Simulação Distribuída em um Hipercubo de Transputers com o Mecanismo Time Warp

Luiz Felipe de Lima Perrone

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENACAO DOS PROGAMAS 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 E COMPUTAÇÃO.

Aprovada por:

Prof. Valmir Carneiro Barbosa, Ph.D.

(Presidente)

Prof. Cláudio Luis de Amorim, Ph.D.

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

RIO DE JANEIRO, RJ — BRASIL
AGOSTO DE 1992

PERRONE, LUIZ FELIPE DE LIMA

Simulação Distribuída em um Hipercubo de Transputers com o Mecanismo Time Warp [Rio de Janeiro] 1992.
ix, 98 p. 29,7cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1992)
Tese – Universidade Federal do Rio de Janeiro, COPPE
1. Simulação 2. Simulação Distribuída 3. Processamento Paralelo
I. COPPE/UFRJ II. Título (série)

Agradecimentos

Ao CEPEL pelos recursos fornecidos para a realização deste trabalho, que permitiram a aquisição da bibliografia de simulação paralela praticamente inexistente nas Bibliotecas acessíveis;

Aos amigos Anna T. C. Albuquerque, Cláudia Portella, Haroldo Gamal e Ricardo C. Correa, que sempre se dispuseram a participar de discussões sobre o tema deste estudo, oferecendo muitas vezes valiosas sugestões além do grande incentivo e amizade;

Aos amigos Delfim X. Martins, Hsing Tse Hao, Eliseu Monteiro Chaves Filho e Raquel P. Coelho por tornarem o laboratório um lugar agradável de se trabalhar;

Aos amigos do grupo de simulação Luis Carlos Quintela, Nahri B. Moreano e Roseli S. Weissman por todo apoio e pela participação na definição de várias partes do sistema;

Ao meu *mui querido* orientador Valmir Caneiro Barbosa, por toda paciência, incentivo e colaboração e por ser um exemplo acadêmico, humano e moral;

A meus pais, Luiz Perrone Netto e Ilka Maria de Lima Perrone por uma vida dedicada a educação, ao desenvolvimento intelectual e moral de seus filhos, sempre nos dando o máximo possível de amor e carinho e

Ao meu irmão, André Luiz de Lima Perrone, amigo acima de tudo, alma gêmea por definição.

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

Simulação Distribuída em um Hipercubo de Transputers com o Mecanismo Time Warp

Luiz Felipe de Lima Perrone

Agosto, 1992

Orientador: Prof. Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

A simulação de eventos discretos provou estar entre os mais difíceis casos de particionamento de problemas para execução paralela. A distribuição de um problema desta natureza para execução paralela envolve bem mais que a decomposição de estruturas de dados e sincronização de iterações.

Neste tipo de simulações existe uma restrição básica quanto ao escalonamento distribuído de eventos representada pelo conceito de causalidade. O conceito causalidade determina uma ordem parcial entre os eventos da simulação a ser necessariamente respeitada para que a simulação produza resultados corretos.

No desenvolvimento de métodos para simulação paralela de eventos discretos, definiram-se duas abordagens distintas para tratar o escalonamento distribuído de eventos. Os algoritmos da escola *conservadora* procuram escalonar eventos somente ao encontrarem uma situação em que pode-se garantir a não ocorrência de erros de causalidade. Este tipo de técnica tem mostrado ao longo do tempo ser incapaz de explorar bem o paralelismo das aplicações e portanto, proporcionalmente, maiores atenções tem sido dispensadas no aprimoramento dos algoritmos da escola *otimista*. Os algoritmos otimistas baseiam-se em uma forma mais flexível de escalonamento que, ao mesmo tempo que permite ocorrência de erros de causalidade para corrigi-los em seguida, permite bom aproveitamento da concorrência natural da aplicação.

O trabalho de David Jefferson no paradigma de *Tempo Virtual* criou novas perspectivas para a simulação de eventos discretos em máquinas multiprocessadas, dentro da escola otimista. O mecanismo otimista *Time Warp*, que implementa este paradigma, tem sido cada vez mais estudado e fundamenta-se hoje como o método mais promissor no desenvolvimento de simuladores paralelos de uso genérico.

Este trabalho apresenta o projeto e análise de desempenho de um mecanismo *Time Warp* para um multiprocessador hipercúbico baseado em *transputers*. Neste projeto,

procurou-se empregar técnicas eficientes de controle de fluxo e gerenciamento de memória, desenvolvidas a partir do protocolo *Cancelback*, de Jefferson, adaptado para execução em um sistema de memória distribuída. Apresenta-se também um novo algoritmo de cálculo de *Tempo Virtual Global*, baseado no algoritmo de registro de estados globais de Chandy e Lamport, que opera em conjunto com o protocolo *Cancelback*. O projeto incorpora, além destas modificações, uma nova proposta no mecanismo de cancelamento de mensagens procurando aproveitar a idéia do *Direct Cancellation* de Richard Fujimoto, para máquinas de memória compartilhada, também em máquinas de memória distribuída.

O sistema foi escrito na linguagem de programação **occam 2** com alguns poucos trechos em *assembly language*. Devido à necessidade de utilizar um método de escalonamento de processos cuja características divergem dos algoritmos implementados em microcódigo no *transputer*, desenvolveu-se uma técnica para sobrepujar a ação normal do processador. Desta forma, apresenta-se aqui um algoritmo de escalonamento preemptivo capaz de tomar decisões próprias de escalonamento sem interferência do processador.

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

Parallel Simulation on a Transputer Hypercube with the Time Warp Mechanism

Luiz Felipe de Lima Perrone

August, 1992

Thesis Supervisor: Prof. Valmir Carneiro Barbosa

Department: Systems Engineering and Computer Science

Discrete-event simulation has proved to be among the hardest problems to be partitioned for parallel execution. Differently from other applications, its distribution involves much more than simple decomposition of data structures and synchronization of loop cycles.

In this kind of simulations, there is always a partial order of events that must be preserved to ensure the correctness of the computation. This partial order is defined by a causality relation between events that define how they must be scheduled in a way as to produce valid results.

In the development of methods for parallel execution of discrete events, two different paradigms have been created to handle the distributed scheduling of events. The *conservative* algorithms try to schedule events in such a way as to avoid the ocurrence of causality errors. This kind of technique has proved not to be able to exploit the parallelism inherent to applications in its full and, therefore, much effort has been put forth in the development of algorithms belonging to the *optimistic* school. The *optimistic* algorithms embody a more flexible event scheduling technique that as well as allowing causality errors to occur, correcting them afterwards, leads to a good exploitation of the application's parallelism.

David Jefferson's work on the *Virtual Time* paradigm has opened new perpectives in optimistic distributed simulation on multiprocessed machines. The *Time Warp* mechanism, that implements this paradigm, has been the object of extensive research and also has been said to offer a great potential as a general purpose strategy for parallel simulation.

This work presents the project and performance analysis of a modified version of the *Time Warp* mechanism for a *transputer* hypercubical multiprocessor. This project includes novel efficient techniques for flow control and memory management based on David Jefferson's *Cancelback* protocol and for message cancellation based on Richard Fujimoto's

Direct Cancellation. Originally, these were developed for shared memory machines, but here equivalent techniques for distributed memory architectures are presented. A new algorithm for *Global Virtual Time (GVT)* computation, that works together with the *Cancelback* protocol, is also presented. This algorithm is based on Chandy and Lamport's global snapshot algorithm.

The system discussed here is written mostly in the **occam 2** programming language with very few parts in assembly. The *Time Warp* mechanism has scheduling requirements that are essentially different from the *transputer* microcoded scheduler, and thus a strategy to override processor control had to be devised. This strategy, used to implement a preemptive scheduling algorithm, is also presented here and can be used as a model for other works when the ultimate scheduling decision must be made without processor interference.

Índice

1 Introdução 1

1.1 Motivação 1

1.2 Organização do Texto 4

2 Uma Visão Geral de Sistemas, Modelos e Simuladores 6

2.1 Conceitos Básicos em Simulação 6

2.2 Simulação de Eventos Discretos 9

2.2.1 Simuladores Seqüenciais 13

2.2.2 Simuladores Paralelos 16

3 Tempo Virtual e o Mecanismo Time Warp 18

3.1 Tempo Virtual 19

3.2 O mecanismo Time Warp 21

3.2.1 O Mecanismo de Rollback 24

3.2.2 Cálculo de *GVT* 29

3.2.3 Controle de Fluxo e Gerência de Memória 29

4 A linguagem occam e os *transputers* 33

4.1 A linguagem de programação **occam** 34

4.1.1 Processos primitivos 34

4.1.2 Canais e protocolos 35
4.1.3 Estruturas de programação 37
4.2 Implementação do modelo **occam** no T800 40
4.2.1 Escalonamento de Processos 41
4.2.2 Canais e o Modelo de Comunicação 42
4.3 Um mecanismo de preempção de processos **occam** no *transputer* 43
4.3.1 Arquitetura do Mecanismo de Preempção 44
4.3.2 A Realização do Mecanismo 47

5 Uma Versão de *Time Warp* **52**
5.1 Visão geral 52
5.2 Os Processos Lógicos 56
5.3 O gerente *Time Warp* 59
5.3.1 Rollback e Cancelamento de Mensagens 62
5.3.2 O algoritmo de Cálculo de *GVT* 66
5.3.3 Uma Adaptação do Protocolo *Cancelback* 72

6 Avaliação Experimental **79**
6.1 O modelo de avaliação de desempenho 79
6.1.1 Localidade Espacial 81
6.1.2 Granularidade 83
6.1.3 Localidade Temporal 84
6.1.4 Lookahead 84
6.2 Resultados Experimentais 87

7 Conclusões **93**

Lista de Figuras

2.1 Processamento de eventos em um processo lógico 12

2.2 Algoritmo de Controle de um Simulador Seqüencial 15

3.1 Estrutura de dados de uma mensagem no *Time Warp* 22

3.2 Estruturas de dados de um processo lógico no *Time Warp* 25

3.3 Exemplo de uma simulação com problemas de fluxo de mensagens 31

4.1 Exemplo de declaração e uso de protocolos simples 36

4.2 Exemplo de declaração e uso de protocolos seqüenciais 37

4.3 Exemplo de declaração e uso de protocolos variantes 38

4.4 Arquitetura de Processos de um Escalonador Preemptivo 45

4.5 Algoritmo do mecanismo de preempção 50

5.1 Arquitetura de um Mecanismo *Time Warp* para *transputers* 53

5.2 Modelo sintético de um processo lógico 56

5.3 Algoritmo básico da Seção de Tratamento de Eventos 58

5.4 Algoritmo básico do processo *Time Warp* 60

5.5 Exemplo de *rollback* por chegada de mensagem retardatária 64

5.6 Exemplo de *rollback* por chegada de anti–mensagem 64

5.7 Exemplo de árvore de causalidade 67

5.8 Algoritmo para registro de estados globais do *Time Warp* 70

5.9 Algoritmo do protocolo *Cancelback* 77

6.1 Exemplo de um diagrama espaço–tempo 80
6.2 Rede toroidal usada na avaliação de desempenho 82
6.3 Modelagem de lookahead 86
6.4 *Speed-up* em um hipercubo com 8 processadores 89
6.5 Tempo de processamento para simulação sem *rollbacks* 90
6.6 Tempo de processamento para simulação com *rollbacks* 91

Capítulo 1

Introdução

1.1 Motivação

No estudo de muitas áreas do conhecimento humano desenvolvem-se sistemas e hipóteses que representam grandes desafios aos seus criadores. Quando é impossível ou muito complexa a validação analítica destes sistemas, a alternativa que se apresenta é a validação através de métodos empíricos.

Os custos, riscos ou o tempo de observação associados a um experimento nem sempre tornam viável a sua realização. Para estes casos a técnica de simulação em computador se adapta perfeitamente. Um modelo simplificado do sistema em estudo pode ser simulado em um computador pertimitindo absorver um conhecimento aproximado de seu comportamento real. Os custos relativos são muito pequenos e a segurança do observador absoluta. Muitas vezes consegue-se, ainda, reduções drásticas no tempo de observação.

Desde o desenvolvimento do primeiro computador digital, explora-se o potencial destas máquinas na simulação dos mais diversos sistemas. Modelos biológicos tem sido usados para prever o comportamento de populações no estudo de ecossistemas. Jogos de guerra permitem avaliar o resultado de diferentes estratégias de combate sem envolver nenhum contingente humano ou material bélico. Novas arquiteturas de computador podem ter seu desempenho avaliado e seu projeto refinado, em sucessivas iterações, até que se atinjam as metas desejadas sem que se despenda material algum.

Como várias outras aplicações para computador, a complexidade das simulações avança sempre mais rápido que o potencial das máquinas. Os requisitos das simulações exigem sempre mais do que os computadores podem oferecer e, por isso, busca-se incessantemente desenvolver técnicas que permitam acelerá-las.

O surgimento de supercomputadores permitiu grandes avanços para uma série de aplicações. Infelizmente, entre estas não se enqüadram muitos casos de simulação, especialmente as simulações de eventos discretos. Muitos modelos de simulação apresentam um alto grau de paralelismo, que não é bem explorado por máquinas com esta nova tecnologia.

As máquinas paralelas, com múltiplos processadores, constituem um ambiente potencialmente mais adequado a este tipo de aplicação. Como veremos mais tarde, muitos modelos são descritos por várias entidades que trabalham em paralelo, como no mundo real, e que por definição adequariam-se perfeitamente à execução em máquinas multiprocessadas.

No entanto, como é de conhecimento geral, não é suficiente possuir tecnologia a nível de *hardware* para que se obtenha grandes acelerações a nível de *software*. Os problemas em simulações realizadas em máquinas paralelas são análogos aos de outras aplicações. É preciso definir e empregar técnicas de sincronização, comunicação, balanceamento de carga, terminação, etc.

A tarefa de paralelização de simuladores seqüenciais, que pode parecer simples a primeira vista devido a estrutura concorrente dos próprios modelos, é, no entanto, bastante complexa. A paralelização de outras aplicações é relativamente mais fácil; muitas vezes basta que se defina uma forma de particionamento de dados, exigindo poucas alterações sobre o algoritmo original. O desenvolvimento de técnicas para simulação em arquiteturas paralelas requer estudos profundos sobre os mais variados temas em processamento paralelo. Este campo de estudos é, ao mesmo tempo, promissor e problemático. Promissor, porque a aplicação exibe alto grau de paralelismo e, portanto, potencialmente atingiria grandes acelerações em sua execução paralela. E problemático, porque para permitir que estas grandes acelerações sejam alcançadas exige soluções eficientes para problemas que representam os maiores desafios em processamento paralelo.

Para estudar e resolver os problemas específicos de simulação em máquina paralelas foi criada uma linha de pesquisas denominada de *simulação paralela*. Esta linha estuda o desenvolvimento de técnicas que beneficiem diretamente os problemas de simulação, mas que obviamente termina por solucionar problemas genéricos de processamento paralelo.

Neste trabalho, estudaremos um caso específico em simulação paralela. O paradigma de *Tempo Virtual*, apresentado por Jefferson [1] no início da década de 80 e orientado para a paralelização de simulações de eventos discretos, mobiliza hoje a atenção de um grande número de pesquisadores. Ao contrário de outros paradigmas, o *Tempo Virtual*, implementado pelo mecanismo *Time Warp*, se apresenta como uma ferramenta genérica de simulação paralela. Da realização de vários experimentos nesse ambiente pode-se concluir que os altos valores de *speed-up* conseguidos não são resultado do uso de um pequeno conjunto de aplicações com características particulares.

A experiência com o *Time Warp*, em arquiteturas com memória distribuída ou compartilhada, tem provado que sob as mais diversas aplicações e condições o mecanismo tem um grande potencial a ser explorado. Desde o projeto inicial, na Rand Corporation, para uma rede de estações de trabalho Xerox SIP–1100, o mecanismo foi portado para outras máquinas como os hipercubos Mark II e Mark III, redes de estações de trabalho

Sun, diferente modelos de BBN Butterfly e redes de *transputers*.

O mecanismo *Time Warp* possui várias características interessantes. É absolutamente simétrico e, portanto, facilmente portável e escalável. Em sua definição, o mecanismo não se liga a nenhuma arquitetura específica ou exige determinado número de processadores, o que permite sua implementação nas mais diferentes máquinas. Exige confiabilidade do sistema de comunicação entre processadores, mas, por outro lado, não requer preservação da ordem de envio das mensagens.

Diferente de outros paradigmas de simulação paralela, o *Time Warp* é transparente ao programa que descreve o modelo simulado. Em outras palavras, não requer declaração da topologia de interconexão dos processos do modelo. Na verdade, na forma como tem sido implementado, consiste de uma camada sobre a máquina provendo serviços similares aos de um sistema operacional. Estes serviços restringem-se às necessidades básicas de uma simulação paralela como manutenção da ordem da causalidade entre eventos, controle de fluxo e gerência de memória.

Por estas características, o *Time Warp* tem sido apontado como a maior promessa de simulador paralelo de uso genérico [8]. Várias aplicações de diferentes naturezas foram testadas com sucesso neste ambiente. Simulações de combate, ou jogos de guerra, como COMMO* [2], CTLS87 [3], STB87 [13] e STB88 [14, 15, 16] atingiram valores de eficiência (*speed-up* dividido pelo número de processadores) da ordem de 50%. Experimentos com modelos biológicos simples como o *Game of Life* [2], *Antopia* [20] e *Sharks World* [23] produziram resultados semelhantes. Outras aplicações como um modelo de bolas de bilhar, conhecido como *Colliding Pucks* [4, 21, 22] e um modelo de rede de comunicação de computadores, *Warpnet* [24] alcançaram desempenhos na mesma ordem.

Não se deve, no entanto, acreditar que estes resultados revelem uma tendência de *upper bound* nos ganhos de desempenho conseguidos com o mecanismo. Fujimoto, ao realizar experimentos com redes de filas em uma máquina de memória compartilhada [9], conseguiu mostrar que com melhorias estruturais, o *Time Warp* pode levar a desempenhos fantásticos. Nesta aplicação conseguiram-se *speed-ups* de até 56.8 em um BBN Butterfly–1 com 64 processadores. Mais ainda, seus experimentos indicam que os resultados obtidos são escaláveis com o tamanho do problema simulado e com o número de processadores da máquina empregada.

A principal característica que distingüe o *Time Warp* de outros paradigmas de simulação paralela é o fato de este permitir que uma aplicação alcance *speed-ups* superiores ao previsto teoricamente por análise de caminho crítico [25]. Experimentos com o jogo de guerra STB88 já demonstram essa possibilidade [18]. Muitos outros estudos comprovam analítica e experimentalmente o potencial deste mecanismo, mostrando ainda que admite otimizações estruturais e algorítmicas.

Neste trabalho, estudamos profundamente o mecanismo *Time Warp*, procurando identificar pontos a serem trabalhados em sua otimização. Seguindo um enfoque não usual na literatura corrente, descrevemos o mecanismo mostrando a inter-relação de suas várias partes. Desse modo, apresentamos soluções não comentadas para algumas dificuldades de projeto identificadas e ainda propomos alterações de aspectos já largamente estudados.

Apresentamos uma nova estratégia de cancelamento de mensagens baseada em uma adaptação de técnicas para máquinas com memória compartilhada à uma máquina com memória distribuída. Apresentamos, também, um novo algoritmo de estimação de *Tempo Virtual Global*, ou *GVT*, e uma implementação distribuída do protocolo de controle de fluxo e cancelamento de memória *Cancelback*, introduzido recentemente por Jefferson [5].

Para avaliar o mecanismo na estrutura aqui mostrada, desenvolvemos uma versão do *Time Warp* para execução em um hipercubo de *transputers* T800 escrita na linguagem de programação **occam**. Esta plataforma paralela tem demonstrado grande potencial para execução de aplicações que exigem alto desempenho. No entanto, pelo projeto dos *transputers* e da linguagem de programação, a implementação de um mecanismo como o *Time Warp* não é muito simples. Dificuldades na implementação de um algoritmo de escalonamento sobre o escalonador em *hardware* dos processadores e na criação de um mecanismo de preempção não previsto na linguagem exigiram soluções não triviais que podem ser aproveitadas em outros programas.

Para avaliar o desempenho desta versão do mecanismo optamos pela utilização de uma carga sintética como aplicação. O desenvolvimento de aplicações reais em simulação paralela já é por si só um trabalho bastante complexo e consumidor de grande tempo. Com o uso de um modelo adequado de carga sintética é possível exercitar diferentes aspectos do simulador paralelo através da variação de alguns parâmetros.

Com o modelo PHOLD, descrito por Fujimoto [8], é possível realizar experimentos simulando características de diferentes aplicações e obter resultados que levam a conclusões mais abrangentes. Estes resultados não estariam, então, limitados a mostrar o comportamento do mecanismo em função de características particulares de uma aplicação. Desta forma, esperamos poder predizer o desempenho de uma variedade de tipos de aplicações reais sem ter, necessariamente, mergulhado nos detalhes mais complexos da implementação de cada uma.

1.2 Organização do Texto

Este trabalho está organizado em sete capítulos.

O Capítulo 2 introduz os conceitos mais importantes em simulação definindo uma terminologia básica que será empregada ao longo do texto. Iniciando pela classificação dos tipos de simulação, apresenta-se, de forma resumida, sucessivas categorizações até chegar à classe a que pertence o objeto deste estudo: a Simulação Distribuída de Eventos Discretos. Alguns outros paradigmas são apresentados para melhor situar o Time Warp no cenário atual de simulação.

O Capítulo 3 apresenta o conceito de *Tempo Virtual*, base teórica do mecanismo *Time Warp*, que é em seguida detalhado em sua estrutura funcional. Neste capítulo discutem-se as técnicas mais comuns e eficientes para tratar questões como escalonamento de processos, manipulação de mensagens, controle de fluxo e gerenciamento de memória no escopo do mecanismo.

O Capítulo 4 é dedicado a expor o ambiente onde foi desenvolvida a implementação deste trabalho. O capítulo inicia apresentando a linguagem de programação **occam**, seu modelo de programação e estruturas principais. Em seguida, discutide-se o aspecto técnico do projeto dos *transputers* que mais dificuldades impõe aos programadores trabalhando a nível de sistema operacional: o escalonamento de processos. Descreve-se um modelo simplificado da arquitetura do processador e seu mecanismo de escalonamento microprogramado, finalizando com a apresentação de um mecanismo de preempção de processos.

O Capítulo 5 apresenta o projeto e a realização de um mecanismo *Time Warp* para execução em *transputers*. Esta versão do mecanismo incorpora uma estratégia não convencional de cancelamento de mensagens, resultante da integração de técnicas para máquinas de memória compartilhada e distribuída. O algoritmo de cálculo de *Tempo Virtual Global (GVT)* utilizado difere também algoritmos tradicionais, baseando-se no algoritmo de registro de estados globais de Chandy e Lamport. Em gerenciamento de memória e controle de fluxo, é introduzida uma variação do protolo *Cancelback* para arquiteturas de memória distribuída.

O Capítulo 6 trata de avaliação de desempenho. Iniciando pela descrição de um modelo flexível de carga sintética, o capítulo prossegue apresentando os resultados obtidos através de experimentos com esta aplicação. Em seguida, analisa estes resultados oferecendo explicações para o comportamento observado em função dos parâmetros variados.

O Capítulo 7 aborda as conclusões obtidas com esta pesquisa. Discute, também, sugestões para trabalhos futuros e indica caminhos para otimizações e extensões do mecanismo aqui proposto.

Capítulo 2

Uma Visão Geral de Sistemas, Modelos e Simuladores

Este capítulo apresenta conceitos básicos em simulação que serão empregados no restante do texto. Ao mesmo tempo em que introduzem-se termos e conceitos, discute-se de maneira concisa as técnicas mais significativas no cenário atual de simulação.

Inicia-se a apresentação com uma breve ilustração da técnica mais genérica de simulação. O capítulo prossegue diferenciando as classes básicas de *Simulação Dirigida por Tempo* e *Simulação Dirigida por Eventos*, focalizando a discussão, a partir de então, nesta última.

Estudando *Simulação Dirigida por Eventos*, também referenciada na literatura como *Simulação de Eventos Discretos*, verificam-se diferentes paradigmas. A simulação seqüencial, tradicional e potencialmente muito lenta, é aqui representada por seu algoritmo básico com uma revisão das estruturas de dados mais adequadas à sua implementação. No campo da simulação paralela de eventos discretos, discutem-se os paradigmas mais importantes apresentando-se suas vantagens e desvantagens.

