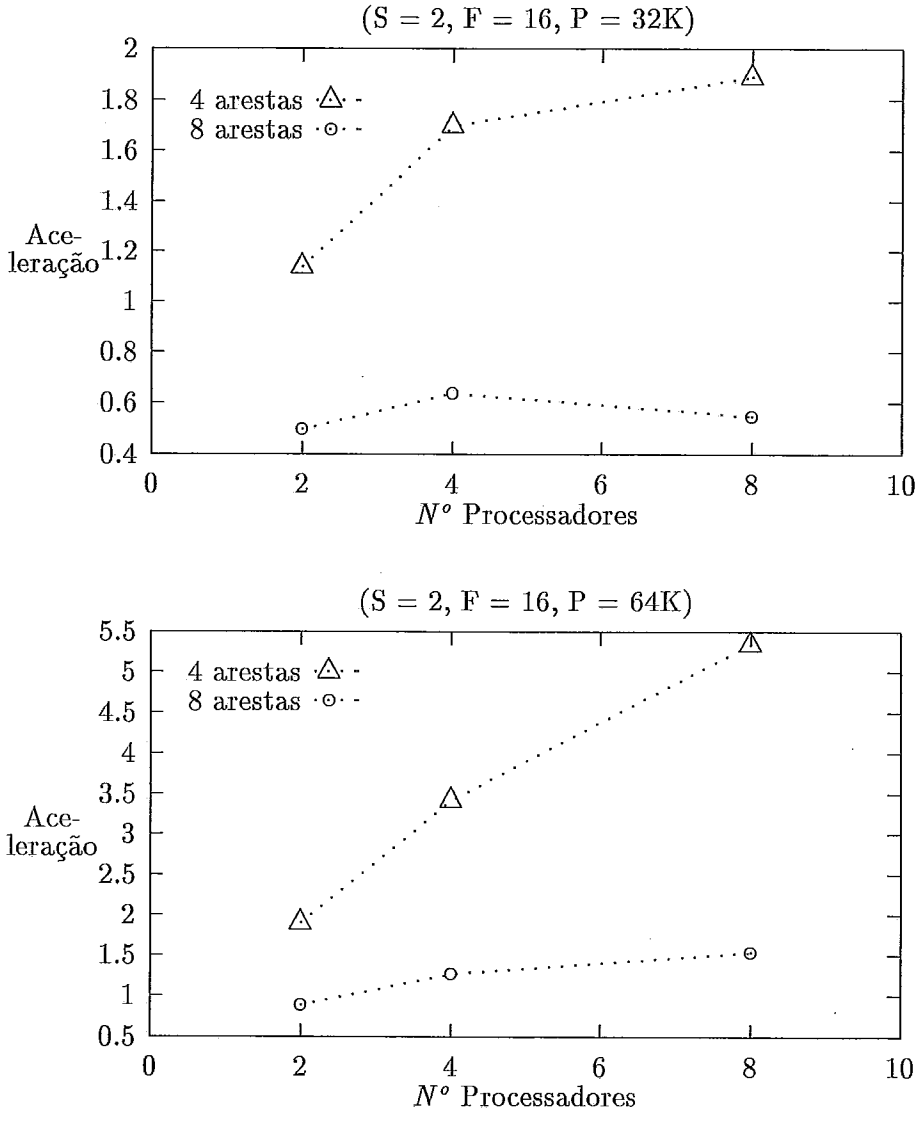

Figura 5.3: Variação do Número de Arestas com S = 2 e F = 16

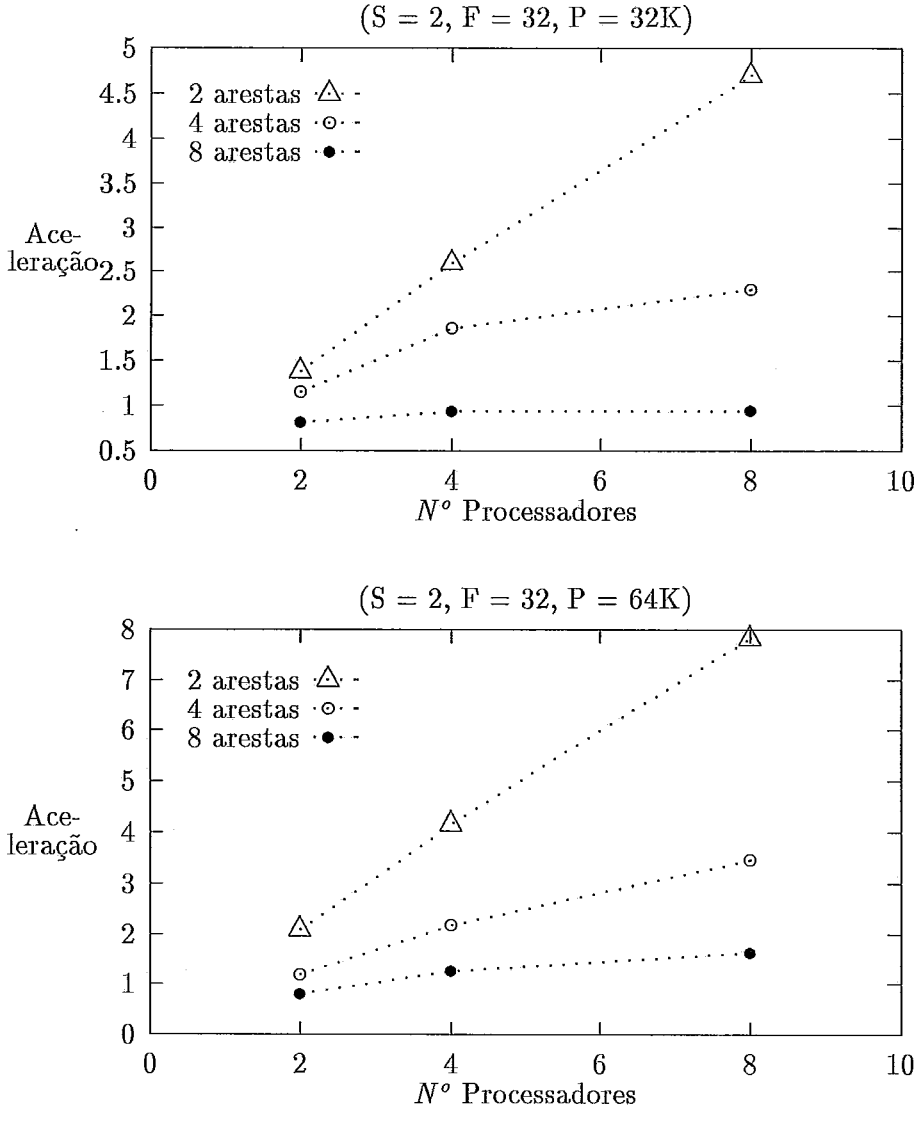

Figura 5.4: Variação do Número de Arestas com S = 2 e F = 32

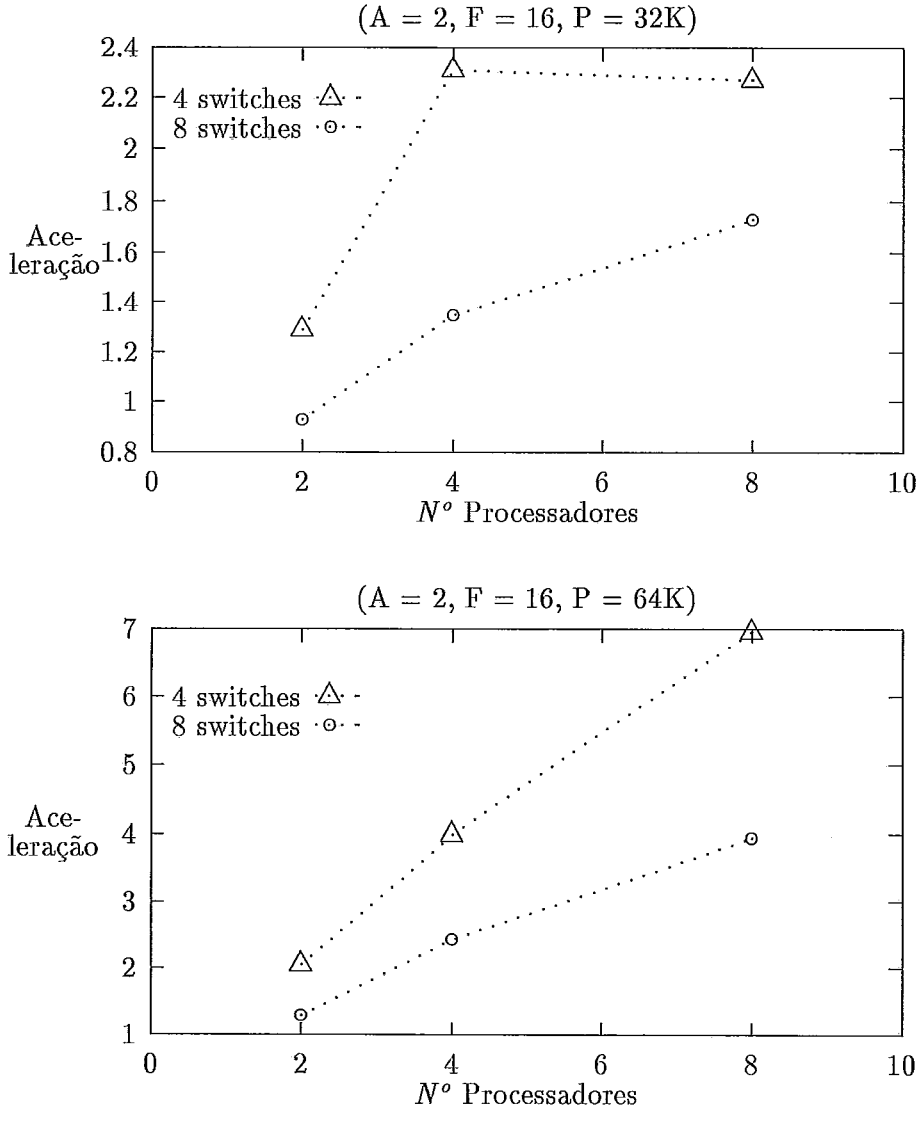

Figura 5.5: Variação do Número de *Switches* com A = 2 e F = 16

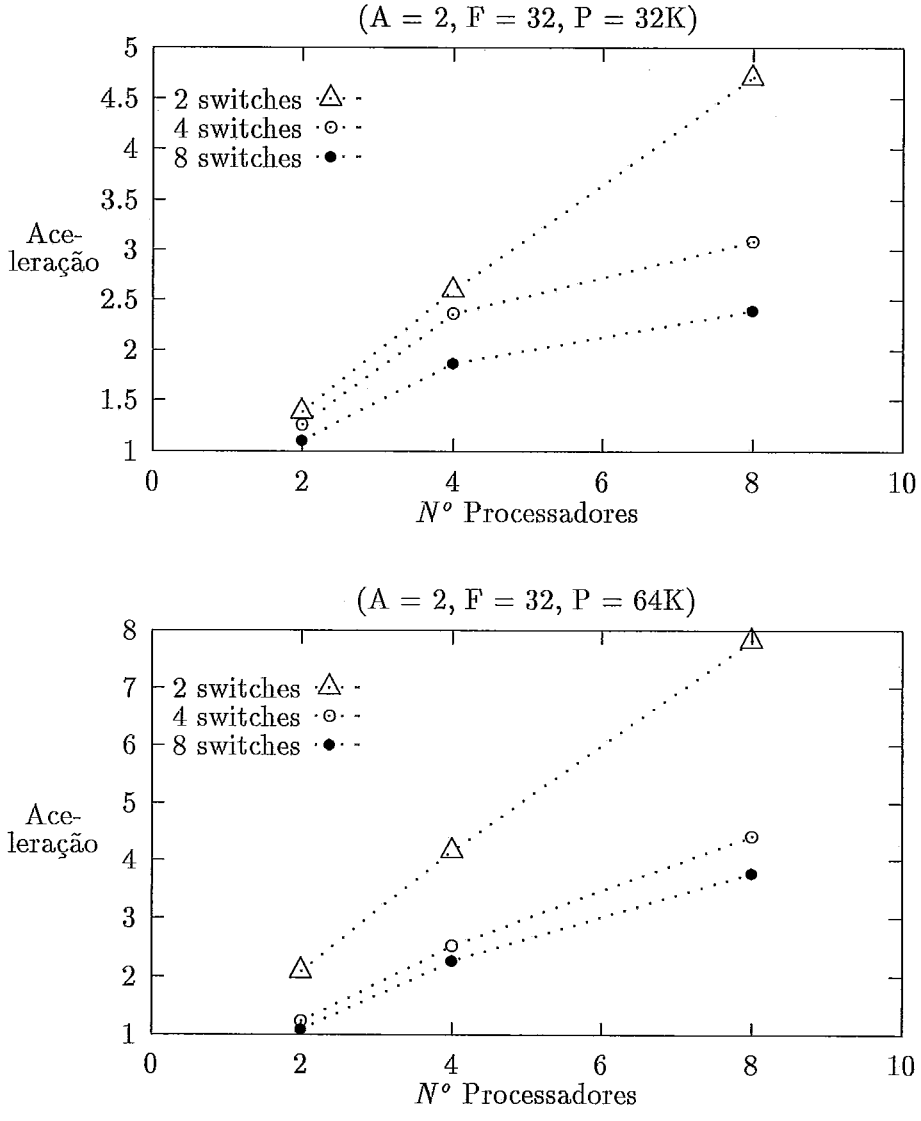

Figura 5.6: Variação do Número de *Switches* com A = 2 e F = 32

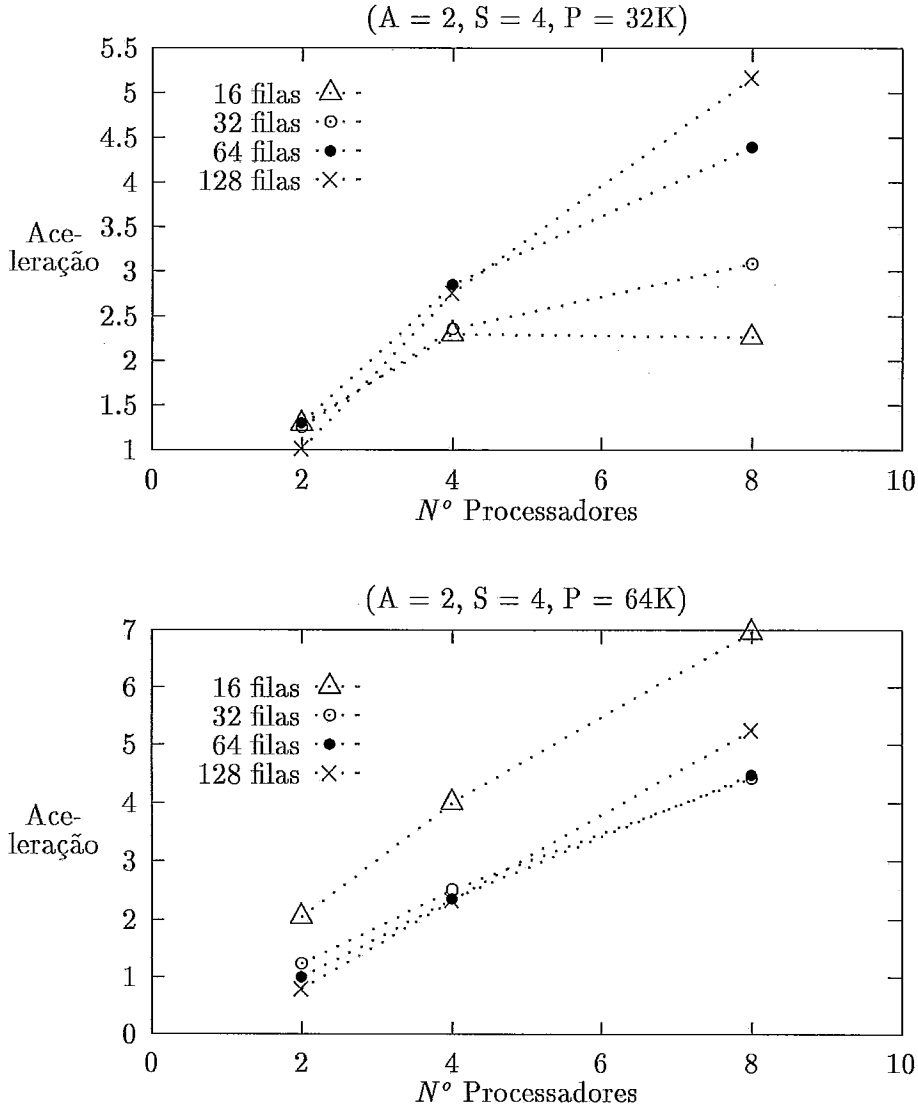

Figura 5.7: Variação do Número de Filas entre *Switches* com S = 4 e A = 2

paralelo que realiza a mesma computação em N processadores. Pela definição, o *speed-up* é limitado superiormente por N, pois se a razão entre