2.1 Conceitos Básicos em Simulação

As técnicas de simulação permitem que através do uso de um modelo simplificado de um sistema possa-se ganhar conhecimento de seu comportamento sob uma variedade de circunstâncias [26]. Para a obtenção de um modelo operacional a partir do qual seja possível obter resultados dentro de uma margem de precisão pré-estabelecida executa-se um processo iterativo de refinamento.

1. Estudando o sistema a ser simulado, procura-se extrair um conjunto de características e propriedades que o descrevam em sua essência. Este conjunto receberá o nome de *modelo*. A partir deste modelo será construído um programa de computador que uma vez executado fornecerá estimativas do comportamento do sistema real. Tanto mais próximos dos resultados com o sistema real serão os obtidos com a execução do modelo, quanto mais completa a sua descrição do sistema. Por outro lado, quanto mais completa e complexa esta descrição, mais lenta será sua execução. Esta etapa é crítica no processo de simulação: um modelo muito complexo, apesar de muito preciso será difícil de trabalhar, ao passo que um modelo muito simplificado poderá não representar com fidelidade o sistema em estudo.

2. Sobre o modelo descrito na etapa anterior, constrói-se um programa de computador. O programa deverá permitir que com o ajuste de alguns parâmetros seja possível criar diferentes cenários de experimentação. Neste programa, o parâmetro mais importante será sempre o *tempo*. O tempo determina dois aspectos importantes da simulação: progresso e terminação.

3. Executa-se o programa sob condições em que o comportamento do sistema real seja conhecido. Comparando resultados obtidos a partir do modelo computacional com o comportamento do sistema real pode-se avaliar a validade dos resultados da simulação. Caso os resultados não aproximem os valores esperados dentro de uma margem de erro arbitrária, retorma-se o processo partindo da primeira etapa. Se ainda assim o modelo puder ser considerado válido e completo, deve-se buscar corrigir sua implementação computacional.

4. Uma vez que o modelo tenha sido suficientemente depurado, realizam-se os diversos experimentos desejados. Nesse etapa poderão ser criadas diferentes situações sob as quais se deseja prever o comportamento do sistema real. Os resultados obtidos serão suficientemente precisos a ponto de poder prever o comportamento do sistema nas situações construídas artificialmente.

A forma como o tempo é tratado em um modelo de sistema determina o tipo de simulação que será realizado. Duas abordagens distintas são possíveis e cada uma adequa-se a uma classe específica de simulação. No entanto, qualquer que seja a abordagem o tempo simulado segue uma única definição apresentada a seguir.

Definição 2.1 Tempo Simulado. *Variável não negativa, sem relação direta com o tempo real, cujo valor aumenta monotonicamente durante a simulação.* □

Na primeira abordagem, mais adequada a modelos que podem ser descritos por sistemas de equações diferenciais, o tempo é tratado como uma grandeza contínua.

Definição 2.2 Simulação Dirigida por Tempo. *Partindo de um dado estado inicial* $S(t_0)$ *que representa o valor das variáveis do modelo em um tempo* t_0*, computa-se os estados subseqüentes* $\{S(t_0+\delta),\ S(t_0+2\delta)\ \ldots,\ S(T)\}$*. Um estado qualquer* $S(t_0+i\delta)$ *é determinado em função do estado anterior* $S(t_0+(i-1)\delta)$ *e do tempo atual.* $T = t_0 + n\delta$*,*

para um valor arbitrário de n, é um instante que marca o final do período de observação do modelo. O valor de T é referenciado na literatura como horizonte da simulação.

Uma simulação como esta recebe a denominação de *Simulação Dirigida por Tempo*, ou *Time Driven Simulation*. Com este tipo de simulação estudam-se sistemas que evoluem continuamente no tempo e por isso os modelos construídos, tentando aproximar este comportamento, procuram repetidamente computar aproximações do estado do sistema a intervalos muito pequenos.

Neste tipo de simulação o tempo evolui em pequenos incrementos δ constantes chamados na literatura de *time unit granularity*, ou granularidade de unidade temporal. A estimativa deste valor representa sempre uma solução de compromisso entre precisão e eficiência e é uma etapa delicada no processo de simulação. Um valor muito grande de δ poderia fazer com que o estado do modelo fosse computado em instantes por demais espaçados no tempo para espelhar com precisão mudanças rápidas no estado do sistema real. Por outro lado, com um δ muito pequeno e um modelo cujo estado tende a mudar mais lentamente no tempo, o simulador dispenderia tempo de computação calculando repetidamente os mesmos valores de estado.

Nem todos os sistemas podem ser descritos por modelos contínuos no tempo. Como dito anteriormente, esta metodologia de simulação adequa-se a modelos matemáticos contínuos em sua essência e há de se convir que muitos e muitos sistemas não admitem este tipo de descrição. Existem até sistemas não contínuos que poderiam ser assim simulados, mas não pode-se negar que exigiriam um custo computacional bem maior que o estritamente necessário.

Há sistemas que sofrem mudanças de estado somente em pontos discretos do tempo espaçados por intervalos irregulares. Para este tipo de sistema, um simulador eficiente somente computaria um novo estado do modelo ao detectar a ocorrência de uma situação que promoveria uma alteração sobre o último estado computado. A simulação de um sistema com esta característica recebe o nome de *Simulação Dirigida por Eventos* ou *Event Driven Simulation*.

Definição 2.3 Evento–discreto *ou* Evento. *Mudança instantânea no estado do sistema que ocorre em um ponto bem definido do tempo.*

Nestas simulações, o tempo simulado não guarda relação direta com o tempo real e evolui de maneira irregular evitando percorrer os intervalos em que o estado do modelo permaneceria estático. Assim, o tempo simulado pode avançar mais rapidamente levando o modelo a uma maior eficiência computacional.

Uma enorme variedade de sistemas, entre os quais redes de computadores, jogos de guerra e sistemas de objetos móveis interagindo no espaço, se mostra adequada a este tipo de modelagem e simulação.

A modelagem de sistemas para Simulações Dirigidas por Eventos segue uma abordagem estruturada de forma bastante diferente das modelagens para os sistemas contínuos

das Simulações Dirigidas por Tempo. A partir da próxima seção passamos a enfocar somente esta primeira forma de modelagem, definindo-a e detalhando-a para o prosseguimento deste estudo.

2.2 Simulação de Eventos Discretos

Os modelos para Simulações de Eventos Discretos costumam ser descritos de maneira bastante estruturada, seguindo uma visão orientada a processos ou mesmo a objetos. Observando o sistema a ser estudado procura-se identificar entidades autônomas que cooperam de alguma forma para seu funcionamento. As comunicações entre estas entidades, consideradas eventos no sistema, são modeladas por mensagens que fluem através de canais que as interligam.

Segundo a metodologia criada por Chandy e Misra [27], os modelos para este tipos de simulação constituem um *Sistema Físico (SF)* que sintetiza as características principais do sistema real. O sistema físico é constituído de *processos físicos* interligados por *canais* por onde fluem *mensagens*. Neste ambiente, o tempo é marcado por uma base de tempo única, comum a todos os processos.

O funcionamento do *SF* ao longo de um intervalo de tempo produz resultados que aproximam o comportamento do sistema real dentro deste mesmo intervalo.

Definição 2.4 Processo Físico. *Entidade autônoma que modela uma parte bem definida do sistema modelado. O processo é caracterizado por um conjunto de parâmetros, que define um estado local, e um conjunto de atividades a serem executadas quando da ocorrência de um evento. No início dos tempos, os parâmetros do processo físico possuem valores que definem seu* estado inicial. □

Definição 2.5 Canal. *Inteligação unidirecional que permite a comunicação confiável entre dois processos físicos. A ordem dos dados comunicados é preservada (propriedade FIFO).* □

Definição 2.6 Mensagem. *Unidade de dados usada em uma comunicação entre processos. Uma mensagem se caracteriza por quatro valores além dos dados que carrega: um tempo de envio, uma identificação do remetente, um tempo de recepção e uma identificação do destinatário. O tempo de recepção de uma mensagem qualquer deve ser sempre não inferior ao seu tempo de envio, caso contrário se estabeleceria a situação absurda em que um dado seria entregue a um destinatário antes mesmo de ter sido enviado.* □

Em uma visão mais genérica, pode-se dizer que um *evento* marca o tempo de ínicio ou término de um conjunto de atividades, a ocorrência de uma determinada ação ou a passagem de tempo. Um evento pode, por exemplo, indicar que uma entidade começa a se mover, que uma mensagem foi gerada ou que se excedeu um limite de tempo. Daqui por diante, no entanto, consideraremos que um evento, em um sistema, marca a recepção de

uma mensagem. Por este motivo, no restante deste trabalho, os termos *evento* e *mensagem* poderão ser usados alternadamente uma vez que, semanticamente, se confundem. Daqui por diante usaremos a notação $E(t,p)$ para representar eventos, onde t indica o tempo de ocorrência do evento E no processo identificado por p.

Segundo Misra [29], para qualquer *SF* imaginável duas condições devem ser satisfeitas. O sistema deve ser *realizável* e *predizível*.

Definição 2.7 Realizabilidade. *Uma mensagem enviada por um* PF *em um tempo* t *é função do estado do processo no tempo* t*) e do conjunto de mensagens que foram recebidas em tempos menores ou iguais a* t. □

Da condição de realizabilidade deriva-se o conceito mais importante em simulação de eventos–discretos: *o conceito de causalidade*. Suponhamos que um evento $E(t',p')$ seja criado em um processo p no tempo t. Este evento é conseqüência de toda a seqüência de eventos ocorridos no processo p desde seu estado inicial, que determina o estado de p no tempo t, e de um evento $E(t,p)$. Ao ser processado em p' no tempo t', o evento $E(t',p')$ provoca uma alteração de estado e possivelmente a criação de outros eventos.

Dizemos que existe uma relação de dependência entre $E(t,p)$ e $E(t',p')$ que define uma ordem em que os eventos devem ocorrer no sistema. Considerando os eventos ocorridos desde o tempo inicial t_0 até o horizonte da simulação T, podemos estender estas relações de dependência construindo cadeias de causalidade começando pelos eventos iniciais e terminando pelos últimos eventos simulados. Claramente, uma condição mínima deve ser garantida neste sistema: nenhum evento pode ocorrer até que todos os outros dos quais este depende tenham ocorrido. Além, disso exige-se que não exista uma dependência cíclica entre os eventos, ou seja, esta relação deve definir uma ordem parcial irreflexiva.

Definição 2.8 Predizibilidade. *Suponhamos que o sistema físico possua algum ciclo. Existe pelo menos um conjunto de processos* $\{ PF_0, \ldots PF_{n-1} \}$*, onde* PF_i *envia mensagens para* PF_{i+1}*, e possivelmente para outros PFs, e recebe mensagens de* PF_{i-1}*, e possivelmente de outros PFs (considerando que a aritmética de índices de* PFs *seja módulo n). Pela propriedade de Realizabilidade, se um* PF_i *envia alguma mensagem em um tempo t, esta dependeria de todas as mensagens que* PF_i *recebeu até t e do seu estado em t, para qualquer i. Não é difícil perceber que a organização dos PFs em ciclo, como neste caso, levaria uma situação inconsistente em que uma mensagem qualquer no ciclo seria definida recursivamente, ou melhor, em função de si própria. Para evitar esta situação, requer-se que em todo ciclo exista um incremento não nulo nos tempos de envio das mensagens. Deve existir ao menos um PF no ciclo para o qual as mensagens enviadas se caracterizam por uma diferença positiva* ε *entre os tempos de envio e recepção. Desse modo qualquer mensagem enviada ao longo do ciclo, em um tempo* $t + \varepsilon$*, será determinada em função do conjunto de mensagens recebidas em tempos menores ou iguais a t e do estado do PF no tempo t. Diz-se que um sistema construído com esta propriedade é garantidamente* predizível, *pois a partir dos eventos iniciais da simulação e dos estados iniciais dos processos, a saída de todo e qualquer* PF *até um tempo t pode ser corretamente computada.* □

Sobre esta definição de modelo constroem-se *Sistemas Lógicos (SL)*, que são a tradução para uma linguagem computacional da estrutura e comportamento do sistema físico. Cada processo físico é simulado por um *processo lógico* e os canais são simulados por trocas de mensagens entre os processos lógicos. Assim, obtém-se uma forma de descrição do modelo constituindo um programa de computador que executado fornece informações aproximadas do comportamento do sistema real, em outras palavras, um *simulador*.

Definição 2.9 Processo Lógico. *Descrição computacional de um processo físico. Possui um identificador único no sistema, um conjunto de variáveis que define seu estado local e um trecho de código seqüencial a ser executado quando da recepção de mensagens. O valor de suas variáveis de estado no início do tempo simulado define o estado inicial do processo. Dependendo da linguagem de programação empregada pode ser implementado como objeto, processo ou outra estrutura de programação.* □

Formalmente, um SL é um sistema distribuído ponto-a-ponto em sua representação mais genérica. O sistema é representado por $G = (N, C)$, um grafo orientado em que cada nó em N representa um PL e cada arco em C um canal unidirecional de comunicação.

Neste sistema, executa uma computação distribuída descrita por uma coleção de eventos que produzem alterações de estado localmente em cada nó do grafo. Cada um destes eventos é representado por uma lista de valores esquematizada na Figura 2.1 e explicada abaixo:

- PL_i, o processo lógico onde ocorre o evento;
- M_{in}, um conjunto com 1 ou mais mensagens, recebidas com tempo de recepção t por PL_i;
- S, o estado local de PL_i antes do evento;
- M_{out}, um conjunto com 0 ou mais mensagens enviadas com tempo de recepção não menor que t;
- S', o estado local de PL_i após o evento.

Uma grandeza adicional representada na Figura 2.1, que no entanto não faz parte da descrição formal de um evento, é o incremento Δ entre o tempo de recepção de um evento (t_r) e tempo de envio do seguinte(t_s). Este incremento pode ser usado para modelar o tempo de processamento do evento e assumir valores constantes ou variáveis durante uma simulação. A forma como são determinados estes valores faz parte da etapa de modelagem, mas depende unicamente da aplicação em estudo.

O processo PL_i é ativado a cada ocorrência de evento e assim processando uma seqüência qualquer de eventos produz uma seqüência associada de estado locais. Os estados do SF simulado pelo SL, são dados pelos estados globais deste último. Dizemos que um SL simula corretamente um SF, quando em sua execução, prevê exatamente a seqüência

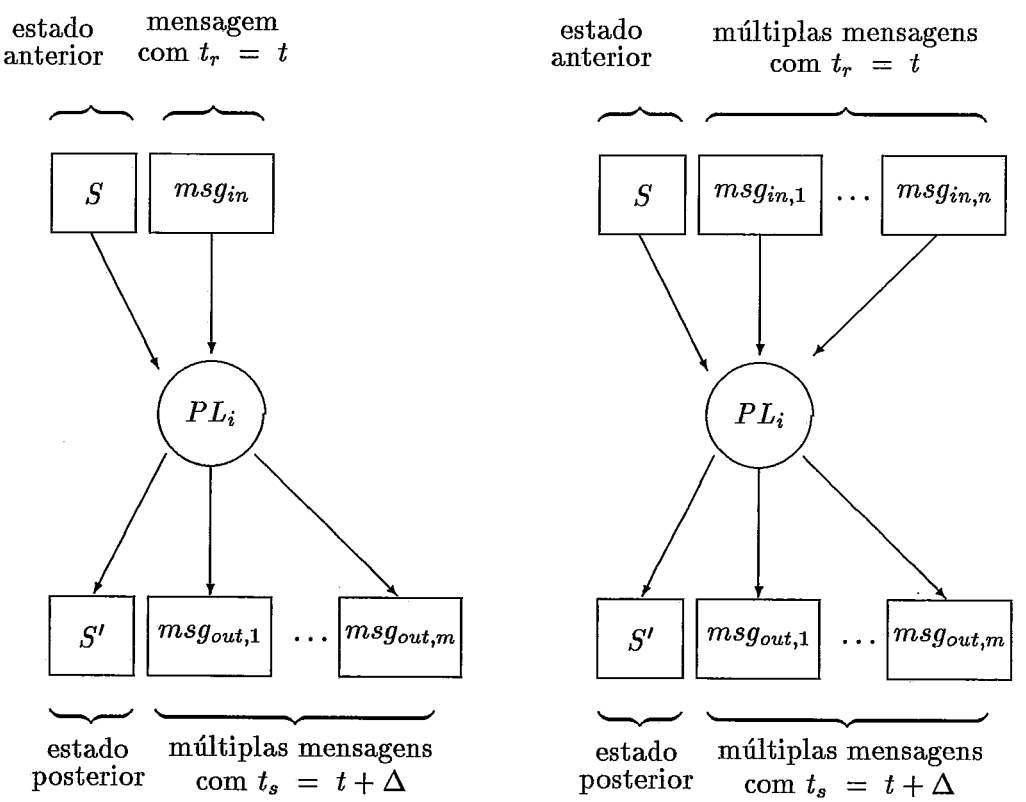

Figura 2.1: Processamento de eventos em um processo lógico

de mensagens transmitidas naquele *SF* terminando em um estado global equivalente ao estado final do *SF*.

Para atingir o objetivo de correção, um simulador qualquer, precisa garantir que os eventos sejam processados em uma ordem coerente, respeitando as relações de precedência estabelecidas. Com este fim são desenvolvidas técnicas de escalonamento de eventos, ou *paradigmas de simulação*, adequadas a cada tipo de arquitetura de computador.

Como veremos a seguir, as técnicas para simulação em máquinas seqüenciais e paralelas são bastante diversas. Para simulação em máquinas seqüenciais existe uma única técnica, bastante simples, que admite variações somente quanto a estruturas de dados empregadas. A simulação em máquinas paralelas não deriva diretamente do caso seqüencial e ainda oferece diferentes paradigmas cujo desempenho pode ser sensível à arquitetura e a aplicação simulada.

Chegando ao fim desta discussão, é importante relembrar as funções dos elementos principais desenvolvidos durante o estudo por simulação para evitar um tipo comum de equívoco. Um *SF* é um modelo bem estruturado que representa o sistema real, apenas uma descrição sistematizada do sistema em estudo para facilitar o desenvolvimento da etapa seguinte. O *SL*, construído sobre a estrutura delineada pelo *SF*, descreve estruturalmente o simulador que será executado em computador.

2.2.1 Simuladores Seqüenciais

Como dissemos, a máquina onde executa este sistema distribuído, que é o *SL*, determina a técnica ou o paradigma de simulação a ser empregado. Entre todos os paradigmas, o mais simples é, sem dúvida, o da simulação seqüencial.

No caso de execução do simulador em uma máquina seqüencial, todo o sistema distribuído que o compõe está contido em um único espaço de endereçamento e todos os seus processos são executados por um único processador. Esta arquitetura permite que uma única base de tempo seja usada para marcar o tempo simulado e que as comunicações entre *PLs* sejam feitas trivialmente, como cópias de dados de um endereço a outro dentro de um mesmo espaço de memória.

O trabalho principal realizado por um simulador, que é manter a ordem de precedência de eventos garantindo a correção da simulação, é bastante simples no ambiente de uma máquina seqüencial. Os eventos são escalonados um a um, a partir dos iniciais, seguindo uma ordem não decrescente de tempos de recepção. Para melhor entender como isto é feito passemos a descrição algorítmica de um simulador seqüencial.

Um simulador seqüencial constitui-se de um conjunto de *PLs* e de um algoritmo de controle que utiliza duas estruturas de dados principais:

Definição 2.10 Relógio. *Uma variável de tipo real que indica o tempo até o qual o sistema físico correspondente foi simulado, ou seja, todas os eventos $E(t,p)$ enviados no sistema físico com $t <$ Relógio, podem ser previstos no sistema lógico em algum ponto*

de sua execução. □

Definição 2.11 Lista de Eventos. *Uma coleção de tuplas da forma* $E = \{t_r, p_r, t_s, p_s, m\}$, *onde* $t_s \geq$ *Relógio e* m *representa uma mensagem. Uma tupla* E *nesta lista, representa um evento que será enviado ao processo destinatário* p_r *no tempo* t_s *caso o processo remetente* p_s *não receba nenhuma mensagem em um tempo* t *onde Relógio* $\leq t < t_s$. □

Nem todas as tuplas inseridas na lista de eventos são transmitidas no SL, pois podem não ocorrer no SF. Na verdade, toda entrada E na lista de eventos é definida como *evento condicional*, pois pode vir a ser cancelada caso p_s receba alguma mensagem em tempo anterior a t_s.

Um simulador seqüencial de uso genérico não tem, no entanto, a capacidade de detectar por si só a necessidade de cancelamentos em sua lista de eventos.Em geral, o simulador apenas oferece à aplicação um mecanismo para remoção de eventos na lista (que funciona de forma análoga ao mecanismo de inserção). A aplicação fica encarregada de manter na lista somente os eventos que ocorrem no SF, removendo os eventos incorretos ao detectá-los.

Exige-se que para todo $PL_i \in SL$, exista pelo menos uma entrada E na lista de eventos para a qual PL_i seja o remetente. Se um PL_i não enviar mais nenhum evento no futuro, a não ser que receba mais mensagens, ou seja considerado terminado, sua entrada na lista de eventos deverá ser $\{t_r, p_r, \infty, i, m\}$, onde t_r, p_r e m contêm valores arbitrários.

O algoritmo de controle do simulador seqüencial é simples e resume-se a escalonar repetidamente a ocorrência de eventos em cada PL do sistema, até que um critério de terminação seja atingido. A única dificuldade, que neste tipo de simulação não chega a ser séria, é a determinação da ordem correta de escalonamento de eventos. A ordem correta, segundo a regra de causalidade, pode ser atingida respeitando-se os procedimentos abaixo.

Seja $E = \{t_r, p_r, t_s, p_s, m\}$ uma entrada na lista de eventos tal que $t_r < t'_r$, para qualquer outra entrada $E' = \{t'_r, p'_r, t'_s, p'_s, m'\}$ presente na mesma estrutura. A esta entrada E atribui-se o nome de *menor tupla*. Quando o evento correspondente à E é escalonado em p_r, o processo destinatário, no tempo simulado t_r, pode-se ter certeza que não será inserida na lista de eventos, a partir deste tempo, nenhuma outra entrada $E^* = \{t^*_r, p^*_r, t^*_s, p^*_s, m^*\}$ tal que *Relógio* $\leq t^*_r < t_r$. Toda nova entrada gerada como conseqüência do processamento de uma entrada recebida em um tempo t_r será associada a um tempo de recepção maior ou igual a t_r.

Ao descobrir uma entrada da lista de eventos que corresponda à menor tupla e disparar seu processamento, o simulador transforma um evento condicional em *evento definitivo*. Este tipo de evento sempre ocorre no SF e não pode ser cancelado no futuro.

Existe sempre a possibilidade de várias entradas da lista de eventos possuirem um mesmo valor para t_r. No que concerne o tratamento de uma menor tupla *múltipla*, várias estratégias podem ser adotadas. As múltiplas entradas podem ser consideradas como partes da informação de um único evento que deve ser tratado como uma operação atômica, muitas vezes chamado de *evento múltiplo* ou *super-evento*. Alternativamente, podem ser

```
(* inicialização *)
Relógio := 0;
lista de eventos := (t_i, m_i);
(* iteração *)
while (critério de terminação não atingido) do begin
      remova a menor tupla (t,m) da lista de eventos;
      simule o efeito da transmissão de m em t;
      Relógio := t;
end;
```

Figura 2.2: Algoritmo de Controle de um Simulador Seqüencial

tratadas como eventos concorrentes que podem ser processados em uma ordem arbitrária ou ainda, consideradas como eventos distintos que devem ser processados em uma ordem determinada. Neste último caso é necessário definir uma forma de determinação da ordem de processamento a ser seguida, o que comumente é chamado de estratégia de *tie breaking*.

O algoritmo apresentado na Figura 2.2 ilustra uma realização simples de um simulador seqüencial que repete o procedimento de seleção de menor tupla e escalonamento do evento associado até que uma condição de terminação seja atingida. Neste simulador é extremamente simples computar um estado global do sistema: todas estruturas de dados estão centralizadas em um único processador e o estado global em um tempo t é diretamente representado pelo conteúdo da lista de eventos, do relógio e das variáveis de cada *PL*.

Este esquema de simulação seqüencial tem sido largamente utilizado desde tempos anteriores ao desenvolvimento de arquiteturas paralelas até os dias de hoje, sem muitas alterações em sua estrutura básica. A estrutura de dados que implementa a lista de eventos é sempre uma fila de prioridades, ou *priority queue*. Existem diversas estruturas de dados, mais ou menos eficientes para implementar esta fila e a seleção desta é crucial para o desempenho do simulador. O tempo gasto em operações de acesso a esta estrutura dominam a complexidade de tempo assintótica do simulador, determinando sua eficiência.

Os simuladores seqüenciais são usados, ainda hoje, para avaliação da aceleração, ou *speed-up*, de simuladores paralelos. Para que esta avaliação seja o mais realista possível, segundo a definição clássica de *speed-up* abaixo, deve se buscar um máximo de eficiência do simulador seqüencial usado como fator de comparação.

Definição 2.12 Speed-up

$$\textit{Speed-up} = \frac{\textit{Tempo de execução do melhor algoritmo seqüencial}}{\textit{Tempo de execução do algoritmo paralelo}}$$

□

Entre os simuladores seqüenciais mais eficientes de que se têm notícia, encontram-

se os que utilizam *splay-trees* na implementação de sua lista de eventos [8, 19]. Outras implementações eficientes empregam estruturas de dados como *hash-tables*, *AVL-trees* ou *red-black trees*. Em [31] encontramos um estudo empírico detalhado do desempenho de algumas destas estruturas como listas de prioridades.

2.2.2 Simuladores Paralelos

Ao longo do tempo, observou-se que grandes simulações de eventos discretos, em máquinas seqüenciais, consomem tempos exageradamente grandes. Este peso computacional aliado, ao paralelismo intrínseco dos modelos tipicamente simulados, torna esta aplicação um candidato ideal ao processamento paralelo [10].

No entanto, a paralelização de simuladores de eventos discretos não é uma tarefa simples. Em sistemas onde a estrutura da computação, ou seja, a ordem de execução de operações, é conhecida em tempos que antecedem sua execução, o procedimento de paralelização fica bastante simplificado.

No tipo de simulação que aqui discutimos, partindo de um pequeno conjunto de eventos, cujos tempos de escalonamento são atribuídos no início da execução, um conjunto de outros eventos vai sendo gerado. Os elementos deste conjunto devem ser escalonados em uma determinada ordem, que só é conhecida a medida que vão sugindo. Por isso, não é possível determinar, em tempo de compilação, uma distribuição de dados para execução paralela que garanta respeito às relações de precedência entre os eventos.

Muitos esforços têm sido direcionados no sentido do desenvolvimento de mecanismos de simulação paralela de eventos discretos. No desenvolvimento de novos paradigmas, surgiram diferentes algoritmos classificados segundo duas escolas principais, que determinam como se realiza o escalonamento distribuído de eventos.

Os algoritmos da escola *conservadora*, cuja criação é atribuída a Chandy, Misra e Bryant [30, 27, 29, 32], baseiam-se em uma técnica de escalonamento que impede a ocorrência de erros de causalidade. A execução de um evento, em um processo lógico, somente é iniciada quando se tem absoluta certeza de que todos os outros eventos de que depende no grafo de precedência da computação já ocorreram. Até que esta condição seja atendida, o processo lógico fica bloqueado, o que pode vir a causar *deadlock* no sistema.

Os algoritmos da escola *otimista*, por outro lado, possuem um conceito mais flexível de escalonamento, o que tende a permitir uma melhor exploração de paralelismo a nível de modelo e de arquitetura de máquina. Nestes algoritmos, um evento pode ser escalonado a qualquer momento acreditando-se, otimisticamente, que todos os outros de que depende já tenham ocorrido. Nem sempre esta suposição se confirma e ocorrem erros de causalidade. Um mecanismo qualquer deve, então, ser capaz de detectar estes erros e corrigí-los fazendo que a simulação retorne ao caminho correto. O primeiro algoritmo apresentado com este conceito foi o *Time Warp*, baseado no paradigma de *Tempo Virtual* desenvolvido por Jefferson [1].

Muitos experimentos foram realizados para analisar o desempenho dos vários algo-

ritmos pertencentes a cada escola. O desempenho dos algoritmos conservadores provou ser extremamente sensível às características da aplicação simulada, levando, em muitos casos, a resultados desapontadores. Devido à estratégia de escalonamento distribuído que empregam, estes algoritmos não permitem uma boa exploração do paralelismo nas aplicações e muitas vezes forçam uma seqüencialização desnecessária na execução de eventos. Além disso, técnicas de prevenção ou detecção e resolução de *deadlocks*, que costumam ser empregadas com estes algoritmos, requerem uma configuração estática do sistema prejudicando sua flexibilidade.

Na escola otimista, resultados promissores têm sido obtidos, em especial com o mecanismo *Time Warp*. Intuitivamente, poderia-se esperar que a estratégia de permitir a ocorrência de erros de causalidade para depois corrigí-los acarretaria uma degradação de desempenho superior a que seria imposta pelos mecanismos de tratamento de *deadlock* dos algoritmos conservadores. No entanto, resultados experimentais [9] e analíticos [33] indicam o contrário. A implementação do mecanismo na forma de sistema operacional, descrita em [2, 6, 1, 4], tem sido submetida às mais diversas aplicações sempre com bons resultados em termos de *speed-up*. Entre os mais importantes resultados encontram-se experimentos com simulações de combate [14, 13, 15], redes de comunicação [24], sistemas biológicos [20] e outros fenômenos físicos [21, 22].

Existe, hoje, uma enorme variedade de paradigmas de simulação paralela de eventos discretos. Fujimoto, em [10, 7] apresenta um apanhado geral dos mais relevantes trabalhos na área, discutindo cada um em suas principais características e provendo vastas referências aos originais. No âmbito deste estudo discutiremos exclusiva e extensivamente, o paradigma de *Tempo Virtual* implementado pelo mecanismo *Time Warp*.

Capítulo 3

Tempo Virtual e o Mecanismo Time Warp

O paradigma de *Tempo Virtual* tem sido objeto de estudo de muitos pesquisadores, desde sua apresentação em 1985, em instituições comerciais e acadêmicas espalhadas por todo o mundo. O paradigma, que provê uma abstração flexível do tempo real, pode ser usado na descrição e sincronização de sistemas distribuídos aplicados a vários problemas, mas principalmente à Simulação Paralela de Eventos Discretos.

O *Tempo Virtual* é implementado pelo mecanismo *Time Warp*, criado na Rand Corporation em 1981 com o intuito de acelerar simulações de grande porte dividindo-as em pequenos processos, ou objetos, executados concorrentemente. A primeira implementação do mecanismo foi escrita em um dialeto de Lisp para execução em uma rede de estações Xerox SIP-1100 (Dolphin) conectadas a um hospedeiro VAX. A esta primeira implementação seguiu-se uma nova, envolvendo o Jet Propulsion Laboratory, Caltech, UCLA e USC, escrita em C e projetada de modo a ser facilmente portável para outras arquiteturas.

Desde então, diversos experimentos foram realizados com este mecanismo, produzindo bons resultados em termos de eficiência e ainda indicando um grande potencial a ser explorado. A cada nova conferência de simulação, muitos novos trabalhos relacionados ao *Time Warp* são publicados e cada vez mais este se firma como o paradigma mais adequado a construção de um simulador paralelo de uso geral.

Este capítulo, apresenta os conceitos e terminologia mais básicos ao *Tempo Virtual* e à estrutura do *Time Warp*. Discutem-se aqui alguns aspectos de ordem teórica do paradigma e outros de ordem prática do mecanismo que servirão de base para o entendimento do trabalho descrito nos capítulos seguintes.

3.1 Tempo Virtual

Seja um sistema distribuído descrito por uma coleção de processos seqüenciais que se comunicam exclusivamente por trocas de mensagens. Digamos que cada processo pode se comunicar com qualquer outro do sistema sem que para isso seja preciso definir um *canal* que os interligue.

Cada processo deste sistema é dotado de um relógio local que marca um *tempo virtual*. Este tempo não tem necessariamente nenhuma relação com o tempo real e é descrito por uma variável de tipo real, podendo assumir valores em um intervalo $[-\infty, +\infty]$, totalmente ordenados pela relação $<$ *menor que*.

Toda mensagem que trafega no sistema é estampada a seguinte lista de valores:

$$L = \{t_r, id_r, t_s, id_s\}$$

onde t_r representa um tempo virtual de recepção, id_r o identificador do destinatário, t_s um tempo virtual de envio e id_s o identificador do remetente. O tempo t_s representa o tempo virtual do remetente no momento do envio e t_r o tempo virtual do destinatário no momento da recepção da mensagem. Estes valores são considerados parte dos dados da mensagem e podem ser lidos pelo recipiente.

Seja um plano $\mathcal{S}$, onde cada ponto P é identificado por um par ordenado (s, t), onde s é uma coordenada espacial e t uma coordenada temporal. Consideremos que as coordenadas temporais sejam tempos virtuais e as espaciais identificadores de processos. Consideremos ainda que cada ponto, no eixo das abcissas deste plano, representa um processo do sistema.

Cada atividade executada em um processo determina uma mudança em seu estado, ou seja, um evento, e pode ser associada ao tempo virtual de sua ocorrência, definindo um ponto no plano $\mathcal{S}$. Desta forma, cada mensagem transmitida estará associada a dois pontos no plano: um evento que caracteriza seu envio e outro que caracteriza sua recepção. Uma mensagem qualquer no sistema representa, não somente uma transferência de informação entre dois pontos do espaço–tempo, mas também uma relação de causalidade entre eventos.

Um evento em $\mathcal{S}$, no ponto (s, t), determina uma computação que ocorre inteiramente no processo s e envolve zero ou mais das seguintes operações onde s pode:

- receber um número arbitrário de mensagens com $t_r = t$ e $id_r = s$ e ler seu conteúdo;
- consultar seu relógio virtual;
- atualizar suas variáveis de estado;
- enviar um número arbitrário de mensagens, que serão rotuladas com $t_s = t$ e $id_s = s$.

Aproveitando a relação de causalidade é possível definir um sistema de relógios lógicos, fracamente acoplado, sobre este sistema distribuído, que envolva os relógios virtuais de todos os processos e possa ser usado para criar uma ordenação parcial de eventos. Dessa forma, fazendo que o sistema obedeça a duas regras fundamentais baseadas nas Condições de Relógio de Lamport [34], define-se um *Sistema de Tempo Virtual*.

Regra 1 *Para uma mensagem qualquer enviada no sistema, o tempo virtual de envio* t_s *é sempre menor que o tempo virtual de recepção* t_r.

Regra 2 *O tempo virtual associado a um evento em um determinado processo é sempre menor que o tempo virtual do evento seguinte que ocorre no mesmo processo.*

Estas regras implicam que as mensagens enviadas por um processo respeitam uma ordenação crescente por tempos virtuais de envio. Analogamente, todas as mensagens recebidas e tratadas por um processo respeitam sempre uma classificação por ordem crescente de tempos de recepção.

Podemos verificar que, com estas regras, o sistema de relógios passa a ter uma relação $CAUSA$ que serve para a construção de uma ordem parcial de eventos. Esta relação é idêntica a relação $ANTES$ definida em [35].

Dizemos que um evento e $CAUSA$ e' se e somente se uma existe uma seqüência de eventos $< e, e_1, e_2, \ldots, e_n, e' >$ tais que:

$$e \ CAUSA \ e_1$$

$$e_1 \ CAUSA \ e_2$$

$$\vdots$$

$$e_n \ CAUSA \ e'$$

onde para cada par de eventos adjacente uma de duas condições é satisfeita:

1. e_k e e_{k+1} ocorrem no mesmo processo p_i, nos tempos virtuais t_i^1 e t_i^2, respectivamente, sendo que $t_i^1 < t_i^2$ e nenhum outro evento ocorre em um tempo virtual t_i' tal que $t_i^1 < t_i' < t_i^2$.
2. e_k é o envio de uma mensagem m de um processo p_i recebida por um processo p_j em e_{k+1}.

Entendendo esta relação, podemos enunciar uma condição que garante a que a execução de uma seqüência de eventos em um sistema de tempo virtual esteja correta:

> Se um *evento A CAUSA um evento B*, então a execução de A e B deve ser escalonada de modo que A seja completado antes de B iniciar.

O mecanismo de que implementa o *Tempo Virtual* deve garantir que a cadeia de causalidade entre os eventos seja sempre respeitada. Se, por exemplo, $t_r(A) < t_r(B)$ e ambos ocorrem no mesmo processador então B só pode ser escalonado após o término de A. Se $t_r(A) < t_r(B)$, A e B ocorrem em processadores distintos p_A e p_B, respectivamente, e o processamento de A resulta no envio de uma mensagem m para p_B com $t_r(A) \leq t_r(B)$, então novamente B só pode ser escalonado após o término de A. Mas se, por outro lado, não existe ligação causal entre os dois eventos, não há restrições à ordem de seu escalonamento.

Esta é a teoria por trás de sistemas de *Tempo Virtual*. A maneira como é implementado pouco, ou nada, deve importar ao usuário de um sistema que incorpora estes conceitos. Para o programador de aplicações apenas é relevante saber que existe um mecanismo garantindo a coerência do sistema de relógios e permite-lhe ordenar parcialmente os eventos que ocorrem em seu programa.

3.2 O mecanismo Time Warp

O mecanismo *Time Warp* implementa o paradigma de *Tempo Virtual* definindo um ambiente para a execução de simulações de eventos discretos ou outras aplicações distribuídas que requeiram uma base base de tempo coerente para a ordenação de atividades. Este ambiente apresenta um conjunto de características interessantes:

distribuição: o mecanismo executa sobre um conjunto de processadores conectados por uma rede de comunicação confiável, que não precisa preservar a ordem das comunicações;

simetria: em cada processador da rede existe uma réplica do mecanismo absolutamente idêntica às alocadas em outros processadores;

transparência: o programa que executa sobre o mecanismo não precisa reconhecer a sua existência nem fazer declarações adicionais ou prover-lhe informações específicas.

Em relação ao desenvolvimento de simulações para execução seqüencial, não existe dificuldade adicional na programação de uma simulação a ser executada sobre o *Time Warp*. O programa deve ser construído como uma coleção de processos seqüenciais e não é necessário que defina ou declare alguma topologia de rede de comunicação. Um processo pode se comunicar com qualquer outro processo do sistema, bastando para isso que conheça a identificação do destinatário.

Os processos executam de forma assíncrona, interagindo através de trocas de mensagens. Toda mensagem que trafega no sistema contém, além de um texto (conjunto de dados de tipos quaisquer), os valores de identificação apresentados na Figura 3.1.

Cada processo do sistema, ou processo lógico, possui uma variável local *lvt*, de tipo **real**, que marca seu tempo virtual. Para cada mensagem enviada para outro processo, o valor desta variável é atribuído automaticamente ao campo *svt*. Da mesma forma, o campo *sid* recebe a identificação do processo remetente. O campo *rvt* é preenchido pela aplicação

svt:	tempo virtual de envio	**(real)**
sid:	identificação do remetente	**(integer)**
rvt:	tempo virtual de recepção	**(real)**
rid:	identificação do destinatário	**(integer)**
signal:	sinal da mensagem	(+ ou −)
text:	texto ou dados da mensagem	**(array of byte)**

Figura 3.1: Estrutura de dados de uma mensagem no *Time Warp*

de acordo com sua estratégia própria de ordenação de eventos e indica o tempo marcado pelo relógio local do destinatário, ou tempo virtual, em que a mensagem será recebida. O campo *rid*, preenchido também pela processo remetente, contém a identificação do destinatário e é usado pelo mecanismo no roteamento da mensagem.

Cada processo lógico executa um ciclo principal, em que, a cada iteração tem seu relógio virtual avançado para o *rvt* da próxima mensagem a processar, recebe a mensagem, processa seu conteúdo e envia novas mensagens, sem se bloquear até que tenha processado todas as mensagens que presentes em sua *fila de entrada*. As mensagens nesta fila são processadas em ordem não descrescente de *rvt*, o que garante que, localmente, o sistema de relógios está correto. Desta forma, o valor de *lvt* de um processo indica seu progresso local, ou seja, todas as mensagens já recebidas com $rvt < lvt$ já foram processadas. No entanto, nada garante que todas mensagens endereçadas ao processo durante a simulação com $rvt < lvt$ já tenham sido recebidas. Como os processos executam assincronamente, alguns deverão estar mais a frente e outros mais atrás no tempo virtual e por isso, não necessariamente, um processo recebe mensagens em ordem não decrescente de tempos virtuais de recepção.

Com isso, erros de causalidade podem vir a acontecer: um processo A mais atrasado, com $lvt = t - \delta$, ao enviar uma mensagem para outro processo B mais adiantado, com $lvt = t$, estaria enviando uma mensagem para um tempo que já pertence ao passado de B. Assumindo que a mensagem retardatária, se fosse processada no tempo correto, teria provocado alguma modificação no estado local do processo B, o fato do processo ter prosseguido no tempo virtual, não tendo processado a tal mensagem, significa que o processo seguiu por um caminho computacional incorreto. Todos os eventos processados em B associados a tempos maiores que o da mensagem retardatária teriam usado como entrada um estado possivelmente diferente do que resultaria de seu processamento.

Nos moldes do sistema que estamos estudando, verifica-se que um processo qualquer não tem como saber que vai receber uma mensagem em um determinado tempo virtual e por isso não tem como se bloquear até que esta mensagem a ele chegue. Mas como então

garantir a ordem correta de processamento de mensagens em cada processo do sistema ?

Os processos que executam sobre o mecanismo *Time Warp* assumem, otimisticamente, que não irão receber mensagens fora de uma ordem correta de processamento. Caso um processo venha a receber uma mensagem com tempo de recepção em seu passado, o mecanismo se encarrega de fazer o processo retroceder no tempo para que receba a mensagem no instante correto, desfazendo o erro de causalidade. A correção toda se faz sem que o processo lógico precise tomar qualquer atitude e, após a mensagem retardatária ser processada, seu tempo local volta a avançar normalmente.

Este processo de retrocesso temporal é chamado de *rollback* e nem sempre se caracteriza por um efeito localizado no processo lógico onde ocorre o erro de causalidade. Ao longo do tempo o processo pode ter enviado mensagens a outros processos causando modificações em seus estados locais. Por isso, fazer o processo retornar ao seu passado no tempo virtual significa não só restaurar um estado de suas variáveis locais, mas também cancelar os efeitos colaterais que produziu sobre outros processos em tempos superiores ao do conflito causal. Em outras palavras, o processo teria que "desenviar" mensagens.

Para que a restauração de um estado no passado de um processo possa ser feita dois itens de informação são necessários:

1. o estado local processo em um instante do tempo virtual anterior ao do conflito causal;
2. uma lista das mensagens enviadas pelo processo em tempos virtuais posteriores ao do conflito causal;

Com o estado local restaura-se o conteúdo que variáveis do processo possuiam em um determinado instante e usando a lista de mensagens pode-se cancelar os efeitos colaterais produzidos sobre outros processos. Para implementar o mecanismo de *rollback*, verifica-se então a necessidade de coletar, segundo alguma estratégia, informações durante a execução do programa. Estas informações ficariam armazenadas em memória e seriam utilizadas sempre que se detectasse a ocorrência de um erro de causalidade.

O armazenamento periódico de estados ao longo de uma computação, chamado de *checkpointing*, consome continuamente espaços de memória até que se esgotem por completo. Para que a computação possa progredir, além desse ponto, é necessário definir uma estratégia de reutilização dos espaços já ocupados, que não serão referenciados no futuro pelo mecanismo de *rollback*.

Esta estratégia, similar ao *garbage collection* de outros sistemas, no *Time Warp* é chamada de *coleta de fósseis* ou *fossil collection*. Chamam-se fósseis, todos os itens de memória associados a tempos virtuais tão antigos que não mais serão necessários no curso da simulação. Determina-se que um item de memória é ou não um fóssil através de um valor chamado *Tempo Virtual Global* ou *Global Virtual Time (GVT)*.

O *GVT* é uma medida estimada do progresso da simulação distribuída, calculada em função dos relógios de todos os processos do sistema e dos tempos virtuais associados às mensagens em trânsito. Esta medida funciona como uma base de tempo global do

sistema, que, diferentemente do relógios virtuais dos processos que avançam e retrocedem, sempre progride.

Com esta descrição superficial do *Time Warp*, podemos identificar estruturas principais que serão detalhadas a seguir à medida que se apresenta a arquitetura do mecanismo. Uma parte do mecanismo é responsável pelo controle local dos processos garantindo que as mensagens são sempre processadas na ordem correta, encarregando-se dos *rollbacks* e do cancelamento de efeitos colaterais. Uma outra parte assume o controle de atividades globais do sistema, como gerência de memória, controle de fluxo e detecção de terminação, atividades que dependem do cálculo do GVT.

3.2.1 O Mecanismo de Rollback

Pode-se dizer com segurança que o coração do *Time Warp* está no mecanismo local de cada processador, que implementa as operações envolvidas em *rollback*. Na implementação original de Jefferson [1, 6, 2] utiliza-se o conjunto de estruturas de dados apresentado na Figura 3.2 para controle de cada processo lógico.

Cada processo possui três filas:

1. uma *fila de entrada de mensagens* onde são guardadas todas as mensagens ao enviadas ao processo lógico ordenadas de forma não decrescente por tempo virtual de recepção;
2. uma *fila de saída de mensagens* onde são guardadas cópias de todas as mensagens enviadas pelo processo lógico ordenadas de forma não descrescente por tempo virtual de envio;
3. uma *fila de estados locais* que armazena retratos de estados passados das variáveis do processo associados ao tempo virtual do instante de sua gravação ordenados de forma não decrescente por este valor.

A fila de saída de mensagens armazena mensagens com uma característica especial que define particularidades não usuais no sistema de filas do *Time Warp*. As cópias ali guardadas são na verdade *anti-cópias* das mensagens enviadas pelos processos, ou *anti-mensagens*. Uma anti-mensagem é uma cópia idêntica da original em todos os campos de cabeçalho e texto, exceto pelo campo de *signal* que será sempre o oposto do valor primeiramente atribuído. Estas mensagens especiais são usadas pelo mecanismo de *rollback* na criação de um instrumento de cancelamento dos efeitos colaterais causados por um processo, ou como dito anteriormente, para "desenviar" mensagens.

A atribuição do campo de sinal da mensagem pode seguir critérios diferentes. Nas primeiras implementações do *Time Warp*, este campo era considerado como controle do mecanismo e, logo, não acessível ao usuário. Mensagens originais recebiam sempre o sinal + e anti-mensagens usadas pelo mecanismo de *rollback* recebiam sempre o sinal −. Nas implementações mais modernas este critério foi alterado de forma a permitir que as aplicações decidam que valor atribuir ao sinal da mensagem. Com isso, torna-se possível

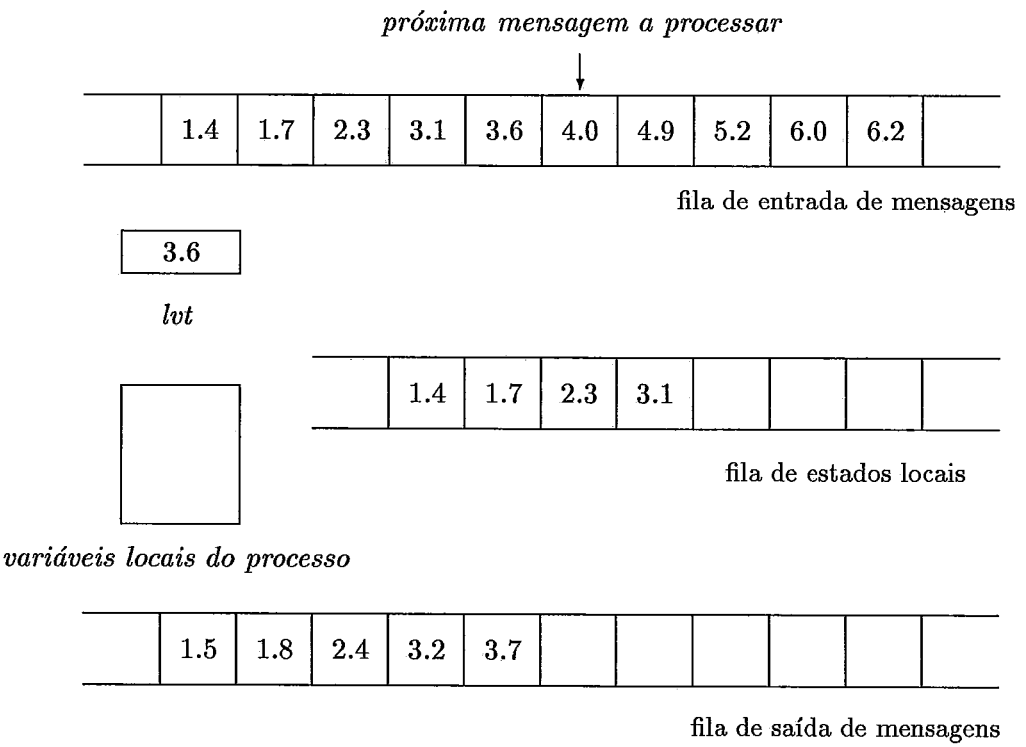

Figura 3.2: Estruturas de dados de um processo lógico no *Time Warp*

que as aplicações usem o mesmo princípio de aniquilação de mensagens empregado no mecanismo de *rollback*. Assim, uma aplicação poderia decidir "desenviar" uma mensagem da mesma forma que o faz internamente o *Time Warp*. Por outro lado, a adição de uma primitiva de cancelamento à programação de aplicações introduz algumas modificações no protocolo de aniquilação, como será visto apropriadamente.