os tempos for maior que N, então existe pelo menos um programa seqüencial mais eficiente, formado pela seqüencialização do programa paralelo.

Os tempos medidos para as simulações paralelas foram avaliados relativamente às simulações seqüenciais que se utilizavam do simulador seqüencial referido acima. Este provou não ser o mais eficiente possível, conforme requer a definição de *speed-up*, uma vez que algumas razões superaram o limite imposto pelo número de processadores usados. Por isso, define-se um novo conceito — aceleração — dada pela razão entre o tempo medido para a execução da simulação seqüencial com uma determinada configuração de parâmetros de caracterização da rede de filas, e o tempo medido para a execução da mesma simulação utilizando-se o simulador distribuído por abordagem de eventos condicionais. Os gráficos que ilustram o desempenho do modelo paralelo foram traçados com o número de processadores como uma dimensão e a aceleração como a segunda dimensão.

As figuras 5.3 a 5.7 mostram a influência das variações de A, S e F, respectivamente, no desempenho do simulador distribuído para duas populações diferentes de *jobs*. De uma maneira geral, é possível notar que quanto menor o número de arestas que partem de um *switch*, quanto menor o número de *switches*, quanto maior o número de filas na seqüência de uma aresta e quanto maior a população, melhor o desempenho da simulação. Todas as condições de melhora do desempenho explicam-se porque aumentam a quantidade de eventos definitivos no sistema, resultado este esperado dado que o simulador se baseia num modelo conservador de simulação distribuída. O modelo de teste não foi muito esclarecedor, no entanto no que se refere ao efeito da proporção entre eventos condicionais e definitivos escalonados pelos processos lógicos no desempenho da simulação. Nos testes realizados, uma grande parte dos eventos condicionais escalonados teve de ser cancelada posteriormente por erro na ordem de processamento dos eventos (ver seção 3.4). Com isso, o impacto maior causado pelos eventos condicionais deveu-se aos cancelamentos e não propriamente à espera por sua transformação em eventos definitivos.

5.3 Conclusões

Os testes realizados para a avaliação do desempenho do modelo de simulação distribuída por abordagem de eventos condicionais expuseram seus dois limites opostos de comportamento. O paradigma conservador pode ter desempenho excelente quando o sistema físico apresenta níveis suficientes de *lookahead* e o sistema lógico que o implementa é capaz de explorá-lo com eficiência. Desempenho este que pode ser qualificado como sofrível quando pouco se pode prever no que

concerne o comportamento futuro do sistema em simulação.