Um *rollback* compreende duas etapas principais: a restauração de um estado local, levando o processo de um tempo virtual t para um tempo virtual $t - \delta$ passado, e o cancelamento das mensagens enviadas pelo processo no intervalo $[t - \delta, t]$. Neste cancelamento são usadas as anti-mensagens guardadas nas filas de saída, enviando-se todas as anti-mensagens produzidas de $t - \delta$ até t para os mesmos destinatários das versões originais. Ao chegarem aos seus destinos esta cópias são inseridas nas filas de entradas dos processos, aniquilando as mensagens originais. Após esta inserção, diminui a quantidade de itens na fila e o efeito final é do desaparecimento das mensagens como se jamais tivessem existido.

Existem algumas situações possíveis no processo de aniquilação entre mensagem e anti-mensagem. A anti-mensagem pode chegar o processo destinatário antes da original ter sido processada, caso em que as duas desaparecem e o processo não percebe a existência de ambas. A anti-mensagem pode chegar ao seu destino após o processo ter tratado a mensagem original; neste caso o processo sofre *rollback*, as mensagens se aniquilam e o processo volta a avançar no tempo processando as mensagens que restaram em sua fila de entrada. A última situação possível ocorre apenas em arquiteturas onde mensagens podem ser roteadas através de rotas diferentes, algumas mais rápidas e outras mais lentas, ou que não preserve a ordem de envio das mensagens. Poderia então acontecer de uma anti-mensagem ser entregue a um processo antes da cópia original e ser inserida na fila de entrada, mas apenas temporariamente, até que a original chegasse causando a aniquilação das duas.

A fila de estados locais do processo armazena cópias dos conteúdos das variáveis de estado do processo. Estes estados podem ser gerados a cada processamento de evento ou segundo uma freqüência menor. Quanto mais freqüente for o registro de estados locais, menor pode ser a distância temporal percorrida no caso de um *rollback*, mas, por outro lado, maior será o consumo de memória.

Vejamos uma situação prática: um processo cujo relógio virtual local está em 170.0 é levado a retroceder ao tempo 123.0 pela recepção de uma mensagem retardatária, no entanto, o último estado registrado antes do tempo 123.0 possui tempo de gravação de 100.0. Entre os tempos 100.0 e 123.0 vários eventos foram processados, mas devido a política de redução de freqüência de gravação de estados o estado seguinte na fila corresponde ao tempo 132.5. A única opção é levar o processo de volta ao tempo 100.0 e daí recomeçar a computação. Se houvessem outros estados na fila entre 100.0 e 123.0 seria possível efetuar *rollback* para um tempo mais próximo.

De acordo com os experimentos de Fujimoto [9], os conflitos causais que levam os processos a retrocederem no tempo resultam em *rollbacks* curtos. Em média, estes retrocessos fariam um processo voltar atrás apenas um ou dois eventos e, por isso, salvar estados com maior freqüência tende a ser mais eficiente. Por outro lado, seus resultados apresentam uma alta taxa de *rollbacks* por segundo em diversas aplicações de teste, o que é suficiente

para indicar que o custo de cada restauração de estado deve ser pequeno para não comprometer o desempenho do sistema. Para reduzir os *overheads* de salvamento/restauração de estados, Fujimoto propõe o uso de *hardware* específico descrito em [11].

Segundo argumento de Jefferson [1] e a larga experiência que se tem hoje com *Time Warp* o custo envolvido com *rollbacks* é bem menor do que se imagina. Não existe possibilidade de ocorrência de *rollbacks* em cascata levando o sistema cada vez mais atrás no tempo virtual: no pior caso todos os processos do sistema retornariam ao mesmo tempo virtual do primeiro conflito causal e em seguida voltariam a progredir. Argumenta-se que analogamente ao argumento de *localidade espacial* que viabiliza os sistemas de memória virtual, existe para os sistemas de tempo virtual o argumento de *localidade temporal*. Os processos, nos programas que executam sobre estes sistemas, enviam mensagens que, em uma grande maioria, chegam no *futuro* de seus destinatários e portanto não causam *rollback*. E além disso, argumenta-se que as mensagens que chegam no passado de seus receptores forçando-os a voltarem no tempo para recebê-las causam *rollbacks* para um passado recente.

Durante muito tempo, somente através de verificações empíricas foi possível comprovar a validade destes argumentos. Somente nos últimos anos surgiram modelos analíticos capazes de caracterizar a dinâmica de um programa sobre *Time Warp*. Gupta [36] apresenta um modelo, baseado em cadeias de Markov, restrito a análise de casos considerando processadores homogêneos e incrementos de tempos virtuais de recepção selecionados em uma distribuição exponencial. Os resultados analíticos obtidos com o modelo são comparados a resultados empíricos obtidos com a execução de um modelo de carga sintética desenvolvido para avaliação de desempenho. Felderman e Kleinrock [37] propõem um modelo para o caso de dois processadores executando *Time Warp* indicando condições para maximização de desempenho de sistemas sincronizados por *rollback*.

Como vimos, durante a execução de um programa que executa sobre um mecanismo baseado em *rollback*, uma quantidade muito grande de informações é coletada e armazenada nas filas do sistema. Estas filas tenderiam a crescer indefinidamente, até esgotar o espaço de memória disponível, impossibilitando o prosseguimento da computação. No entanto, observando-se uma característica da computação no *Time Warp* foi possível definir uma estratégia básica de gerenciamento que permite um contínuo reaproveitamento de memória.

> Os estados e mensagens armazenados nas filas do *Time Warp* não são essenciais ao mecanismo de *rollback* durante toda a extensão de tempo em que se realiza uma computação.

Para provar esta afirmativa seria necessário provar primeiro que apesar de alguns processos sofrerem repetidos avanços e retrocessos no tempo virtual, o sistema simulado, como um todo, sempre progride. Uma vez determinada uma grandeza que pudesse medir este progresso global, seria simples verificar que os itens nas filas do sistema pertencentes a um passado virtual muito distante não seriam mais necessários ao mecanismo de *rollback*. Uma grandeza como esta, serviria como uma "base de tempo global" para o sistema assíncrono e permitiria, entre outras coisas, a detecção de terminação da computação.

Jefferson, em [1], define esta grandeza e mostra como sua importância se estende a diversas questões relacionadas ao funcionamento do *Time Warp*. As provas completas para as questões acima propostas são apresentadas nesta referência.

Definição 3.1 Tempo Virtual Global *ou* Global Virtual Time (GVT). *Em um estado global qualquer do sistema distribuído, o* GVT *é definido como o valor mínimo entre (1) o mínimo dos relógios virtuais locais de todos os processos do sistema e (2) o mínimo tempo virtual de recepção (rvt) das mensagens em trânsito naquele estado global.* □

Na definição original de Jefferson, o *GVT* é apresentado como uma propriedade de um retrato global do sistema obtido instantaneamente em um tempo real r. No entanto, como observado por Chandy e Lamport [28], em um sistema desprovido de um relógio físico global é impossível obter tal retrato. Conclui-se, assim, que a definição original é puramente matemática e de pouca valia na construção de um algoritmo para a obtenção de um valor numérico para esta grandeza. Por pretender ser semânticamente mais correta e computacionalmente prática, a definição acima considera que o *GVT* seja calculado sobre um conjunto de dados que pode ser realmente obtido em uma computação distribuída. Por isso, consideramos que um valor para o *GVT* pode ser somente estimado em função de um estado global do sistema, que pode ser computacionalmente obtido, mas jamais instantâneamente no tempo real.

Partindo da definição de *GVT*, pode-se inferir que nenhum processo do sistema pode regredir para um tempo virtual inferior a este valor. Se não existe nenhum processo no sistema com tempo virtual local inferior a um valor de *GVT*, então nenhum processo poderá causar *rollback* para um tempo virtual inferior a este valor. em outro através do envio de mensagens. Quaisquer mensagens que um processo venha a enviar no futuro virtual, possuirão tempos de envio maiores ou iguais ao valor de seu relógio local e tempos de recepção ainda maiores que os de envio, de acordo com a regra de causalidade. Estes tempos de recepção serão, portanto, superiores ao *GVT* e, conseqüentemente, estas mensagens não poderiam provocar *rollback* para um tempo anterior.

Uma outra potencial causa de *rollback*, mensagens em trânsito, poderiam causar retrocesso para tempos inferiores ao *GVT*, caso não tivessem sido consideradas em sua definição. Uma mensagem em trânsito é uma mensagem que ainda não foi entregue ao processo a que se destina e cujo efeito do tempo virtual de recepção sobre o relógio local deste processo ainda não foi contabilizado. Como o *GVT* é definido como um valor mínimo em um conjunto de tempos que inclui os tempos de recepção destas mensagens, pode-se concluir que é, na verdade, um *lower bound* dos tempos virtuais de um sistema.

Como nenhum processo pode regredir para um tempo inferior ao *GVT*, todos os itens de memória (estados e mensagens) associados a tempos virtuais menores que este valor não serão mais necessários ao mecanismo de *rollback*. Qualquer conflito causal em um tempo real após a estimativa de um valor de *GVT*, exigirá *rollback* para um tempo virtual necessariamente maior ou igual a este valor. Mensagens e estados com tempos virtuais menores que o *GVT* são, portanto, itens de memória que não serão mais referenciados pelo mecanismo de *rollback* durante a computação. Por terem se tornado virtualmente velhos

demais para o *Time Warp*, estes itens recebem o nome de *fósseis* e podem ser descartados para que a memória seja reaproveitada no armazenamento de outras mensagens e estados.

Além de ser usado para solucionar o problema de gerência de memória no *Time Warp*, o *GVT* é importante em outros aspectos da simulação. Como será discutido adiante, o *GVT* é usado também para controle de fluxo de mensagens entre processadores, detecção de terminação e para avaliar o progresso da simulação em relação ao tempo real em estudos de avaliação de desempenho.

3.2.2 Cálculo de *GVT*

Partindo da definição formal de *GVT*, diversos algoritmos foram propostos para estimar este valor. Para arquiteturas de memória compartilhada, seu cálculo reduz-se a um problema trivial, pois o estado global do sistema está sempre, instantâneamente, disponível.

Em sistemas de memória totalmente distribuída, o cálculo de *GVT* requer um algoritmo não trivial. Como apontado por Lin e Lazowska [38], existem duas dificuldades básicas na computação do *GVT*:

- as mensagens em trânsito no sistema não podem ser diretamente acessadas, o que é chamado de *problema de mensagens em trânsito*;
- os atrasos de propagação de mensagens entre pares de processadores podem ser diferentes, mesmo assumindo que os atrasos para um par específico de processadores seja constante. Este problema é chamado de *problema de notificação simultânea.*

No projeto de um algoritmo para cálculo de *GVT*, além de ser imperativa a resolução dos dois problemas acima por questões de correção, outros aspectos importantes devem ser considerados. É desejável que um algoritmo de *GVT* possa executar, corretamente, sobre arquiteturas distribuídas cujo meio de comunicação preserve ou não a ordem das mensagens enviadas. Além disso, a questão de desempenho é muito importante. Alguns algoritmos exigem que a computação seja interrompida durante o cálculo de *GVT*, outros requerem o uso de mensagens de controle que por vezes degradam a execução do sistema.

Na literatura encontram-se diversos algoritmos para cálculo de *GVT*. Alguns são mais genéricos, como o de Lin e Lazowska [38], e podem ser facilmente adaptados para diferentes arquiteturas. Alguns, como o de Samadi [40], resolvem o problema de mensagens em trânsito através do uso de mensagens de confirmação de recepção, ou *acknowledgement*, aumentando a complexidade de mensagens do algoritmo. Outros, como o apresentado por Bellenot [41], propõem soluções específicas para implementações de *Time Warp* sobre determinadas arquiteturas.

3.2.3 Controle de Fluxo e Gerência de Memória

O esgotamento de memória, no *Time Warp*, criado pelo aumento constante de informações usadas pelo mecanismo de *rollback*, é apenas uma parte do problema de gerenciamento

memória neste sistema. Devido as diferenças de velocidade de execução dos processos em diferentes processadores, um fluxo desequilibrado de mensagens pode, também, vir a causar esgotamento das áreas de armazenamento disponíveis.

Vejamos um exemplo prático desta situação. Seja um sistema de dois processadores executando uma simulação *Time Warp* que envolve três processos lógicos. Consideremos que:

- existe um sistema de comunicação, subjacente ao *Time Warp*, que contém um conjunto de *buffers* para armazenamento temporário das mensagens recebidas pelo processador até serem entregues ao mecanismo de simulação;
- uma vez entregues ao *Time Warp* as mensagens são inseridas nas filas de entrada dos processos a que se destinam;
- o processo *A1* executa no processador 0 e leva um tempo α para processar uma mensagem;
- o processo *A2* executa no processador 0 e leva um tempo α para processar uma mensagem;
- o processo *B* executa no processador 1 e leva um tempo β para processar uma mensagem;
- os processos se comunicam trocando mensagens segundo o grafo mostrado na Figura 3.3.
- $\beta = n\alpha$, onde $n >> 1$.
- para cada par de mensagens trocadas entre *A1* e *A2*, uma mensagem é enviada de *A1* para *B*.

Na situação exemplificada pode-se verificar que, após certo tempo de execução, os espaços disponíveis ao *Time Warp* esgotam-se. O processo *A1* envia mensagens em uma taxa maior que a capacidade de processamento de *B*. As mensagens que chegam no futuro virtual de um processo vão sendo inseridas em sua fila de entrada, até que o tempo virtual local atinja seu tempo de recepção e o evento seja escalonado. Nesse meio tempo, a fila de entrada cresce até que não existir mais memória para armazenar um novo elemento da fila.

A partir do instante em que a memória do *Time Warp* se esgota, este deixa de receber mensagens do sistema de comunicação. As mensagens que chegam, então, ao processador vão sendo inseridas em *buffers* do sistema de comunicação. Caso o mecanismo de simulação, eventualmente, execute uma coleta de fósseis recuperando espaço para recepção de novas mensagens, os *buffers* são descarregados e o sistema como um todo continua a evoluir. Caso contrário, os *buffers* acabam por ficar todos ocupados e o processador passa a não aceitar mais mensagens, o que potencialmente acarreta *deadlock* no sistema.

Para evitar que desbalanceamentos de taxas de processamento provoquem este tipo de situação, torna-se necessário definir um mecanismo de controle de fluxo de comunicação.

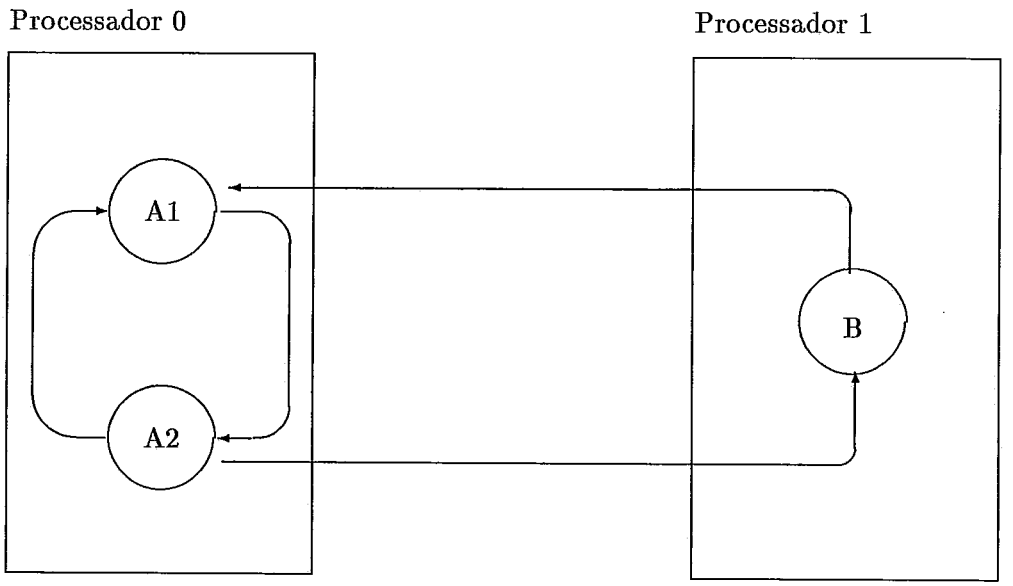

Figura 3.3: Exemplo de uma simulação com problemas de fluxo de mensagens

No contexto do *Time Warp*, este mecanismo não deve considerar controle de fluxo e gerenciamento de memória local separadamente, mas como aspectos de um único problema que é a utilização racional e eficiente do espaço de armazenamento de dados. A solução para este problema deve, obviamente, harmonizar-se com as características naturais do mecanismo otimista de simulação.

Como apontado por Jefferson [5], as técnicas mais comuns de controle de fluxo não se adequam às características do *Time Warp*. Protocolos que buscam sincronizar remetente e receptor através de bloqueio opõem-se à premissa básica dos mecanismos otimistas. A técnica otimista de simulação consegue extrair maiores *speed-ups* justamente por permitir que cada processo evolua segundo seu próprio ritmo. Gafni [42] afirma que este assincronismo leva a melhores desempenhos e, portanto, eliminar diferenças entre relógios virtuais locais não é uma solução para desbalanceamentos de fluxo.

Message Sendback, o primeiro protocolo adequado a solução deste problema no âmbito do *Time Warp*, deve-se a Gafni. O algoritmo baseia-se na construção de um conjunto de elementos das filas de estados e mensagens passíveis de descarte sem prejuízo para a simulação. O descarte de alguns dos elementos deste conjunto resultaria na liberação de um espaço de memória que poderia ser utilizado pelo *Time Warp* para receber mensagens do sistema de comunicação. Este protocolo, no entanto, trata da gerência de memória no escopo de processo lógico, não sendo capaz de enxergar o problema completo no escopo do processador.

Um protocolo pouco mais complexo, apresentado por Jefferson em [5], é similar ao de Gafni, mas oferece uma solução mais completa. O protocolo *Cancelback* propõe uma solução ótima ao problema de controle de fluxo e gerência de memória. Além disso, permite que, ao contrário do que a intuição indica, o mecanismo *Time Warp* execute uma simulação distribuída no mesmo espaço de memória ocupado pela equivalente seqüencial. Obviamente, neste caso de extrema restrição de memória, o desempenho da simulação distribuída estaria seriamente comprometido. A apresentação do protocolo completo e de sua adaptação para máquinas de memória distribuída são deixadas para mais adiante.

Capítulo 4

A linguagem occam e os transputers

Segundo Burns [43], o termo genérico *transputer* descreve uma família de dispositivos programáveis VLSI incluindo, entre outros, controladores de disco, processadores gráficos e processadores 16 e 32–bit de uso geral. O desenvolvimento de processadores desta categoria foi claramente direcionado à implementação em *hardware* de um modelo de programação concorrente.

O modelo de programação definido pela linguagem **occam**, fortemente influenciado pelo CSP introduzido por Hoare [44], exerceu papel chave no projeto dos *transputers*. Pode-se mesmo dizer que o *transputer* foi concebido como uma máquina **occam** de modo que suas características principais resultaram da necessidade de implementação eficiente do modelo implícito na linguagem.

Enquanto relativamente confortável e bastante adequada a usuários em alto nível, a interface de programação **occam** dos sistemas baseados em *transputers* oferece algumas dificuldades a programação a nível de sistema operacional. O programador não dispõe de recursos de alocação dinâmica de memória, não pode manipular endereços diretamente e ainda deve se restringir ao escalonamento de processos microprogamado no processador.

O mecanismo *Time Warp* exige a utilização de alguns recursos e técnicas não convencionais na programação **occam** de sistemas de *transputers*. É preciso implementar um algoritmo preemptivo de escalonamento baseado em tempo virtual, o que entra em desacordo com o escalonador em *hardware*. O registro de estados locais exige que o TW conheça, de alguma forma, o espaço de endereçamento usado para variáveis de estado dos processos lógicos.

Neste capítulo, apresentamos as características principais do *transputer* T800, usado

no hipercubo NCP I, máquina alvo desta implementação de Time Warp, e da linguagem de programação **occam**. O capítulo inicia com uma descrição resumida do modelo de programação **occam**, prossegue detalhando a implementação deste modelo no *transputer* e sua arquitetura interna e finaliza com a introdução de um mecanismo genérico de preempção de processos empregado na implementação do *Time Warp*.

4.1 A linguagem de programação occam

As linguagens de programação, em geral, são construídas como um reflexo da arquitetura da máquina que executa seus programas. Durante muito tempo a máquina de Von Neumann predominou como arquitetura e por isso, os produtos dos compiladores eram seqüências de instruções a serem executadas uma de cada vez.

O surgimento das arquiteturas multiprocessadas trouxe a necessidade de desenvolvimento de linguagens capazes de expressar o paralelismo oferecido pelas máquinas. A concorrência natural de vários problemas poderia explorar eficientemente o potencial das máquinas que permitem execução paralela de intruções.

No entanto, as linguagens de programação convencionais não permitem expressão alguma de concorrência, pois não foram concebidas com esse intuito. Algumas dessas linguagens foram adaptadas para programação concorrente, mas deixando sempre ao programador a responsabilidade de sincronização entre processos.

A linguagem **occam** foi, historicamente, a primeira a ser concebida tendo em mente o conceito de execução paralela e permite uma expressão clara e natural de paralelismo, provendo estruturas próprias para comunicação e sincronização. O modelo de programação, baseado no CSP de Hoare [44], é constituído de uma rede de processos que se comunicam através de canais de capacidade 0. Ou seja, a comunicação em **occam** se dá pelo método síncrono, também chamado de *rendezvous*.

A linguagem foi desenvolvida de modo que as mesmas técnicas de programação usadas para programação em um processador possam ser usadas para programar uma rede de processadores. O programador **occam** preocupa-se apenas com o projeto, codificação e verificação do programa. A decisão da configuração, conhecida como *pós–fragmentação*, é deixada para uma etapa final independente.

O restante desta seção descreve brevemente as estruturas principais da linguagem **occam**. Maiores detalhes sobre estas estruturas e sobre outros aspectos de linguagem são apresentados nas referências [43, 47, 48].

4.1.1 Processos primitivos

Os programas em **occam** constituem-se de processos. Um processo **occam** pode tomar formas de diferentes granularidades, desde um *processo primitivo* até um programa completo. Existem cinco tipos de processo primitivo em **occam**: atribuição, recepção de

mensagem, envio de mensagem, STOP e SKIP.

Uma atribuição em **occam** tem a forma:

```
V := e
```

onde V é uma variável e e é uma expressão do mesmo tipo. Uma expressão pode ser formada de um conjunto de variáveis, constantes, operadores e chamadas de função. O processo termina imediatamente sem sofrer suspensão ou bloqueio.

Um processo primitivo de comunicação, envio ou recepção de mensagem, é visto em **occam** no mesmo nível de uma atribuição. Como a comunicação é sincronizada, o processo que envia um dado por um canal fica bloqueado até que o destinatário esteja pronto para recebê-lo. Quando isto acontece, o dado é copiado do processo emissor para o receptor. O efeito global de uma comunicação é, portanto, idêntico ao de uma atribuição em que os elementos envolvidos, a variavél e a expressão, pertencem a processos distintos.

Um canal **occam** é unidirecional e pode ser usado somente por um par de processos. O canal é usado, lógica e sintaticamente, da mesma forma estejam os dois processos comunicantes no mesmo processador ou em processadores diferentes. A única diferença entre estes dois casos estaria na configuração do programa que faria o mapeamento do canal lógico sobre uma linha de comunicação física.

A sintaxe dos processos primitivos de comunicação é extremamente reduzida, como podemos ver no exemplo abaixo:

```
c ! v1  -- processo de saída
c ? v2  -- processo de entrada
```

onde o símbolo ! representa envio, o símbolo ? representa recepção, c representa um canal e v1 e v2 variáveis de tipo idêntico ao do canal.

Exitem ainda dois outro processos primitivos, STOP e SKIP, que são considerados especiais por não representarem nenhuma ação dentro do programa. O processo STOP nunca termina e seu único efeito é o de interromper a execução do programa. Seu uso é reservado a situações drásticas como a identificação de condições de erro e é usado implicitamente por algumas estruturas da linguagem quando empregradas incorretamente. O processo SKIP também não realiza nenhuma ação e é considerado o inverso do STOP, pois sempre termina.

4.1.2 Canais e protocolos

Os canais **occam** tem tipos definidos em suas declarações e por eles podem trafegar somente dados destes mesmos tipos. A declaração de tipo de um canal é, na verdade, uma definição de protocolo de comunicação. O tipo de um canal pode ser qualquer um dos tipos escalares da linguagem, como BYTE, INT, REAL32 ou BOOL, ou então ser formado por

```
CHAN OF INT canal.int:
CHAN OF [10]BOOL canal.seq.bool:
CHAN OF INT::[ ]BYTE canal.tamanho.variavel:
INT i, tamanho:
[10]BOOL array.bool:
[100]BYTE array.byte:
SEQ
  canal.int ? i
  canal.bool.seq ? array.bool
  canal.tamanho.variavel ? tamanho::array.byte
```

Neste exemplo, o canal chamado `canal.int` representa o tipo mais simples de protocolo que se pode definir para um canal **occam**. Por este canal passa um, e somente um, dado de um dos tipos primitivos da linguagem. O canal `canal.seq.bool` é um exemplo um pouco mais sofisticado de protocolo; por este flui um tipo de dado estruturado, uma sequência de elementos do mesmo tipo de tamanho fixo, ou seja, um *array*. No canal `canal.tamanho.variavel` podem trafegar dados inteiros agrupados em *arrays* de tamanho variável. O tamanho destas estruturas está, no entanto, limitado ao valor máximo de 255 elementos não por limitação da linguagem, mas da variável `tamanho` usada neste exemplo, que tem tipo `BYTE`.

Figura 4.1: Exemplo de declaração e uso de protocolos simples

uma coleção estruturada destes tipos. A linguagem permite a definição de três tipos de protocolos: simples, seqüenciais e variantes.

Os protocolos simples são usados para passagem de um único objeto por um canal. Este objeto pode ser de tipo escalar simples ou então organizado em forma de vetor. No trecho de programa da Figura 4.1 apresentamos um exemplo para declaração e uso de canais de cada um destes tipos.

Os protocolos seqüenciais definem canais por onde podem trafegar conjuntos heterogêneos de dados em uma mesma transação lógica. Os dados não precisam ser agrupados em nenhuma estrutura de dados, mas apenas transmitidos um a um pelo canal respeitando uma ordem pré-estabelecida. Exemplos de declaração de protocolos seqüenciais são apresentados na Figura 4.2.

Os protocolos variantes são o tipo mais flexível de protocolos **occam**. Nos outros casos, o tipo de dado, ou da seqüência de dados, transmitidos é fixo e definido na declaração do canal. Com apenas estes primeiros tipos de protocolos os processos estariam restritos a transmitir sempre os mesmos tipos de dados na mesma seqüência por um dado canal.

Como o padrão de comunição entre dois processos pode exigir a transmissão de dados segundo diferentes protocolos, não havendo um terceiro tipo seria necessário utilizar

```
CHAN OF INT; BOOL; REAL32 canal1
CHAN OF INT; INT::[ ]BOOL; REAL32 canal2:
INT i, j, tamanho:
[300]BOOL array.bool:
BOOL b:
REAL32 x, y:
SEQ
  canal1 ?  i; b; x
  canal2 ?  j; tamanho::array.bool; y
```

Figura 4.2: Exemplo de declaração e uso de protocolos seqüenciais

diferentes canais interligando um mesmo par de processos. Por cada um dos canais se daria a comunicação em um determinado protocolo devendo existir um canal diferente para cada padrão exigido.

O uso de protocolos variantes permite que um único canal seja empregado na comunicação entre um par de processos, mesmo que esta siga diversos padrões. Um protocolo variante agrupa sob um mesmo nome um conjunto de padrões de comunicação diferentes cada um rotulado com um identificador único. Na utilização de um canal com este tipo de protocolo, para que os dados sejam interpretados em seu formato correto, especifica-se sempre qual o padrão de fluxo considerado na operação. A Figura 4.3 ilustra a declaração de um protocolo variante e operações de recepção e envio em um canal com este protocolo.

O último tipo de protocolo previsto na linguagem é pré-definido e seu uso não é recomendado aos programadores. O protocolo `ANY`, chamado de *protocolo anárquico*, não define nenhum padrão de fluxo de dados. Um canal declarado com este tipo de protocolo permite o tráfego de todas as combinações de tipos e tamanhos de dados possíveis. No entanto, o compilador não realiza nenhum tipo de checagem em operações nestes canais e erros de execução podem ocorrer devido à sua utilização incorreta.

4.1.3 Estruturas de programação

Um programa **occam** é construído de blocos que agrupam declarações de dados e processos primitivos formando unidades lógicas com funções bem definidas dentro de um escopo maior. Cada bloco pode conter outros blocos respeitando-se as regras de escopo usuais.

Para a formação deste blocos a linguagem define um tipo especial de estrutura chamado de *construtor*. Existem cinco tipos de construtores em **occam** que podem ser divididos em dois grupos. `SEQ`, `WHILE`, `IF`, como veremos a seguir, assemelham-se a estruturas de outras linguagens de alto nível. `PAR` e `ALT`, por outro lado, formam um grupo de construtores particularmente dedicado à construção de processos concorrentes.

```
PROTOCOL dados
  CASE
    vet.real; INT::[ ]REAL32
    vet.int; INT::[ ]INT
    resposta; BOOL
:
CHAN OF DADOS canal
PAR
  SEQ
    canal !  vet.int; 3::[1, 2, 3]
    canal !  TRUE
  [100]REAL32 vr:
  [100]INT vi:
  BOOL r:
  INT t:
  SEQ
    canal ?  CASE
      vet.int; t::vi
        ... processa vetor de inteiros
        ... processa vetor de reais
    canal ?  CASE resposta; r
```

Figura 4.3: Exemplo de declaração e uso de protocolos variantes

SEQ agrupa um número arbitrário e ilimitado de sub-processos que devem ser executados seqüencialmente. Os sub-processos podem ser processos primitivos ou chamadas de sub-rotina (procedimentos ou funções). O agrupamento é considerado, por si próprio, um processo de maior granularidade.

O construtor WHILE age sobre um único processo determinando sua re-execução a cada vez que a avaliação de uma expressão lógica associada resulta em verdadeiro. O processo considerado pelo construtor pode ser tanto um processo primitivo como um processo definido por um SEQ.

Em **occam**, o construtor IF condiciona a execução de um processo ao resultado de uma expressão lógica. Em outras linguagens, o comando condicional tem a forma

```
IF expressão lógica THEN
  blocoV
ELSE
  blocoF
```

onde *blocoV* é executado caso a expressão resulte em verdadeiro e *blocoF* caso contrário. A estrutura do construtor equivalente em **occam** é mais sofisticada permitindo que em uma mesma estrutura associe-se a execução de um número arbitrário de processos a expressões lógicas diferentes:

```
IF
  expressão1
    processo1
  expressão2
    processo2
  ⋮
  expressãoN
    processoN
```

As expressões lógicas são avaliadas seqüencialmente até que uma delas resulte em verdadeiro, quando se executa o processo associado. Quando uma expressão assume este valor, a execução do programa continua a partir do processo que sucede o construtor IF. Exige-se que pelos menos uma das expressões resulte em verdadeiro, caso contrário considera-se atingida uma condição de erro executando-se STOP.

O construtor PAR é uma das maiores vantagens que **occam** oferece em relação às linguagens de programação convencionais. Através desta estrutura, o programador define, de maneira simples e clara, processos que devem ser executados concorrentemente. A forma genérica de uma construção PAR é:

```
PAR
  ... processo1
  ... processo2
  :
  ... processoN
```

Os processos de um PAR não podem compartilhar variáveis, ou seja, comunicam-se exclusivamente através de canais. Um construtor deste tipo tem sua execução terminada somente quando todos seus processos internos tiverem terminado.

Uma variação deste construtor permite associar prioridades aos processos concorrentes. Se dois processos se encontram em um mesmo processador e ambos estão prontos para executar, recebe preferência aquele que tiver maior prioridade. No exemplo de construção abaixo, o *processo1* tem prioridade mais alta que todos os outros.

```
PRI PAR
  ... processo1  -- mais alta prioridade
  ... processo2
  :
  ... processoN  -- mais baixa prioridade
```

Para associar um mesmo nível de prioridade a um conjunto de processos basta agrupá-los em uma construção PAR sob uma entrada de um PRI PAR.

Definida a estrutura lógica de um programa concorrente através de construções PAR, a tarefa de particionamento para execução em vários processadores requer um esforço mínimo. Na verdade, todo o esforço requerido a partir deste ponto é considerado esforço de configuração e não mais de programação. Grupos de processos paralelos são formados e atribuídos a processadores diferentes com uma variação do construtor PAR, o PLACED PAR. Os canais que interligam os processos atribuídos a processadores vizinhos são mapeados sobre *links* físicos através de comandos de configuração.

4.2 Implementação do modelo occam no T800

Internamente, o T800 consiste de memória RAM, unidade central de processamento (UCP), unidade de ponto flutuante (UFP) e um sistema de comunicação conectados por um barramento de 32 bits. A memória interna é bastante rápida, porém limitada a 4 K bytes. O barramento permite acesso a um espaço contínuo, constituído da RAM interna e de memória adicional externa.

O sistema de comunicação é formado por quatro controladores de *link* que operam linhas seriais de até 20 M bits/s. Os controladores de *link* funcionam de maneira independente, por acesso direto à memória, e podem operar simultaneamente com a UCP. Dessa forma, no caso extremo, um *transputer* pode gerenciar a um mesmo tempo a comunicação por todos os quatro canais em ambas as direções e ainda executar um processo interno.

A UCP do T800 foi projetada seguindo uma arquitetura simples e poderosa. Existem apenas três registradores de 32 bits diponíveis ao programador e normalmente usados para aritmética inteira e de endereços. Os registradores Areg, Breg e Creg formam uma pilha (*stack*) de *hardware* onde A é o topo da pilha. A carga de um valor em Areg força a passagem do seu conteúdo antigo para Breg após a transferência do conteúdo de Breg para Creg. Da mesma forma, a retirada do valor em Areg causa a passagem do valor de Creg para Breg após o valor antigo de Breg ter sido transferido para Areg. A UFP possui uma pilha semelhante para a avaliação de expressões de ponto flutuante formada pelos registradores FAreg, FBreg e FCreg.

Existem outros poucos registradores visíveis ao programador usados para controle de execução e endereçamento. Um deles é *instruction pointer*, Iptr, que armazena sempre o endereço da próxima instrução a ser executada. O outro, o *workspace pointer*, Wptr, funciona como base de endereçamento para acesso às variaveis locais e canais de um processo concorrente.

Nesse ponto é preciso introduzir conceitos superficiais do modelo de programação **occam**, que influenciou o projeto do *transputer*. Um programa executando em um *transputer* é constituído de processos concorrentes que se comunicam exclusivamente através de troca de mensagens por canais unidirecionais. Segundo sugerido em [49], um processo pode ser encarado como uma caixa preta que inicia, executa um número de ações, que podem incluir atribuições, recepção ou envio de mensagens por um canal, e termina. Cada processo pode ser constituído por vários outros processos e se comunicar por um número arbitrário de canais.

A comunicação é sincronizada. Se um canal é usado como entrada por um processo e saída por outro, a comunicação só se realiza quando ambos os processos estiverem prontos. No momento da sincronização, o dado a ser transferido é copiado do processo remetente para o processo destinatário.

4.2.1 Escalonamento de Processos

Cada processo possui um conjunto de dados referenciados localmente que são agrupados com estruturas de controle do *transputer* formando um *workspace* de endereço base WP. As variáveis locais do processo são alocadas em deslocamentos positivos a partir de WP. As posições encontradas em deslocamentos negativos de WP contém valores usados no escalonamento, comunicação e temporização. O número de palavras de memória usadas para controle depende do tipo de ações realizadas pelo processo; processos sem comunicação usam 2 palavras, processos com comunicação direta usam 3 palavras e assim por diante.

Um *transputer* executa um processo de cada vez e a concorrência em um processador é implementada por algoritmos que dividem o tempo de processamento entre os vários processos concorrentes. Os processos podem assumir prioridades de somente dois níveis: alta ou baixa. Processos de baixa prioridade são escalonados por um algoritmo *round-robin* e podem ser interrompidos pela ativação de um processo de alta prioridade. Processos de alta prioridade seguem outra forma de escalonamento: uma vez ativado um processo, este

executa ininterruptamente até que realize uma ação que cause seu próprio bloqueio. Os processos de baixa prioridade só voltam a ser executados quando não há mais nenhum processo de alta pronto para execução.

Os processos prontos para executar são organizados pelo *transputer* em duas filas segundo suas prioridades. Para isto existem dois conjuntos de registradores, que não devem ser manipulados diretamente pelo programador senão para consulta. Os registradores FptrReg0 e BptrReg0 funcionam como ponteiros para o início e fim, respectivamente, da lista de processos de alta prioridade. Os registradores FptrReg1 e BptrReg1 são análogos trabalhando sobre a lista de processos de baixa prioridade.

4.2.2 Canais e o Modelo de Comunicação

A cada canal declarado em um programa **occam** corresponde uma posição, ou palavra, de memória alocada em tempo de compilação. No início da execução de um programa, a estas posições é atribuído um valor diferente de qualquer identificação válida de processo. Este valor especial indica que o canal está vazio e, segundo a nomenclatura definida pela INMOS, é identificado por uma constante chamada `NotProcess.p`.

Um identificador de processo é formado pelo endereço de seu *workspace* somado a um dígito binário que representa seu nível prioridade. Quando um dos dois processos realiza uma operação de comunicação pelo canal que os interliga, seu identificador é atribuído à variável canal. O processo é retirado da fila de escalonamento e permanece no canal até que o processo no outro extremo do canal realize a operação dual. Quando um processo acessa um canal e o encontra ocupado, os dois processos sincronizaram-se e portanto a transferência dos dados pode ser efetuada. Em seguida, ambos processos voltam a ser escalonados, sendo cada um inserido na fila de escalonamento correspondente ao último dígito binário de seu identificador de processo.

A descrição acima é válida para canais internos a um processador. Quando a comunicação envolve processos em *transputers* distintos, o canal é mapeado sobre um *link* físico e o procedimento de transferência um tanto diferente. A sincronização e a comunicação são implementadas por *controladores de link*. Estas unidades buscam os dados no *workspace* do processo remente por acesso direto à memória (DMA) e transferem os dados por uma linha serial quando ambos os processos se sincronizam.

Um último detalhe a esclarecer sobre a implementação da linguagem **occam** no *transputer* refere-se ao protocolo anárquico `ANY`. Aparentemente, este protocolo permite a retipagem de dados comunicados. Em outras palavras, como os dados transmitidos formam uma cadeia binária, para recebê-los bastaria realizar uma operação de recepção por canal para um conjunto de variáveis cujo tamanho em dígitos binários igualasse o comprimento da cadeia.

Na implementação corrente da linguagem, uma operação deste tipo só se completa com sucesso se o canal com protocolo `ANY` estiver mapeado sobre um *link* físico. Para processos no mesmo processador que se comuniquem através de um canal de tal tipo, o formato dos dados recebidos deve ser idêntico ao dos dados tranmitidos.

Na compilação do envio de um grupo de dados, a seqüência transmitida é fraccionada em blocos. Existem instruções de transmissão para blocos do tamanho de um *byte*, `outbyte`, e para blocos de tamanho de uma palavra de memória, `outword`. E instruções de recepção de blocos de tamanhos correspondentes, `inbyte` e `inword`. As mesmas instruções operam sobre canais internos ou *links*, mas com comportamentos diferentes para cada caso.

De acordo com o tipo e a ordem dos dados em um comando de envio de mensagem, o compilador gera uma determinada seqüência de instruções composta de `outbyte` e `outword`. Para receber os dados, ao compilar uma recepção por canal, uma seqüência análoga de instruções é gerada, onde a cada `outbyte` corresponde uma `inbyte` e cada `outword` uma `inword`.

Nas comunicações com canais internos, estas instruções enxergam os dados transmitidos como blocos com tipos rigorosamente associados e nas operações sobre canais físicos, como cadeias binárias. Em alguns casos, uma comunicação sobre canal interno tipo `ANY` em que os dados são transmitidos em um formato e recebidos em outro, a operação termina normalmente. Em muitos outros casos, a operação falha e o efeito aparente é de uma comunicação que não termina. Tentativas de retipagem de dados durante a comunicação são, portanto, inseguras e devem ser sempre evitadas.

4.3 Um mecanismo de preempção de processos occam no *transputer*

Por trás de todo mecanismo de simulação distribuída existe uma política de escalonameno de eventos com critérios bem definidos. No caso dos métodos otimistas, que permitem a ocorrência de erros de causalidade, é importante que o algoritmo de escalonamento permita rapidez na correção uma vez identificado o erro. Quanto mais tempo demora a correção, maior a chance de uma computação incorreta se propagar.

No mecanismo *Time Warp*, o critério fundamental de escalonamento determina que, em um processador, deve ser sempre executado em primeiro lugar o evento associado ao menor tempo virtual. Seguindo esta estratégia, fica eliminada a possibilidade de *rollbacks* causados por processos dentro do mesmo processador. A causa de conflitos de tempos virtuais deve-se, somente, a interação entre processadores que possuem relógios independentes e progridem a diferentes taxas de processamento.

Mensagens retardatárias ou anti-mensagens, potenciais causas de *rollback*, são sempre alienígenas, ou seja, vindas de um outro processador. O processamento de mensagens deste tipo deve ter prioridade sobre o que acontece no processador. Ao receber uma mensagem externa, o núcleo de *Time Warp* de um processador deve, imediatamente, comparar seu tempo virtual de recepção verificando se causa uma modificação na ordem de escalonamento corrente.

Na eventualidade da mensagem recebida possuir um *rvt* inferior ao relógio local, o mecanismo de *rollback* deve ser acionado, os efeitos da computação incorreta corrigidos e o

processamento retomado considerando a nova mensagem. Em outras palavras, a detecção de um erro de causalidade força uma interrupção, ou preempção, na execução do processo correntemente selecionado.

O *transputer* não permite, no entanto, este tipo de escalonamento. Um algoritmo microprogramado toma as decisões de seleção de processos a executar, sem interferência externa. Para implementar, então, o *Time Warp* em processadores deste tipo é essencial, antes de tudo, que se defina uma forma de implementar algoritmos de escalonamento preemptivos. Várias experiências foram feitas na Rand Corporation [45], com versões de *Time Warp*, usando algoritmos não-preemptivos. Embora na referência possamos encontrar uma extensa comparação entre algoritmos deste tipo, não se admite, em nenhum momento, que algum destes possa ser mais eficiente que o equivalente preemptivo.

4.3.1 Arquitetura do Mecanismo de Preempção

No *transputer* há somente uma situação em que um processo pode ser interrompido: um processo de baixa prioridade perde o controle da UCP para um processo de alta prioridade. Por si só esta característica indica como deve ser estruturado um programa qualquer que tenha por objetivo a implementação de preempção.

Quando um processo de baixa prioridade é interrompido pela ativação de um processo de alta prioridade, o primeiro fica suspenso até que o segundo termine sua execução ou execute um comando que cause seu bloqueio. Quando o processo mais prioritário libera a UCP, o processo interrompido volta a executar.

Um algoritmo de escalonamento preemptivo poderia ser implementado, então, como um processo de alta prioridade selecionando processos da simulação que executam em baixa prioridade. Fazendo uso somente de recursos de linguagem de programação de alto nível, é possível realizar esta seleção através do envio de mensagens.

Na definição de uma estrutura básica para um escalonador preemptivo para *transputers*, chegamos à arquitetura de processos mostrada na Figura 4.4. Nesta estrutura o escalonador é definido como processo de alta prioridade que se comunica através de conjuntos de canais de dois tipos. Um par de canais interliga o escalonador a um sistema de comunicação que o conecta aos outros processadores da arquitetura. Um conjunto de pares de canais de entrada e saída o interliga aos processos lógicos da simulação que executam em baixa prioridade.

Nesta estrutura, um processo lógico consiste, basicamente, de um trecho de código executado a cada ocorrência de eventos e repetido indefinidamente até a terminação da simulação. A cada repetição do ciclo, o processo lógico executa os seguintes passos:

1. recebe um evento do escalonador,
2. processa o evento;
3. envia zero ou mais eventos para outros processos lógicos através do escalonador e

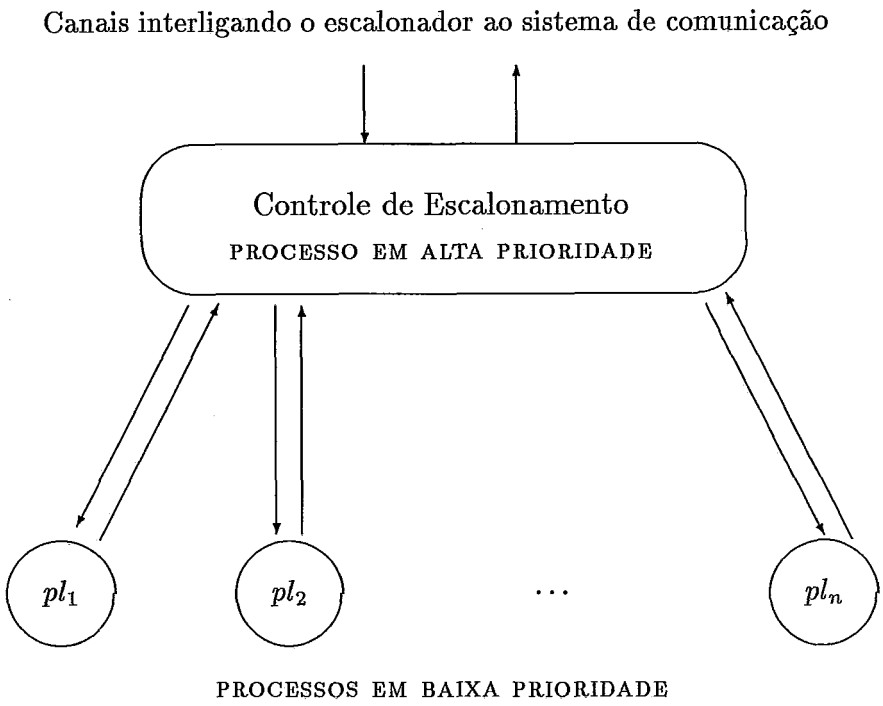

Figura 4.4: Arquitetura de Processos de um Escalonador Preemptivo

4. envia uma mensagem especial ao escalonador indicando o fim do processamento do evento.

Um processo lógico não tem capacidade de detectar o momento de terminação; esta é uma função de outra entidade no sistema. O processo deixa de executar o ciclo de processamento de eventos somente quando recebe do escalonador uma mensagem especial.

Em uma fase de inicialização que antecede a execução do primeiro evento da simulação, todos os processos lógicos informam ao escalonador dois endereços de memória, cuja função será explicada mais adiante:

- o endereço de seu *workspace*;
- o endereço de um ponto especial no início de ciclo principal de seu algoritmo; o primeiro comando no ciclo do processo corresponde a uma recepção de mensagem por seu canal de entrada. A execução do processo lógico será recomeçada a partir deste ponto sempre que sofrer interrupção do escalonador da simulação.

Após enviarem mensagens contendo estes dados ao escalonador, os processos lógicos entram em seu ciclo principal e bloqueiam-se esperando receber uma mensagem por seu canal de entrada. A fila de processos de baixa prioridade começa então absolutamente vazia e assim fica até que o escalonador dispare a execução de algum processo lógico através do envio de uma mensagem. A primeira mensagem enviada, selecionada entre os eventos iniciais da simulação, é a de menor *rvt* e é entregue ao processo lógico indicado em seu campo *rid*.