O ponto forte do paradigma está no fato de evitar o sobrecarregamento do sistema distribuído com mensagens nulas, tal qual o faz o modelo conservador clássico para garantir a progressão da simulação sem a ocorrência de *deadlock*. Aliado a isto, o isolamento do controle da simulação, ditado pelo paradigma, em módulos distribuídos de gerência, possibilitou uma otimização do modelo com o objetivo de diminuir o tráfego de mensagens. Em [Fujim90], Fujimoto chegara à conclusão que métodos conservadores podem ter êxito em pacotes de simulação, nos quais o algoritmo de sincronização de eventos é otimizado e ao usuário cabe a configuração de determinados módulos para a adaptação a sistemas específicos.

A concentração de tarefas inerentes ao paradigma, se permite a diminuição do tráfego de mensagens, introduz no entanto o cancelamento de eventos condicionais que não mais ocorrerão. A eliminação excessiva de eventos condicionais pode impactar substancialmente o desempenho da simulação. Há uma perda associada à gerência da estrutura de dados de armazenamento de eventos, exarcebada com a inserção e remoção de eventos inúteis.

Este texto tentou, na medida do possível, debater questões relativas a simulação distribuída em geral, apontando dificuldades encontradas e meios utilizados para combatê-las. Um número razoável de modelos, conservadores e otimistas, foi apresentado procurando dar subsídios à avaliação do paradigma seguido pelo simulador. Novos caminhos de pesquisa têm sido trilhados, porém questões continuam pendentes. Pouco tem sido mostrado, por exemplo, com relação a sistemas de tempo real. Os modelos abordados não oferecem meios de garantir que limites de tempo no processamento de eventos não sejam ultrapassados.

Um caminho de pesquisa com potencial para estender consideravelmente a aplicação de técnicas de simulação distribuída diz respeito à utilização destas na área de programação paralela em geral. Um simulador paralelo garante, ainda que processando eventos concorrentemente, que os resultados obtidos ao final da simulação seriam os mesmos se os eventos tivessem sido processados seqüencialmente, na ordem crescente dos rótulos de tempo. Por analogia, considere uma computação qualquer que possa ser dividida em tarefas e atribua a estas tarefas rótulos de tempo de acordo com a lógica do programa e de maneira a respeitar as dependências de dados entre as tarefas. Os paradigmas aqui descritos poderiam ser aplicados com o intuito de executar este conjunto de tarefas em paralelo, se estas forem tomadas como eventos a serem processados.

Esta abordagem pode ser estendida a um outro nível de granularidade. Processadores super-escalares, que despacham várias instruções simultaneamente para serem executadas nas unidades funcionais disponíveis, poderiam se basear em métodos como o *Timewarp* para o controle das dependências entre as instruções.

O simulador implementado está disponível para uso, tencionando ser uma plataforma para a execução de sistemas lógicos de simulação. Caso a aplicação se encaixe no perfil para o qual um bom desempenho pode ser alcançado, o simulador pode vir a ser uma ferramenta útil na aceleração da velocidade de execução da simulação.

Bibliografia

[ACSC91] Amorim, Claudio L.; Citro, Ricardo; Souza, Alberto Ferreira de & Chaves Filho, Eliseu M. *The NCP I Parallel Computer System.* COPPE/UFRJ, Programa de Engenharia de Sistemas e Computação, Technical Report ES-241/91, Abril 1991.

[Ayani89] Ayani, Rassul. *A Parallel Simulation Scheme Based on Distances between Objects.* Proceedings of the SCS Multiconference on Distributed Simulation, Março 1989.

[Burns88] Burns, Alan. *Programming in Occam 2.* Addison-Wesley Publishing Company, 1988.

[CL85] Chandy, K. M. & Lamport, Leslie. *Distributed Snapshots: Determining Global States of Distributed Systems.* ACM Transactions on Computer Systems, (3)1, Fevereiro 1985.