Um processo de baixa prioridade, correspondente ao processo lógico selecionado, é então inserido na respectiva fila de escalonamento do *transputer*. O processo executa o tratamento do evento e volta a se bloquear na comunicação com o escalonador ao enviar os novos eventos gerados. O escalonador é ativado, sincronizando-se com o processo lógico, e a transferência da mensagem é efetuada. Desta forma se dão as comunicações de todos os novos eventos que o processo venha a gerar, até que envie uma mensagem especial ao escalonador indicando que terminou o processamento de seu evento corrente e um novo pode ser escalonado. O processo lógico volta ao início de seu ciclo, bloqueando-se a espera de um outro evento e a fila de baixa prioridade do *transputer* torna a ficar vazia.

O escalonador avalia as mensagens nas filas de entrada dos processos lógicos buscando um novo evento a escalonar, que novamente será o que possui menor tempo virtual associado. A partir daí, o padrão de processamento do escalonador se repete e assim seria até que detecção de alguma condição de terminação, não fosse a possível chegada de uma mensagem vinda de outro processador com *rvt* menor que o associado ao evento correntemente em execução.

A chegada de uma mensagem nestas condições indica, claramente, que um erro nas decisões de escalonamento foi cometido. Um ou mais processos deverão sofrer *rollback* e os efeitos colaterais que produziram terão de ser cancelados. Quanto mais cedo for efetuada a correção deste erro, mais rapidamente pode-se recomeçar a computação a partir de um

ponto considerado, otimisticamente, correto. O instante ideal para iniciar o procedimento de correção é o da detecção do erro de causalidade, em que pode haver um processo de baixa prioridade suspenso, aguardando que o escalonador se bloqueie para terminar de processar o evento que recebeu.

De acordo com o procedimento normal do *transputer*, o processo de baixa prioridade interrompido pela ativação do escalonador continuaria a executar assim que este último se bloqueasse. Para impedir que o processo continue em um caminho incorreto da simulação é preciso evitar que retome sua execução. A operação necessária para atingir-se este objetivo é, no entanto, não trivial e não pode ser implementada usando-se apenas os recursos de linguagem de alto nível.

Caso o processo lógico continuasse a executar, como o escalonador haveria disparado a execução de um novo evento, possivelmente, em outro processo lógico, haveriam dois processos na fila de baixa prioridade. Cada processo estaria tratando eventos de diferentes fases do sistema, uma correta e outra não. O critério de escalonamento do *Time Warp* estaria sendo desrespeitado, pois apenas um dos dois poderia possuir o menor *rvt* entre os eventos no processador.

Assim, a resolução do problema do escalonamento depende de impedir a continuação da execução do processo interrompido, ou seja, de sua preempção. Para atingir este objetivo é preciso interferir no mecanismo de escalonamento do *transputer* como será visto adiante.

4.3.2 A Realização do Mecanismo

Na interrupção de um processo de baixa prioridade, o conteúdo dos registradores de controle de execução, Wptr e Iptr, é transferido para um outro par de registradores, WptrSave e IptrSave. Os registradores de controle recebem novos valores, apontando instruções e dados no escopo do processo de alta prioridade. Quando este processo termina ou se bloqueia, o conteúdo dos registradores de controle, no momento da interrupção, é restaurado a partir dos valores anteriormente salvos. Desse modo o processo de baixa prioridade volta a executar a partir do ponto em que estava no momento de sua interrupção.

De acordo com a documentação do *transputer* [49], se um processo de alta prioridade é ativado e o conteúdo do registrador WptrSave aponta para um *workspace* inválido então nenhum processo de baixa prioridade foi interrompido. Quando o processo de alta prioridade se bloqueia ou termina sua execução, o escalonador microcodificado verifica se há mais algum processo de alta prioridade pronto para executar na fila correspondente. Não havendo nenhum, verifica os valores armazenados nos registradores WptrSave e IptrSave. Se o valor em WptrSave for igual à constante `NotProcess.p`, que representa um valor inválido, o escalonador busca um novo processo para execução na fila de baixa prioridade. Se o valor for válido, o processo interrompido reassume o controle da UCP.

De acordo com estas informações, para impedir que um processo de baixa prioridade interrompido volte a executar, a primeira atitude que se deve tomar é substituir o valor do resgistrador WptrSave pela constante `NotProcess.p`. O valor antigo deve ser salvo para

que o processo possa ser recomeçado em um instante posterior a ser determinado pelo escalonador da simulação. Caso isto não seja feito o endereço do *workspace* do processo jamais será referenciado novamente pelo escalonador do *transputer* e o processo deixa de existir.

Esta operação não implementa, por si só, um mecanismo de preempção. O processo interrompido deve ter seus efeitos colaterais cancelados e ainda recomeçar sua execução a partir de um ponto diferente do em que foi suspenso. O *workspace* de um processo no *transputer* possui, como dito anteriormente, algumas estruturas de controle usadas pelo escalonador ou pelo mecanismo de comunicação entre processos. Entre estas estruturas, a posição em um deslocamento de -1 em relação ao endereço do *workspace* possui importância chave no escalonamento.

Quando um processo de qualquer prioridade é suspenso, o valor do registrador Iptr, apontando para a próxima instrução de programa a executar, é armazenado nesta posição de memória. Ao ser escalonado, o processo continua a executar do ponto onde foi interrompido, restaurado a partir do endereço indicado pelo valor salvo. Para forçar um processo suspenso a recomeçar sua execução a partir de um determinado ponto de seu código, basta então substituir o valor contido nesta posição pelo valor desejado.

Fazendo com que todo processo interrompido pelo mecanismo de preempção seja recomeçado a partir do primeiro comando de seu ciclo principal, pode-se garantir que, no máximo, haverá um processo lógico pronto para execução a cada instante. Este primeiro comando, como definido na Seção 4.3.1, é sempre uma recepção de mensagem vinda do escalonador que obriga o processo lógico a se bloquear até que o escalonador o selecione enviando-lhe um evento.

Existe ainda mais algumas situações a considerar antes de completar a definição de um mecanismo de preempção. Na primeira, verifica-se que ao ser ativado, o escalonador da simulação pode não interromper o processo de baixa prioridade do *transputer*, que representa o processo lógico em execução, no meio de uma instrução de programa, mas durante uma comunicação. Este seria o caso de um processo lógico que se bloqueia ao tentar enviar novo evento ao escalonador e este último, ao ser ativado, não encontra um endereço de *workspace* válido em WptrSave. Nesta situação o processo lógico foi interrompido, mas não o processo de baixa prioridade que o representa no processador. O processo de baixa prioridade não estava com o controle da UCP ou aguardando na fila correspondente, mas bloqueado em um canal.

Sabendo como se processam as comunicações por canais internos em um *transputer*, conclui-se que o *workspace* do processo de baixa prioridade estaria armazenado em alguma das posições de memória que representam os canais por onde se comunica. Para que o processo lógico seja interrompido, é preciso então cancelar a comunicação retirando o processo do canal, ajustar a variável de controle em seu *workspace* para que recomece do ponto correto e inserí-lo na fila de baixa prioridade do processador. A operação de cancelamento de uma comunicação já iniciada por uma das partes é chamada de *reset* de canal e é implementada por uma instrução do *transputer*, `resetch`. Esta instrução toma como operando o endereço de um canal, atribui a posição de memória que representa o canal o valor `NotProcess.p` e retorna no topo da pilha o valor antigo contido nesta

posição.

A última situação a considerar diz respeito a um caso particular da execução da porção do mecanismo *Time Warp* alocada ao processador do hipercubo onde executa o *TDS2*, o ambiente de desenvolvimento de programas. O processador onde executa este ambiente está ligado a dispositivos de entrada e saída como monitor de video e unidade de disco. Para acessar estes dispositivos, o *TDS2* cria processos de baixa prioridade que servem de interface entre programas e os servidores de entrada e saída. Logo, nesse processador nem todos os processos de baixa prioridade corresponderão a processos lógicos da simulação.

Se um processo pertencente ao *TDS2* for interrompido pelo escalonador da simulação deve receber tratamento diferente dos processos lógicos. O mecanismo de preempção não deve considerar jamais a possibilidade de cancelar uma operação destes processos, pois estaria interrompendo uma atividade sem relação direta com a simulação e que poderia ter efeitos imprevisíveis na execução do sistema. A regra básica é, interrompendo-se um processo de baixa prioridade que não pertence a simulação, sempre deixá-lo continuar, sem preempção. A preempção foi introduzida somente no nível de escalonamento de processos lógicos e não no nível de escalonamento de todos processos do processador.

Para distingüir um tipo de processos de outro existe um artifício simples informado através de comunicação com a INMOS. O *TDS2* atribui aos seus processos somente endereços de *workspace* maiores que os endereços dos processos definidos por um programa. Para descobrir se um processo pertence ao programa basta comparar o endereço de seu *workspace* com o endereço do início da área de memória livre definida pelo *TDS2*, representada pelo vetor de inteiros `freespace` que declara. Se o endereço de *workspace* for menor que este valor, certamente representa um processo definido pelo programa. Caso contrário será um processo do *TDS2*.

Considerando todos estes detalhes pode-se esboçar o funcionamento do mecanismo de preempção. Começamos assumindo que o escalonador da simulação mantém três estruturas de dados:

- uma variável `pc` que armazena o identificador do processo lógico correntemente selecionado; esta variável é atualizada a cada novo processo escalonado;
- um vetor `wp` que armazena os endereços de *workspace* de todos os processos lógicos, inicializado antes do início da simulação;
- um vetor `ip` que armazena os endereços do comando de recepção de mensagem no início do ciclo principal de cada processo lógico, tambe'm inicializado antes do início da simulação.

O roteiro do mecanismo de preempção é apresentado na Figura 4.5 seguindo a notação da linguagem **occam**. A variável `WdescIntSave` representa o registrador WptrSave, acima mencionado, que não é acessado por instrução especial de máquina. Este registrador é mapeado em memória na posição em deslocamento 11 a partir do endereço onde são carregados os programas e pode ser manipulado por instruções comuns de acesso

```
IF
  (rvt.novo.evento <= rvt.evento.corrente)
    INT WdescIntSave:
    PLACE WdescIntSave AT 11:
    INT Wptr.mais.alto, Wptr.salvo:
    SEQ
      Wptr.salvo := WdescIntSave /\ #FFFFFFFE
      ... inicializa Wptr.mais.alto
      ... reset no canal de entrada
      ... reset no canal de saída
      ... corrige valor de Iptr no workspace
      IF
        (havia processo esperando em algum canal)
          ... insere processo na fila de baixa prioridade
        TRUE
          SKIP
      IF
        (saved.Wdesc < highest.Wptr.of.EXE)
          SEQ
            WdescIntSave := NotProcess.p
            IF
              NotInLowQueue(Wptr.salvo)
                ... insere processo na fila de baixa
              TRUE
                SKIP
        TRUE
          SKIP
      ... escalona novo evento
```

Figura 4.5: Algoritmo do mecanismo de preempção

à memória. O valor contido neste registrador, no momento da interrupção de um processo, é na verdade um descritor de processo e não um endereço de *workspace* e, logo, seu último dígito binário indica a prioridade do processo interrompido. O endereço do *workspace*, armazenado em `Wptr.salvo`, é obtido zerando-se este último dígito.

No passo seguinte do algoritmo, o valor de `Wptr.mais.alto` é obtido a partir do endereço de base do vetor `freespace` declarado pelo *TDS2*. Continuando, os canais que ligam o processo lógico interrompido, `pc`, ao simulador são reinicializados e o endereço a partir do qual o processo continua a executar, após a interrupção, é ajustado para apontar para o ponto indicado em `ip[pc]`. A posição de memória a ser corrigida é apontada por `wp[pc]-1`. Em seguida, um teste é feito para determinar se algum processo foi encontrado esperando em canal e inserí-lo na fila de escalonamento de baixa prioridade, caso o teste resulte positivo.

O próximo passo implementa a preempção do processo interrompido, caso este não seja um processo criado pelo *TDS2*. O valor `NotProcess.p` é escrito no registrador WptrSave para que o *transputer* não retome automaticamente sua execução. Antes de inserir o processo na fila de baixa prioridade, é necessário verificar se o processo já não está na fila. Esta situação é, aparentemente, absurda, pois um processo não deveria estar na fila de baixa prioridade ao mesmo tempo em que o endereço de seu *workspace* está em WptrSave.

O escalonador do *transputer* funciona sempre retirando um processo de uma das filas, executando-o e depois reinserindo-o na mesma fila, caso não tenha terminado. Portanto, se um processo de baixa prioridade é interrompido, é porque estava executando e, logo, estava fora de qualquer fila. O mesmo endereço de *workspace* estar em WptrSave e na fila de baixa prioridade pode então parecer uma situação que não ocorre. Porém, segundo comunicação com pessoal de suporte da INMOS e outros pesquisadores, descobriu-se que a situação é possível e deve ser considerada sob pena de produzir erros que levariam o escalonador do *transputer* a não mais tratar um conjunto arbitrário de processos.

No último passo do algoritmo, o mecanismo de preempção ativa o escalonador da simulação para disparar a execução de um novo evento.

No capítulo seguinte, veremos como uma versão do mecanismo *Time Warp* para *transputers* pode ser construída sobre a estrutura de escalonamento preemptivo aqui definida.

Capítulo 5

Uma Versão de Time Warp

Como indicado no Capítulo 3, o mecanismo *Time Warp* possui duas propriedades que tornam sua implementação bastante prática: distribuição e simetria. De acordo com estas propriedades, a estrutura de controle do mecanismo está distribuída sobre uma rede de processadores, mas cada processador da arquitetura executa uma réplica de um mesmo código.

Neste capítulo apresentamos e discutimos uma versão de *Time Warp* adaptada para arquiteturas de memória distribuída baseadas em *transputers*. Esta versão do mecanismo segue as mesmas propriedades tradicionais, mas se organiza de modo diferente das versões desenvolvidas para hipercubos MarkII e MarkIII [6, 1].

Ao invés de dividir o mecanismo em partes encarregadas de *Controle Local* e *Controle Global*, esta implementação particiona-o em diversos processos, seguindo uma estrutura funcional bastante diferente. A estrutura apresentada a seguir fundamenta-se no mecanismo de escalonamento desenvolvido no Capítulo 4, adequando-se as dificuldades impostas pelo modelo de programação da linguagem **occam** e pelas características do *transputer*.

5.1 Visão geral

A Figura 5.1 apresenta a arquitetura de processos da versão do mecanismo *Time Warp* que estudaremos daqui por diante. Cada processador do hipercubo, possui um conjunto idêntico de processos que se interrelaciona da mesma forma.

Em cada processador executa um processo especial, de alta prioridade, encarregado de tratar comunicações com outros processadores: o *Processador de Comunicação (PC)*,

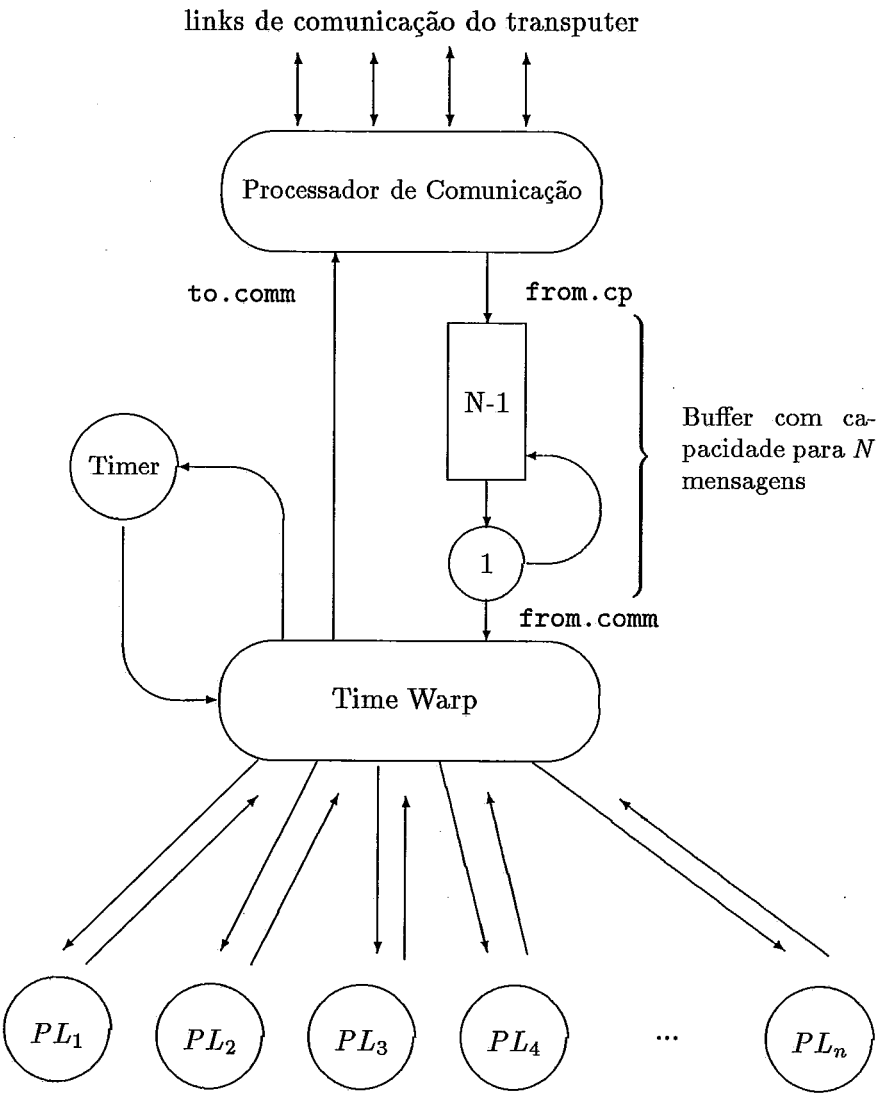

Figura 5.1: Arquitetura de um Mecanismo *Time Warp* para *transputers*

apresentado em [39]. Este processo funciona como uma unidade de *hardware* que provê serviços de comunicação ponto-a-ponto e serviços avançados, como *broadcast* em diferentes modalidades. O processo interliga a estrutura local de *Time Warp* às alocadas em outros processadores, fazendo o controle, multiplexação e demultiplexação dos *links* do *transputer*.

O *Processador de Comunicação* oferece um conjunto de instruções à camada de simulação sobre ele construída, que operam sobre dois canais:

- `from.comm`, que tem um protocolo variante de acordo com o tipo de instrução invocado. Por este canal fluem instruções e dados do simulador para a camada de comunicação que implementa os serviços;
- `to.comm`, que tem um protocolo simples para descrever a entrega dos dados vindos de outro processador para a porção local do simulador. Por este canal fluem somente dados recebidos de outros processadores e endereçados ao simulador.

Outros dois processos de alta prioridade, juntamente com o PC, formam a camada de comunicação do mecanismo *Time Warp*, que permite o desacoplamento entre processamento e comunicação que maximiza o desempenho dos *transputers*. Este dois processos implementam um conjunto de *buffers* que provê um armazenamento intermediário das mensagens vindas de outros processadores. Quando o simulador não está pronto para receber mensagens externas, os *buffers* as recebem do PC, que pode imediatamente voltar a rotear o tráfego entre processadores.

Caso o volume de mensagens externas recebidas fosse tal que ocupasse todos os *buffers* internos do PC, uma situação de *deadlock* poderia ser criada. O PC não poderia mais tratar das comunicações entre processadores até que o *Time Warp* consumisse as mensagens que armazena. Uma ou mais rotas entre processadores poderiam ficar obstruídas podendo levar à paralização de todo o sistema de comunicação. A quantidade de *buffers* a ser alocados nesta camada intermediária poderia ser determinada teoricamente, mas preferiu-se alocar um número arbitrariamente grande verificando-se experimentalmente sua suficiência.

Existem dois processos *buffer*, um com capacidade para armazenar somente uma mensagem e outro com capacidade de armazenamento para $(N-1)$ mensagens, para um valor de N determinado *a priori*. O processo com maior capacidade funciona como um *buffer* circular de $N-1$ posições, sempre pronto a receber mensagens vindas pelo canal `from.cp` até que todas suas posições estejam ocupadas. Uma mensagem só é passada adiante para o processo de um *buffer*, quando recebe um pedido de entrega de mensagem pelo canal `msg.req`. O processo de menor capacidade, funciona continuamente emitindo um pedido para o *buffer* circular, recebendo a mensagem e bloqueando-se até que o simulador esteja pronto para receber uma mensagem. Desta forma os canais `from.cp` e `from.comm` podem operar concorrentemente adequando o fluxo de mensagens da camada de comunicação à capacidade de processamento do simulador.

O processo *Time Warp* implementa as funções de controle local da simulação agindo, também, como parte do mecanismo de controle de atividades globais como veremos adi-

ante. Todas as funções da simulação, como gerenciamento de filas, *rollback*, registro de estados, cancelamento de mensagens, controle de fluxo, cálculo de *GVT* e entrada e saída são controladas por este processo. O *Time Warp* executa, também, em alta prioridade escalonando os processos lógicos *(PLs)*, de baixa prioridade, como descrito na Seção 4.3.1.

Somente mais um processo executa no nível de alta prioridade, o processo *Timer*. Em determinadas atividades, o *Time Warp* necessita de uma forma de marcação de tempo real. A linguagem **occam** provê um tipo especial de canal para este fim, também chamado `TIMER`, mas com uma semântica que não atende às necessidades do mecanismo. Como veremos ao discutir mais profundamente o funcionamento do simulador, um mecanismo de marcação de tempo semelhante a um alarme de contagem regressiva deve estar disponível. Espera-se que este mecanismo possa ser disparado, congelado ou reinicializado em um instante qualquer e ainda enviar um sinal ao simulador caso a contagem seja terminada.

O processo *Timer* comunica-se com o *Time Warp* através de dois canais:

- `to.timer`, por onde recebe comandos do simulador que podem ser `t.start` ou `t.stop`. Ao receber um `t.start`, o processo começa uma contagem regressiva a partir de um valor pré-estabelecido. Se um comando deste tipo for emitido durante uma contagem já iniciada, o processo recomeça a partir do valor inicial. Um comando `t.stop` interrompe uma contagem antes que se complete, como que congelando um cronômetro, que fica parado até ser ordenado o início de uma nova contagem.
- `from.timer`, por onde envia um sinal `t.wake.up` indicando que a contagem foi terminada e, portanto, um determinado intervalo de tempo real se passou.

Os processos restantes, no esquema da arquitetura, são os processos lógicos da simulação, que executam em baixa prioridade. Estes processos possuem um identificador local, atribuído pelo próprio simulador e usado para acessar suas estruturas de dados dentro do *Time Warp*, que são organizadas em forma de vetor. Os processos possuem, também, uma identificação global atribuída pelo programador da simulação que é informada ao simulador em fase de inicialização para a construção de tabelas de roteamento.

Dois vetores de canais ligam os processos lógicos ao simulador:

- `from.simulator`, por onde o simulador entrega a cada processo lógico eventos a processar. Além dos dados contidos na mensagem do evento, o simulador informa ao processo lógico a identificação global do seu remetente *(sid)* e seu tempo virtual de envio *(svt)*.
- `to.simulator`, por onde o processo lógico envia mensagens de evento para que simulador as despache até o processo destino indicado pelo identificador global *rid* no tempo virtual de recepção *rvt*.

Partindo desta visão geral do mecanismo *Time Warp*, nas seções seguintes discutiremos em detalhe os aspectos mais importantes do sistema e de sua implementação. Começando pelo mais simples, apresentamos a seguir a forma geral de um processo lógico.

```
VAL INT pid IS id.local:
VAL INT gpid IS id.global:
... Declaração de Variáveis de Estado
... Declaração de Variáveis Temporárias
SEQ
  ... Seção de Inicialização
  ... Seção de Tratamento de Eventos
  ... Seção de Terminação
```

Figura 5.2: Modelo sintético de um processo lógico

Em seguida, descrevemos o funcionamento do processo principal do simulador atentando para soluções propostas para os maiores problemas de projeto e realização.

5.2 Os Processos Lógicos

O modelo de programação de processos lógicos aqui adotado, semelhante ao definido por Jefferson[2], divide o código em seções bem definidas cada uma com uma função específica em uma etapa da simulação (ver Figura 5.2).

Antes de mais nada o processo define dois valores de identificação: `pid` que é um identificador usado apenas pelo processo *Time Warp*, no mesmo processador, para acesso a estruturas de dados do processo lógico e `gpid` que é um valor que identifica unicamente o processo no sistema. Inicialmente, os valores de identificação global dos processos são conhecidos apenas em seu escopo, para serem depois difundidos por todos os processadores do sistema na construção de tabelas de roteamento de mensagens.

As variáveis declaradas no processo podem ser de dois tipos, *variáveis de estado* ou *variáveis temporárias*. As variáveis de estado constituem o conjunto de dados que define um estado do processo e que deverá ser periodicamente salvo pelo *Time Warp* para implementação do mecanismo de *rollback*. O modelo de programação da linguagem **occam** proíbe terminantemente que processos dentro de um mesmo *transputer* compartilhem dados e, portanto, seria impossível ao *Time Warp* acessar quaisquer dados declarados no escopo de um processo lógico. Normalmente, o compilador detecta e acusa situações que contrariem este modelo, mas existe uma maneira de contornar esta restrição através de uma opção de compilação.

No escopo mais externo do programa, que contém os processos **occam** da arquitetura do mecanismo, é declarado um grande vetor `state.var` e outros vetores auxiliares

`state.ptr` e `state.size`. O primeiro é um vetor de elementos de 32 bits (dimensão de uma palavra de memória no *tranputer* T800) de tamanho igual à soma dos tamanhos de todas as variáveis dos processos lógicos. Sobre este espaço são mapeadas todas as variáveis de estado de cada processo em posições contíguas. Para um processo qualquer identificado localmente por `pid`, a área do vetor `state.var` alocada tem tamanho `state.size[pid]`, começando na posição indicada por `state.ptr[pid]`.

Os processos lógicos compartilham o mesmo vetor `state.var`, mas acessam áreas sempre distintas. O vetor será compartilhado, efetivamente, entre processos lógicos e simulador, mas não entre os próprios processos lógicos. Os processos lógicos trocam informações entre si através de mensagens que caracterizam eventos. As informações referentes ao estado de um processo lógico poderiam, também, ser passadas ao simulador através de mensagens, mas com um custo adicional. O simulador, podendo acessar diretamente o espaço de endereçamento de cada processo lógico, torna-se capaz de registrar estados com um custo mínimo.

Um outro tipo de variáveis representa dados temporariamente empregados pelo processo no tratamento de um evento, sem consistir informação que faça parte do estado de um processo. Os valores destas variáveis podem ser alterados arbitrariamente entre o processamento de dois eventos consecutivos e não são registrados pelo *Time Warp* e nem resturados quando ocorre um *rollback*.

Após as declarações de variáveis, encontramos em um processo lógico a Seção de Inicialização. Esta parte do processo é executada na fase de inicialização do *Time Warp* em que os identificadores globais são informados ao simulador. Nesta seção agrupam-se todos os comandos de inicialização de variáveis, de estado ou temporárias, geração de eventos iniciais e inicialização de estruturas de controle do escalonador preemptivo. O próprio processo descobre o endereço de seu *workspace* e do ponto de onde recomeça a executar quando for interrompido, passando estas informações ao *Time Warp* que as armazena nos vetores `wp[ ]` e `ip[ ]`. O envio destas informações marca o término da seção.

A Seção de Tratamento de Eventos consiste de um ciclo executado indefinidamente, até que o simulador informe o processo da terminação da simulação. Como apresentado na Figura 5.3, este ciclo inicia com a recepção de uma mensagem vinda do simulador que indica de quantas mensagens se compõe o próximo evento a executar. Normalmente, um evento corresponde a uma mensagem, mas quando várias mensagens são endereçadas a um mesmo processo lógico para serem processadas em um mesmo *rvt* todas estas fazem parte de um único evento chamado, neste caso de *evento múltiplo* ou *super-evento*. Todas as mensagens que compõem um mesmo evento devem ser recebidas antes que o processamento do eventos se inicie, armazenando-se em variáveis temporárias ou de estado os dados que forem necessários.

No processamento do evento, os dados recebidos nas mensagens são usados para determinar alterações sobre as variáveis de estado do processo e para gerar novos eventos que são enviados ao simulador pelo canal `to.simulador[pid]`. A variante de protocolo do canal usada neste caso é semelhante a `d.msg` empregada na recepção de mensagens. Uma mensagem é enviada através de um comando:

```
suicídio := FALSE
WHILE NOT(suicídio)
  SEQ
    from.simulator[pid] ?  CASE
      mcount; n.de.eventos
        SEQ
          now := Clock(pid)
          i := n.de.eventos
          WHILE (i > 0)
            SEQ
              from.simulator[pid] ?  CASE d.msg; sid; svt; tam::buffer
              ... atualiza variáveis
              i := i - 1
          ... processa evento
          to.simulator[pid] !  no.message
      terminate.lp
        suicídio := TRUE
```

Figura 5.3: Algoritmo básico da Seção de Tratamento de Eventos

```
to.simulator[pid] !  u.msg; rid; rvt; tam::buffer
```

onde `rid` é o identificador global do processo destinatário, `rvt` o tempo de recepção da mensagem, `tam` o seu tamanho em *bytes* e `buffer` um vetor contendo os dados a serem transmitidos. A primeira posição deste vetor será usada de maneira especial pelo *Time Warp* para implementar um serviço oferecido ao programador da simulação. Este campo armazena um *sinal* da mensagem, atribuído pela aplicação, para que esta possa também criar anti-mensagens que cancelem comunicações anteriores.

Ao terminar de enviar as novas mensagens geradas a partir do processamento do evento, o processo lógico envia uma mensagem especial para o simulador, `no.message`, que indica que não há mais mensagens a transmitir e o escalonador pode iniciar o próximo evento. O ciclo torna a se repetir e a seção só termina quando o processo lógico recebe do simulador a mensagem `terminate.lp`.

A Seção de Terminação, a última executada pelo processo lógico, trata de passar ao simulador os dados coletados durante a simulação para que sejam registrados em dispositivo de entrada e saída, neste caso em arquivo em disco. Através de mensagens especiais, os processadores que não tem acesso direto aos dispositivos, enviam informações para um processador específico habilitado a realizar operações de escrita em arquivo. Os dados registrados são determinados pelo próprio processo lógico; todas mensagens enviadas nesta fase são fielmente gravadas em arquivo.

5.3 O gerente *Time Warp*

O processo *Time Warp* é responsável, em cada processador, pelo controle de vários aspectos da simulação. Como veremos a seguir, este processo realiza gerência de memória, controle de fluxo, detecção de terminação, roteamento de mensagens e escalonamento, entre outras funções.

Diversas atividades no escopo deste processo exigem alocação dinâmica de memória, um recurso não é oferecido pela linguagem **occam**. Para contornar esta dificuldade definiu-se uma estratégia semelhante a adotada no caso das variáveis de estado dos processos lógicos. Estimando-se a memória livre em um processador após a carga do sistema, definine-se um vetor de palavras de memória (inteiros de 32 bits), `memory[ ]`, dimensionado de modo a cobrir todo este espaço. O vetor é dividido em partes de tamanho fixo, formando um conjunto de páginas de memória que podem ser reservadas ou liberadas de acordo com as necessidades do processo. Os elementos nas filas de mensagens ou estados são sempre do tamanho de uma página de memória, dimensionada para conter inteiramente o maior item.

Apesar de cada página ser formada de elementos de tipo inteiro, através de conversões forçadas, qualquer tipo de dados pode ser armazenado neste espaço. Uma página de memória pode, então, armazenar um estado ou mensagem sejam quais forem os dados aí contidos.

```
SEQ
  ... registra estado inicial dos processos
  ... inicializa tabelas locais de alocação de processos
  ... coleta eventos iniciais da simulação e
      informações para o mecanismo de preempção
  ... escalona evento de menor rvt (se houver algum)
  WHILE (gvt < +∞)
    SEQ
      PRI ALT
        from.comm ?  CASE p.msg; tam.msg::buffer.msg
          ... trata mensagem vinda de outro processador
        ALT i=0 FOR NPROCS
          (NOT(cancelback.on)) & to.simulator[i] ?  CASE
            u.msg; rid; rvt; tam.texto::texto
              ... copia mensagem para uma pagina de memória e despacha
            no.message
              ... salva estado do processo e escalona próximo evento
        (evento.completo AND (NOT(cancelback.on))) & from.timer ?  CASE
          t.wake.up
            Computa.GVT(terminação)
  ... termina processos lógicos
```

Figura 5.4: Algoritmo básico do processo *Time Warp*

Conhecendo as caracteríticas básicas do sistema de memória de um processador, pode-se passar a descrição do algoritmo do processo *Time Warp*, apresentado na Figura 5.4, também chamado de gerente da simulação. Este algoritmo descreve, em alto nível, as atividades principais realizadas pelo gerente. Aspectos mais profundos ligados a estas atividades serão apresentados no restante do capítulo.

O algoritmo inicia registrando os estados locais iniciais de cada processo lógico como definidos nas Seções de Inicialização de cada um. Ao término do registro, cada uma das filas de estados dos processos alocados neste processador contém um elemento associado ao tempo inicial da simulação, representado por convenção pelo valor $-\infty$.

No passo seguinte, as tabelas de alocação de processos são inicializadas com dados recebidos dos processos lógicos neste processador. Duas tabelas de processos são definidas no gerente:

- `local.proc.table[pid]`, que associa a cada valor de identificador de processo alo-

cado a este processador um identificador global único no sistema;

- `global.proc.table[gpid]`, que associa a cada valor de identificador global de processo o identificador do processador onde está alocado.

Nesta etapa, cada processador preenche totalmente sua tabela local, mas somente a porção da tabela de alocação global referente os processos que abriga. O preenchimento das tabelas globais é terminado no passo seguinte do algoritmo, em que os processadores trocam entre si os pedaços de tabela construídos localmente, até todas as tabelas globais tenham sido completadas. Este último passo é sincronizado por barreira, de forma que nenhum processador pode passar ao seguinte sem que todos os outros processadores estejam prontos para fazê-lo.

O algoritmo prossegue na fase de inicialização, coletando de cada processo lógico os eventos iniciais da simulação. Os processos são atendidos em ordem crescente de identificadores locais e o *Time Warp* recebe de cada um uma seqüência de eventos iniciais terminada por mensagem especial indicando fim de sua Seção de Inicialização. Para cada evento inicial recebido, o gerente aloca uma página de memória, despachando-o para o destino indicado na mensagem. Eventos endereçados a um processo no mesmo processador são inseridos na fila de entrada do destinatário. Eventos remetidos a processos em outros processadores são inseridos na fila da saída do remetente e encaminhados ao *PC*, que os envia ao processador indicado na tabela `global.proc.table`, onde está alocado o destinatário. A mensagem que termina a seqüência de eventos iniciais, informa ao gerente o endereço do *workspace* do processo lógico e o ponto de onde deve ser recomeçado no caso de preempção.

Neste ponto, o gerente está pronto a iniciar a localmente a simulação e escalona o evento de menor *rvt* presente nas filas de entrada dos processos lógicos que trata. Havendo ou não eventos iniciais para escalonar, o processo *Time Warp* entra no ciclo principal de seu algoritmo, em que permanece até que o valor do tempo virtual global, `gvt`, chege a $+\infty$ quando a simulação é considerada terminada por consenso de todos os outros processadores.

O processamento realizado no ciclo principal do gerente trata, de forma priorizada, a recepção de mensagens de diferentes tipos e origens em um `PRI ALT`, comando de seleção da linguagem **occam**. Neste comando estão representados os três tipos de canais de entrada do processo:

- `from.comm`, por onde chegam mensagens de outros processadores. Para cada mensagem recebida é aplicado um tratamento, que varia de acordo com seu tipo e com a disponibilidade de memória no processador. Havendo espaço, a mensagem é armazenada em uma página de memória e inserida na fila de entrada do processo lógico a que se destina, possivelmente causando *rollback*. Quando a ocupação de memória atinge um valor crítico é ativado o protocolo de controle de fluxo e gerenciamento de memória *Cancelback*, descrito adiante. Este protocolo pode causar um bloqueio temporário das atividades no processador, até que se possa recuperar um número mínimo de páginas de memória. Durante esta fase, todas as mensagens de evento

recebidas são devolvidas ao remetente. Outras mensagens de controle continuam a ser tratadas pelo processador.

- `from.simulator[ ]`, por onde chegam mensagens segundo o protocolo `u.msg`, usado na recepção de eventos enviados pelos processos lógicos para serem despachados pelo gerente ou então, sinais de fim de evento representados por um sinal, segundo o protocolo `no.message`. Ao receber um evento de um processo lógico, o gerente armazena os dados comunicados em uma página de memória, fazendo o roteamento da mensagem de acordo com o processador onde está alocado o destinatário. O recebimento de um sinal de fim de evento indica ao *Time Warp* que o estado do último processo lógico executado deve ser salvo e inserido na fila apropriada. Caso a variável booleana `cancelback.on` possua valor verdadeiro, o protocolo *Cancelback* está ativado e nenhuma atividade que consuma memória pode ser realizada. Neste caso, mensagens de evento não pode ser recebidas por não haver como armazená-las e sinais de fim de evento não podem ser tratados, pois não há espaço para salvar um estado do processo.

- `from.timer`, por onde chega um sinal vindo do processo *Timer*, indicando que um intervalo pré-estabelecido de tempo real foi completado sem que nenhum evento tenha sido escalonado neste processador. Toda vez que o escalonador não encontra nenhum evento a processar, dispara uma contagem no *Timer*. Caso algum evento surja antes da sinalização de fim da contagem, o *Timer* é congelado. A recepção de um sinal `t.wake.up` indica a possilidade de a simulação ter terminado globalmente. Para avaliar se esta suspeita é verdadeira o gerente dispara o cálculo do tempo virtual global. Se o valor resultante for igual a $+\infty$ a simulação está encerrada e o processo termina. A recepção de um sinal por este canal fica desabilitada caso o processamento do último evento escalonado não tenha se completado. Sem este cuidado, suspeitar que a simulação houvesse terminado seria um ato de precipitação, pois o processamento do evento corrente poderia gerar ainda um ou mais novos eventos.

Atingido o tempo virtual global $+\infty$ o ciclo termina e o gerente passa a comandar os processos lógicos a encerrarem sua Seção de Tratemento de Eventos, passando a executar a Seção de Terminação. Os processos são, então, atendidos em seqüencialmente enviando-se os dados a serem escritos em disco ao processador ligado ao dispositivo.

O restante deste capítulo explica, em detalhes, as principais atividades realizadas no processo *Time Warp*, comentadas até agora superficialmente.

5.3.1 Rollback e Cancelamento de Mensagens

Mensagens trocadas entre processos lógicos em um mesmo processador não causam *rollback*. Como todo o processamento de eventos respeita as regras de causalidade, os novos eventos produzidos têm sempre tempo virtual de recepção maior que o valor de relógio local no momento de sua criação. Como os eventos em um processador são escalonados em ordem não decrescente de *rvt*, os novos eventos produzidos possuem tempos de recepção sempre no futuro virtual dos seus destinatários e, por isso, não causam *rollback*.

As mensagens vindas de outros processadores, no entanto, podem chegar no passado virtual dos processos lógicos, pois os processadores não seguem nenhuma regra global de escalonamento. Por ser otimista, o mecanismo *Time Warp* assume que decisões de escalonamento podem ser tomadas localmente com grande probabilidade de não estarem incorrendo em erros de causalidade.

Como explicado anteriormente, a correção de um erro de causalidade implica não somente na restauração de um estado no passado virtual do processo onde ocorre o conflito, mas também no cancelamento dos efeitos colaterais causados sobre outros processos. Estas operações são implementadas, sobre um conjunto de estruturas de dados que lhes serve de base.

Nas implementações tradicionais para sistemas de memória distribuída, este conjunto de estruturas é formado por filas de entrada e saída de mensagens e filas de estados de processos, tal qual apresentado na Seção 3.2.1. A cada fila de entrada de mensagens associa-se um ponteiro de *próxima mensagem a processar* incrementado toda vez que se completa o processamento do evento corrente. Quando um *rollback* ocorre, este ponteiro é reajustado para uma mensagem que antes fazia parte do passado virtual do processo, a de maior tempo de recepção inferior ao tempo do conflito causal. O estado restaurado é, também, o de maior tempo virtual inferior ao tempo do conflito.

Vejamos a situação apresentada na Figura 5.5 que mostra a fila de entrada de um processo lógico. Uma mensagem com $rvt = 0.7$, vinda de outro processador, chega ao processo quando a próxima mensagem a processar possui $rvt = 2.3$. Assumindo que estados são gravados ao término do processamento de cada evento, o estado que deve ser restaurado foi gravado no tempo virtual 0.5. O ponteiro de próxima mensagem a processar deve ser ajustado para a mensagem retardatária e todos os eventos resultantes do processamento dos eventos seguintes na lista deve ser cancelado.

Um outro tipo de situação em que ocorre *rollback* é mostrado na Figura 5.6. Uma anti–mensagem no passado do processo lógico cancela uma mensagem ao ser inserida em sua lista de entrada. O ponteiro de próxima mensagem a processar volta para a mensagem que sucedia a mensagem cancelada na lista. O estado que deve ser restaurado foi produzido ao fim do processamento do evento que antecedia a mensagem cancelada na lista.

Em ambas as situações, as implementações tradicionais de *Time Warp* fariam o cancelamento dos efeitos colaterais produzidos pelo processamento dos eventos posteriores ao tempo do conflito enviando anti-mensagens. Mesmo algumas implementações para máquinas de memória compartilhada operam desta forma [5]. A decisão de enviar as anti–mensagens pode ser tomada em dois momentos diferentes, o que cria dois métodos possíveis para o cancelamento de mensagens, cujo desempenho é analisado em [17].

O método chamado de *Cancelamento Agressivo*, ou *Aggressive Cancellation*, determina que as anti–mensagens devem ser enviadas no momento da detecção do erro de causalidade. Desta forma, o avanço da computação incorreta é detido no menor tempo possível. Por outro lado, nada garante que, ap'[216zos o *rollback*, o reprocessamento dos eventos que haviam gerado aquelas mensagens não as recrie exatamente iguais. Em casos como este, verifica-se que uso de *Cancelamento Agressivo*, que pode permitir ganhos de

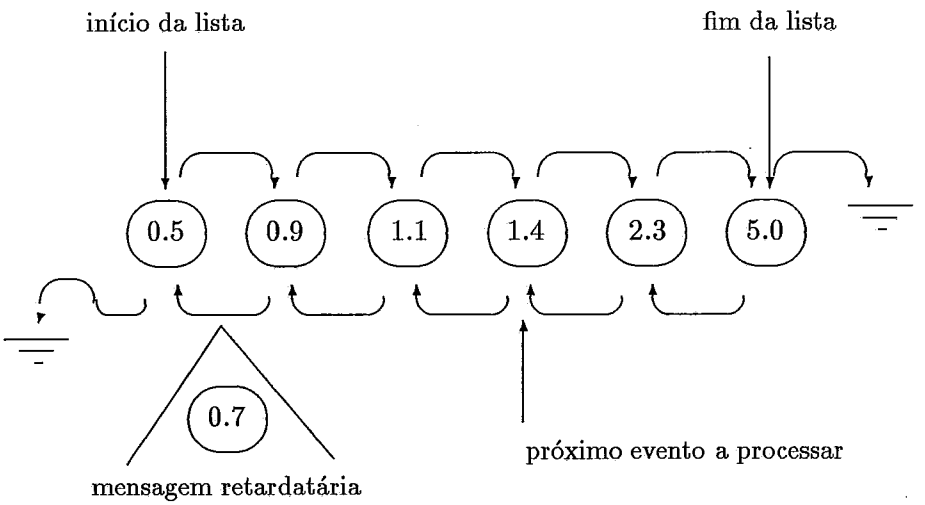

Figura 5.5: Exemplo de *rollback* por chegada de mensagem retardatária

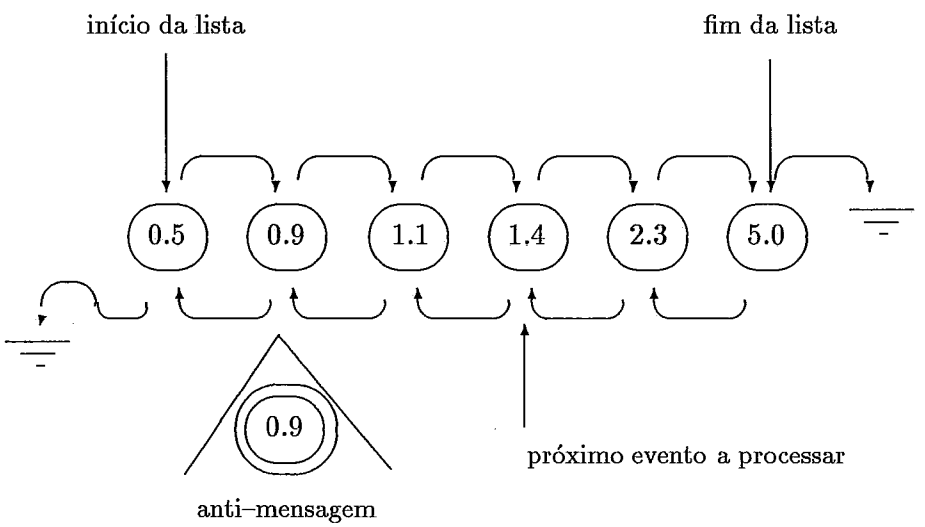

Figura 5.6: Exemplo de *rollback* por chegada de anti–mensagem

tempo, pode também cancelar eventos corretos da simulação.

A alternativa a esta estratégia é não enviar as anti–mensagens até que se tenha certeza que os eventos não serão recriados quando a computação volta a progredir. Este método é chamado de *Cancelamento Preguiçoso*, ou *Lazy Cancellation*. Segundo esta política, após a restauração do estado passado, a computação volta a evoluir e só são enviadas anti–mensagens para os eventos que vão sendo criados diferentes dos antigos. O custo desta estratégia se reflete sobre, pelo menos, três aspectos:

- o tempo levado para cancelar o avanço de uma possível computação incorreta aumenta;
- a utilização do espaço de memória ocupada tende a ser menos eficiente, uma vez que algumas páginas ficam alocadas por um tempo maior que o estritamente necessário;
- existe um custo de processamento para a comparação das novas mensagens geradas com as antigas.

Outras implementações para sistemas de memória compartilhada, realizam o cancelamento de eventos sem o uso de anti–mensagens [9]. Como simulador tem acesso direto a todo o espaço de endereçamento utilizado na simulação, ao invés de enviar uma anti–mensagem para cancelar seu equivalente, o simulador pode simplesmente apagá-la quando determinar necessário. Este mecanismo depende de um conjunto de estruturas de dados diferentes das tradicionais. Como não faz uso de anti–mensagens, o mecanismo dispensa a existência de filas de saída.

Este mecanismo, chamado de *Cancelamento Direto*, ou *Direct Cancellation*, é compatível com ambas as estratégias de cancelamento acima definidas. As modificações nas estruturas de dados do *Time Warp* são, relativamente, simples. Cada mensagem, na fila de entrada de um processo lógico, possui ponteiros que formam um *registro de causalidade*. No processamento da mensagem, a medida que outros eventos vão sendo gerados, os ponteiros de seu registro de causalidade são ajustados para indicar o endereço de cada um destes eventos. Desta forma, observa-se que *árvores de causalidade*, que espelham as relações de precedência entre os eventos, vão sendo criadas a partir dos eventos iniciais da simulação. Uma vez que o simulador decida cancelar os eventos decorrentes do processamento de uma determinada mensagem, a sub-árvore começada nesta mensagem é percorrida apagando-se todos seus nós, exceto a raiz, possivelmente causando *rollback* em outros processos.

Analisando estes mecanismos de cancelamento, pode-se observar que uma solução diferente e mais eficiente pode ser proposta para implementações de *Time Warp* em máquinas de memória distribuída. Em geral, em cada processador, a memória utilizada pela simulação é dividida em itens agrupados em um *pool* comum a todos os processos.

No caso desta implementação, a situação não é diferente. Como o processador pode acessar diretamente todos os endereços em seu espaço de memória, o gerente da simulação pode implementar árvores de causalidade ligando os eventos que ocorrem no mesmo processador. Os eventos escalonados para processos em outros processadores criariam anti–mensagens inseridas em filas de saída. Esta solução resulta, claramente, de uma fusão de

técnicas de cancelamento de mensagens para arquiteturas de memória compartilhada e distribuída.