[CM79] Chandy, K. M. & Misra, J. *Distributed Simulation: a Case Study in Design and Verification of Distributed Programs.* IEEE Transactions on Software Engineering (SE-5)5, Setembro 1979.

[CM81] Chandy, K. M. & Misra, J. *Asynchronous Distributed Simulation via a Sequence of Parallel Computations.* Communications of the ACM, (24)4, Abril 1981.

[CS89a] Chandy, K. M. & Sherman, R. *The Conditional Event Approach to Distributed Simulation.* Proceedings of the SCS Multiconference on Distributed Simulation, Março 1989.

[CS89b] Chandy, K. M. & Sherman, R. *Space-Time and Simulation.* Proceedings of the SCS Multiconference on Distributed Simulation, Março 1989.

[Drumo90] Drumond, Lúcia M. de Assumpção. *Projeto e Implementação de um Processador Virtual de Comunicação.* Tese M.Sc., Universidade Federal do Rio de Janeiro, COPPE, Engenharia de Sistemas e Computação, 1990.

[Fujim90] Fujimoto, R. M. *Parallel Discrete Event Simulation.* Communications of the ACM, (33)10, Outubro 1990.

[GDM91] Ghosh, Sumit; DeBenedictis, Erik & Meng-Lin Yu. *A Provably Correct, Non-Deadlocking Parallel Event Simulation Algorithm.* Proceedings of the IEEE Annual Simulation Symposium, Abril 1991.

[GT88] Groselj, Bojan & Tropper, Carl. *The Time-of-Next-Event Algorithm.* Proceedings of the SCS Multiconference on Distributed Simulation, Julho 1988.

[Hoare78] Hoare, C. A. R. *Communicating Sequential Processes.* Communications of the ACM, (21)8, Agosto 1978.

[Inmos88] *TDS - Transputer Development System.* Inmos, Prentice-Hall Inc., 1988.

[Inmos89] *Transputer Aplications Notebook - Architecture and Software.* Inmos, 1ª Edição, Maio 1989.

[Kreut86] Kreutzer, Wolfgang. *System Simulation - Programming Styles and Languages.* Addison-Wesley Publishing Company, 1986.

[Jeffe85] Jefferson, David R. *Virtual Time.* ACM Transactions on Programming Languages and Systems, (7)3, Julho 1985.

[Lampo78] Lamport, Leslie. *Time, Clocks and the Ordering of Events in a Distributed System.* Communications of the ACM, (21)7, Julho 1978.

[Lubac89] Lubachevsky, Boris D. *Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks.* Communications of the ACM, (32)1, Janeiro 1989.

[Misra86] Misra, Jayadev. *Distributed Discrete-Event Simulation.* ACM Computing Surveys, 18(1), Março 1986.

[PWM79] Peacock, J. Kent; Wong, J. W. & Manning, Eric G. *Distributed Simulation using a Network of Processors.* Computer Networks, (3)1, Fevereiro 1979.

[SBW88] Sokol, Lisa M.; Briscoe, Duke P. & Wieland, Alexis P. *MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution.* Proceedings of the SCS Multiconference on Distributed Simulation, Julho 1988.

[SCS89] Som, T. K.; Cota, B. A. & Sargent, R. G. *On Analyzing Events to Estimate Possible SpeedUp of Parallel Discrete Event Simulation.* Proceedings of 1989 Winter Simulation Conference, Dezembro 1989.

[ST85] Sleator, Daniel Dominic & Tarjan, Robert Endre. *Self-Adjusting Binary Search Trees.* Journal of the ACM, (32)3, Julho 1985.

[Stone87] Stone, Harold S. *High-Performance Computer Architecture.* Addison-Wesley Publishing Company, 1987.

[Trive82] Trivedi, Kishor S. *Probability and Statistics with Reliability, Queuing and Computer Science Applications.* Prentice-Hall Inc., 1982.