Uma árvore de causalidade, segundo este esquema, pode conter dois tipos de nó:

- eventos enviados para processos lógicos no mesmo processador, ou *evento internos* e
- anti-cópias de eventos enviados para processos lógicos alocados em outros processadores.

Uma árvore qualquer, tem como raiz um *evento externo*, ou seja, um evento recebido de um processo lógico que não alocado no mesmo processador ou um evento inicial da simulação. As folhas desta árvore são eventos internos ainda não processados ou então anti–mensagens nas filas de saída. Os nós internos da árvore são sempre eventos internos.

Vejamos o exemplo da Figura 5.7. A raiz da árvore é um evento já processado pelo processo lógico 0 que gera dois outros eventos destinados a processos lógicos pertencentes ao mesmo processador. Estes eventos, ao serem processados, geram outros que vão incrementando a árvore. Quando um evento externo é adicionado à estrutura, forma-se uma folha da árvore, pois este evento será computado em outro processador, para o qual não é possível estender um ponteiro de memória válido. Os outros nós terminais da árvore, representam eventos que ainda não foram processados e, portanto, não poderiam tera criado outros como conseqüência.

Organizado desta forma, o mecanismo de cancelamento de mensagens pode produzir melhores resultados que outras implementações quando se executam simulações de alta *localidade espacial*. Em outras palavras, simulações programadas de forma a privilegiar a comunicação entre processos alocados em um mesmo processador podem se beneficiar de um controle mais eficiente sobre a expansão de erros de causalidade. Com este mecanismo, associado a *Cancelamento Agressivo* ou *Preguiçoso*, a correção de conflitos causais confinados em um processador pode ser bastante eficiente. Quando a esfera de influência de um conflito se estende por vários processadores, ainda assim a computação tende a ser corrigida em tempo menor. Este mecanismo permite, também, uma melhor utilização de memória. Não se criam réplicas de mensagens trocadas entre processos no mesmo processador, diminuindo o tamanho das filas de saída.

5.3.2 O algoritmo de Cálculo de *GVT*

O *Tempo Virtual Global (GVT)* é uma grandeza de extrema importância no funcionamento do mecanismo *Time Warp*. Como dito anteriormente, o *GVT* é empregado para controle de fluxo, gerenciamento de memória e detecção de terminação. Por tratar de uma grandeza definida em função de valores distribuídos nos espaços de memória de diversos processadores, o cálculo de *GVT* exige o uso de um algoritmo distribuído e não trivial.

Os problemas de *mensagens em trânsito* e *notificação simultânea* admitem diversas soluções, mais ou menos eficientes. Desde a primeira implementação de *Time Warp* vários algoritmos para cálculo de *GVT* foram propostos. Cada um destes algoritmos propõe

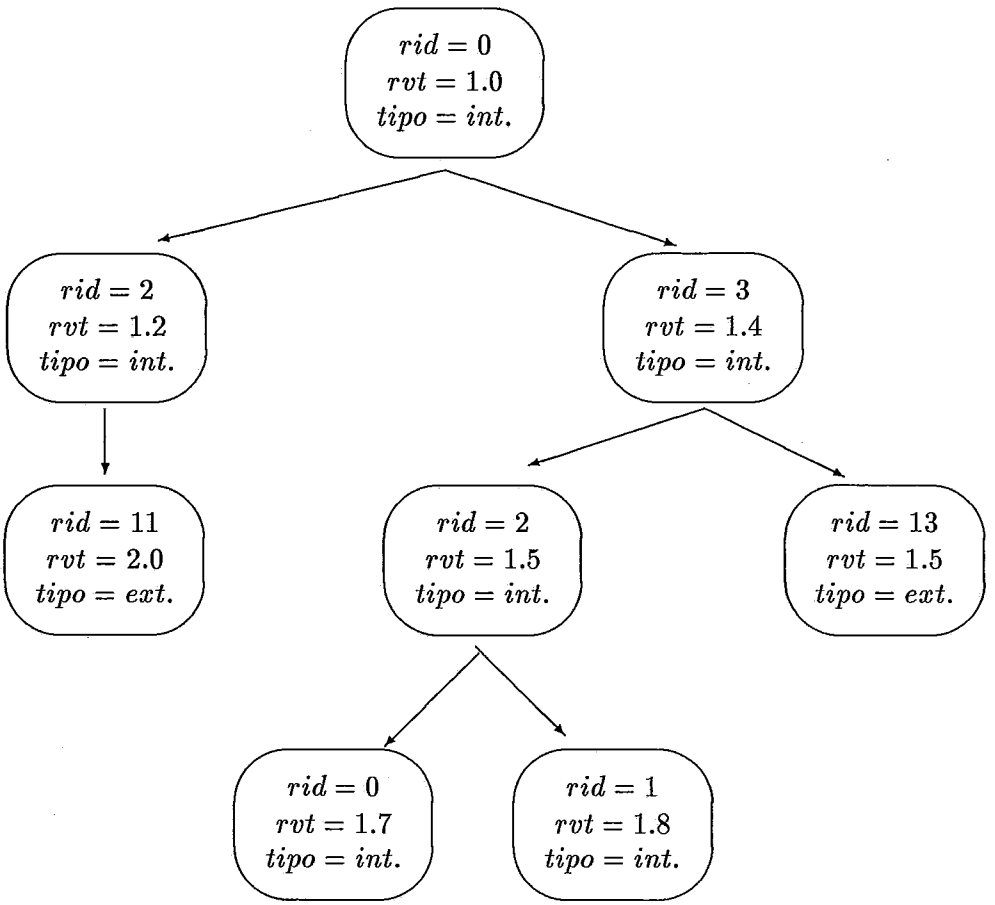

Figura 5.7: Exemplo de árvore de causalidade

soluções próprias para os problemas associados a esta computação, no entanto, nenhum destes foi concebido dentro da visão que apresentamos a seguir.

Para fins de cálculo de GVT, podemos tratar o sistema definido pelo mecanismo *Time Warp* e os processos lógicos da simulação de forma bastante simplificada. Consideremos que todo o *Time Warp* seja representado por um grafo orientado em que cada vértice representa um processador e cada aresta uma linha de comunicação entre um par de processadores. Embora as ligações físicas entre os processadores sigam a arquitetura definida pela rede de comunicação do hipercubo, o *Processador de Comunicação* permite que um processador troque mensagens com qualquer outro e, portanto, deve-se considerar que o sistema, ao menos logicamente, tem a topologia de um grafo completo. Em cada processador, ou vértice do grafo, estão alocados processos lógicos que constituem um subconjunto dos processos da simulação.

Um estado global $\mathcal{GS}$ deste sistema pode ser definido como uma coleção de valores de relógios virtuais e seqüências de mensagens em trânsito pelos canais lógicos entre cada par de processadores. A partir dos dados registrados em $\mathcal{GS}$ podemos relacionar algumas grandezas que nos auxiliam a enunciar formalmente o GVT. Seguindo uma notação semelhante a de Lin [38], apresentamos abaixo uma redefinição de GVT, modificada de acordo com as necessidades do protolo *Cancelback* discutido na seção seguinte.

Sejam:

- $G = (P, C)$, o grafo completo que representa o sistema, onde P é o conjunto de processadores e C o conjunto de canais lógicos de comunicação implementados pelo *Processador de Comunicação*;
- m, uma mensagem qualquer trocada entre dois processadores;
- $rvt(m)$, o tempo virtual de recepção da mensagem m;
- $svt(m)$, o tempo virtual de envio da mensagem m;
- $TRF_{p,q}(\mathcal{GS})$, o conjunto de mensagens em trânsito enviadas de um processador p para um processador q registradas em $\mathcal{GS}$;
- $TRF_q(\mathcal{GS}) = \cup_{p \in P} TR_{p,q}(\mathcal{GS})$
- $TRB_{q,p}(\mathcal{GS})$, o conjunto de mensagens em trânsito enviadas de volta de um processador q para um processador p registradas em $\mathcal{GS}$;
- $TRB_p(\mathcal{GS}) = \cup_{p \in P} TRB_{q,p}(\mathcal{GS})$
- $lc_{i,p}(\mathcal{GS})$, o relógio virtual local do processo i alocado no processador p, como consta em $\mathcal{GS}$;
- $ML_p(\mathcal{GS})$, o mínimo local de tempos virtuais do processador p definido como

$$ML_p = \min \left[\min_{\forall i \in p} lc_{i,p}(\mathcal{GS}), \min_{m \in TRF_p(\mathcal{GS})} rvt(m), \min_{m \in TRB_p(\mathcal{GS})} svt(m) \right] \quad (5.1)$$

O GVT, no estado global $\mathcal{GS}$, pode então ser expresso em uma única equação:

$$GVT(\mathcal{GS}) = \min_{p \in P} ML_p(\mathcal{GS}) \tag{5.2}$$

Como discutido na Seção 3.2.1, não é possível obter computacionalmente um estado global de um sistema distribuído, instantaneamente no tempo real, e justamente por isso pode-se questionar sob aspectos práticos e semânticos, a definição original apresentada por Jefferson. Por outro lado, é possível obter eficientemente estados globais através do algoritmo de *snapshots* de Chandy e Lamport [28] (ver Figura 5.8). Através deste algoritmo pode-se obter registros de estados globais concomitantemente com uma computação principal executada sobre o sistema distribuído. Os estados globais, ou *snapshots*, são registrados de forma distribuída, ou seja, cada processador fica responsável pela gravação de uma porção do todo. A composição do *snapshot* é realizada em uma etapa posterior ao registro e pode ser efetuada de diversas maneiras.

Para calcular um valor de GVT a partir da equação de definição apresentada acima, os dados de entrada requisitados resumem-se às informações contidas em um estado global. Seguindo esta linha de raciocínio não é difícil perceber uma forma bastante simples de calcular $GVTs$:

1. Usando o algoritmo de Chandy e Lamport registra-se um estado global do sistema. Cada processador registra dois valores: um estado local, representado pelo valor mínimo de tempo virtual entre os relógios dos processos lógicos que executa e o estado dos canais de entrada do processador, representado pelo menor valor de tempo virtual associado às mensagem em trânsito.

2. Quando o algoritmo termina localmente em um processador, o mínimo entre os valores registrados é calculado e enviado a um processador escolhido arbitrariamente. Todos os processadores remetem seus mínimos locais a um mesmo destinatário encarregado de calcular o GVT.

3. O processador escolhido para calcular o GVT, ao terminar de registrar seu mínimo local e receber os mínimos locais de todos os outros processadores, computa o mínimo entre todos estes valores. O valor obtido é o GVT associado ao estado global registrado.

4. O processador que computa o GVT atualiza internamente o seu valor e informa a todos os outros processadores o valor calculado.

5. Os demais processadores ao receberem o novo GVT, atualizam seus valores internos e o algoritmo termina.

Neste ponto é necessário ressaltar um fato: o valor de mínimo local definido na Equação 5.1 considera mensagens em trânsito nos canais de saída de cada processador e o algoritmo de *snapshots* apresentado permite que cada processador registre as mensagens em trânsito nos seus canais de entrada. Esta diferença entre a definição formal de mínimo local e o valor calculado pelo algoritmo não acarreta, no entanto, erro algum no valor final de GVT computado.

Para um nó p que inicia o espontaneamente:

```
-- registre o valor mínimo de relógio virtual neste processador;
-- antes de enviar qualquer outra mensagem, envie um token
   por cada canal orientado para fora de p;
no.de.tokens := 0
WHILE (no.de.tokens < NUMERO.DE.PROCESSADORES)
  ALT
    from.comm ? token
      -- registre o estado do canal que vem do processador que remeteu
         o token como o valor mínimo de timestamp dentre a seqüência
         de mensagens recebida desde a gravação do estado de p
```

Para um nó p que inicia ao receber um token de um outro nó q

```
no.de.tokens := 0
WHILE (no.de.tokens < NO.DE.PROCESSADORES)
  ALT
    (no.de.tokens = 0) & from.comm ? token
      -- registre o valor mínimo de relógio virtual neste processador;
      -- envie um token por cada canal orientado para fora de p;
      -- grave o estado do canal que vem do processador que remeteu
         o token como sendo o tempo virtual +∞;
    (no.de.tokens > 0) & from.comm ? token
      -- grave o estado do canal que vem do processador que remeteu
         o token como sendo o valor mínimo de timestamp dentre a
         seqüência de mensagens recebida desde a gravação do estado
         de p
```

Observação: entenda-se aqui por *timestamp* o tempo virtual de recepção para mensagens recebidas diretamente do remetente e o tempo virtual de envio para mensagens devolvidas pelo destinatário.

Figura 5.8: Algoritmo para registro de estados globais do *Time Warp*

Um valor correto de GVT é o mínimo entre os valores de relógios virtuais de todo o sistema e os valores de *timestamp* das mensagens em trânsito por todos os canais do sistema. Esteja o estado de um canal incluído no cálculo de mínimo local de um ou outro processador, nenhuma diferença é produzida sobre o valor final do GVT. Importante é que os estados de todos os canais sejam registrados e que sejam incorporados em algum elemento do conjunto de valores remetidos ao processador que realiza a etapa final do cálculo computando a Equação 5.2.

Para completar a definição do algoritmo falta ainda definir seus iniciadores. Existem duas situações diferentes que requerem o cálculo de um novo valor de GVT: esgotamento da memória disponível ou detecção de terminação. Na primeira hipótese, o problema é resolvido através de *Coleta de Fósseis* ou da aplicação de um algoritmo de controle de fluxo, neste caso o protocolo *Cancelback*. Na segunda hipótese, um processador ao completar um determinado período de tempo sem eventos para escalonar pode assumir que a simulação terminou globalmente e disparar o cálculo do GVT. Caso o valor calculado seja $GVT = +\infty$ a simulação estará terminada e cada processador tomará as medidas necessárias para o encerramento do processamento.

Ambas condições de exceção são deteminadas localmente por cada processador e, portanto, podem ser atingidas simultaneamente por vários processadores. Assumindo que um processador inicia o cálculo de um novo GVT tão cedo quanto julgue necessário, o algoritmo que computa este valor deve suportar múltiplos iniciadores. O algoritmo de Chandy e Lamport permite que mais de um processador inicie a execução de um mesmo *snapshot*, mas requer uma implementação um pouco mais sofisticada.

Seja como for, uma implementação qualquer deste algoritmo computa, para cada processador, o estado local e o estado de seus canais de entrada. O protocolo de *Cancelback*, descrito a seguir, manipula um valor que semanticamente é local a cada processador, mas que em um sistema distribuído não pode ser computado a partir de dados locais. Este valor é chamado de *PVT (Processor Virtual Time)* e segue a definição abaixo, que corresponde ao valor ML_p usado na formalização do valor de GVT.

Definição 5.1 PVT. *Contribuição local de um processador ao* GVT*; calculado como o mínimo entre:*

- *o mínimo valor de relógio virtual dos processos lógicos locais;*
- *o mínimo valor de* rvt *das mensagens em trânsito enviadas por processos neste processador para processos em outros processadores;*
- *o mínimo valor de* svt *das mensagens em trânsito devolvidas por outros processadores a um processo remetente alocado neste processador.*

Um processador qualquer tem sempre acesso imediato às suas variáveis locais e portanto o primeiro fator que compõe o PVT pode ser facillmente calculado. Os valores de *timestamp* de mensagens em trânsito, no entanto, representam ao cálculo do PVT a mesma dificuldade que oferecem ao cálculo do GVT. No cálculo de GVT nada importa

se o estado de um canal é registrado em um ou outro processador se no final da computação todos os estados de canais do sistema tiverem sido incluídos no cálculo do mínimo global. Para calcular o PVT de um processador, podemos empregar o mesmo algoritmo de *snapshot* usado na computação do GVT, para registrar as mensagens em trânsito no sistema, fazendo uma pequena adaptação. Como os estado dos canais de saída de um processador são registrados por seus vizinhos, se fizermos que estes remetam os valores obtidos para o processador que deseja calcular seu PVT, este terá a seu dispor todos os valores necessários e poderá efetuar o cálculo.

A situação que requer o cálculo de um PVT, convenientemente, coincide com um tipo de situação que exige o cálculo de GVT, falta de memória. O problema de falta de memória, para ser resolvido eficientemente, necessita de ambos estes valores, que devem ser calculados sobre um mesmo estado global. Esta situação difere do cálculo de GVT na suspeita de terminação, em que não existe necessidade de cálculo de PVT.

Para cada um destes dois casos foi criado um modo específico de iniciar *snapshots* usando tipos diferentes de *tokens*. No caso de terminação, um processador inicia um *snapshot* usando *tokens* marcados com uma constante T e os processadores que recebem estes executam o algoritmo original. Quando um processador inicia um *snapshot* por falta de memória, emprega *tokens* marcados com outra constante C. Os vizinhos do iniciador ao receberem *tokens* assim marcados iniciam o localmente o algoritmo de *snapshot*, caso ainda não estejam participando da computação, passando adiante novos *tokens* marcados com T, e não C.

Um processador que recebe um *token* tipo C por um determinado canal, fica incumbido de enviar ao remetente do *token* o estado do canal assim que terminar de registrá-lo. A computação de PVT termina quando o iniciador recebe de cada um de seus vizinhos o valor mínimo de *timestamp* das mensagens em trânsito, nos seus canais de saída, registradas no *snapshot*. Desta forma, em uma única execução do algoritmo de registro de estados globais calculam-se os dois valores usados no gerenciamento de memória e controle de fluxo.

5.3.3 Uma Adaptação do Protocolo *Cancelback*

Controle de fluxo e gerenciamento de memória são dois aspectos de um mesmo problema em um sistema que executa o mecanismo *Time Warp*. O mecanismo de *rollback* exige que uma quantidade potencialmente muito grande de informações seja armazenada e em determinados momentos um processador não pode prosseguir com a simulação por falta de memória. O protocolo *Cancelback* [5] apresenta-se como uma solução única para os problemas relacionados com a recuperação de espaços de memória e permite que uma simulação possa continuar a executar mesmo em situações extremas.

Existem três tipos de estruturas de dados, alocadas dinamicamente, que à medida que a simulação evolui vão consumindo a memória de um processador:

- mensagens nas filas de entrada,

- mensagens nas filas de saída e
- estados.

Baseando-se na definição do primeiro mecanismo de controle de fluxo, *Message Sendback*, o protocolo *Cancelback* determina uma forma unificada de tratamento para estes diferentes tipos de dados.

O conceito fundamental sobre o qual é construído o protocolo identifica uma forte semelhança entre desbalanceamentos de fluxo de mensagens, que provocam um constante aumento nas filas, e o excesso de itens nas filas de estados. Ambas as situações são decorrentes de diferenças brutais nas taxas de processamento dos processos lógicos e podem ser contornadas através de uma estratégia de minimização de diferenças. A seguir vemos casos que ilustram estas situações.

Dois processos lógicos, A e B, alocados em processadores distintos possuem taxas de processamento muito diferentes que determinam uma grande desnível entre os tempos virtuais marcados por seus relógios. Digamos que o processo A avança mais rapidamente que o processo B e portanto seu relógio virtual está adiantado em relação ao de B. Se A produz um fluxo constante de mensagens em direção a B que não pode processá-las na mesma taxa, as fila de entrada de B tende a crescer indefinidamente. Mesmo que periodicamente execute-se o procedimento de *coleta de fósseis*, o espaço de memória ocupado pelas mensagens de A no processador que executa B não pode ser recuperado. Estas mensagens estarão sempre no futuro virtual de B e portanto terão tempos de recepção superiores ao GVT. Eventualmente o espaço de memória no processador de B esgota-se e a simulação não pode prosseguir.

A situação no processador de A pode, da mesma forma, tornar-se crítica. Como o processo A avança muito rapidamente, a taxa com que itens de memória são alocados para armazenar seus estados tende a ser superior a taxa com que memória pode ser liberada pelo procedimento de *coleta de fósseis*. O mesmo ocorre com as anti-cópias das mensagens que são enviadas para B que fazem a fila de saída de A crescer mais e mais. Novamente, encontramos uma situação em que os itens que estão consumindo mais memória não podem ser descartados por estarem associados a tempos virtuais superiores ao GVT da simulação e que pode levar ao término prematuro da simulação.

O protocolo *Cancelback* apresenta uma solução elegante, além de eficiente, para todos estes problemas. Situações em que a simulação teria de ser abortada são minimizadas através do *cancelamento* de itens nas filas dos processos que evoluem a taxas muito superiores às dos outros. *Cancelback* age sempre no sentido de retardar os processos que estão por demais à frente dos outros no tempo virtual tentando balancear taxas de processamento, controlando o fluxo de mensagens ao mesmo tempo em que faz gerenciamento de memória. Além disso, como Jefferson prova em seu artigo, *Cancelback* permite aproveitamento máximo de memória chegando ao ponto tornar possível executar simulações sob o *Time Warp* ocupando o mesmo espaço requerido por uma simulação seqüencial.

Antes de apresentar o algoritmo do protocolo, vejamos como *Cancelback* resolve problemas como os ilustrados acima no exemplo dos processos A e B. Digamos que

estes processos estão alocados em processadores P_i e P_j, respectivamente, juntamente com vários outros processos lógicos em cada processador. Quando a quantidade de memória disponível em P_j desce abaixo de um limite pré-estabelecido, o protocolo *Cancelback* é ativado e primeiramente tenta liberar espaço através de *coleta de fósseis*. Caso isso não seja possível, para que a simulação tenha condições de prosseguir, o protocolo inicia uma busca nas filas de todos os processos até encontrar o elemento sobre o qual pode aplicar uma operação de *cancelamento* provocando o menor prejuízo possível para o progresso da computação. O critério para escolha deste elemento é bastante simples: busca-se o item que possui maior tempo virtual associado superior ao PVT do processador. Este critério de seleção permite que se descubra o processo lógico mais a frente no tempo virtual. O item de memória encontrado pode ser uma mensagem em uma fila de entrada ou em uma fila de saída ou ainda um estado, obviamente, em uma fila de estados.

A operação de *cancelamento* pode assumir formas diferentes depedendo do tipo do elemento sobre o qual se aplica:

- se o item for uma mensagem em uma fila de entrada, este é removido da fila, e devolvido ao remetente com campo de sinal invertido. Ao retornar ao processador de onde partiu, a mensagem é inserida na fila de saída do processo que a enviou cancelando a anti-cópia ali presente e causando *rollback* para um tempo anterior ao de seu envio. Após o *rollback*, o processo volta a avançar no tempo e eventualmente re-gera a mesma mensagem. Este é o caso em que a falta de memória é causada por processos receberem mensgens mais rápido do que podem processar.

- se o item for uma anti-cópia de mensagem em um fila de saída, este é removido da fila e enviado para o mesmo destino da mensagem original cancelando-a na fila de entrada do receptor. O processo que enviou a mensagem, e depois a anti-cópia, é forçado a realizar um *rollback* para um tempo anterior ao da geração da mensagem original e quando volta a avançar gera novamente a mesma mensagem e a envia ao mesmo destinatário. No processo que recebe a anti-cópia acontece o mesmo procedimento de uma aniquilação normal de mensagens: caso a mensagem original já tenha sido processada, o processo sofre *rollback* e depois volta a avançar como se jamais a tivesse recebido.

- se o item for um estado, o processo a que pertence é forçado a realizar um *rollback* para o estado anterior ao escolhido e o estado é descartado.

Podemos verificar que a operação de cancelamento sempre libera, no mínimo, o espaço de um elemento das filas e é implementada de forma a harmonizar-se com os mecanismos pré-existentes no *Time Warp*. Diferente de outros mecanismos de controle de fluxo que agem devolvendo as mensagens que excedem a capacidade de memória do receptor, transferindo o problema de alocação de espaço de um processador para outro, o *Cancelback* libera espaço em ambos os processadores. Além disso, as mensagens devolvidas não são armazenadas temporariamente no remetente que tentaria enviá-las em um instante posterior, mas cancelam itens em sua fila de saída e obrigam o processo a retroceder no tempo virtual e recriá-las quando volta avançar.

O critério de escolha do item a ser cancelado permite que se penalize sempre o processo mais adiantado no tempo virtual, o que não deve produzir efeitos negativos sobre a simulação porque justamente por estar avançando a uma taxa de processamento muito grande causa problemas na utilização de memória. Se assumirmos que a memória é dividida em páginas de tamanho fixo e que estados ou mensagens ocupam sempre o espaço de uma página, então pode-se provar que este critério produz resultados ótimos.

Para que se aplique este protocolo exigem-se algumas condições básicas oferecidas por esta implementação de *Time Warp*: meio de comunicação confiável entre processadores, estados completos registrados ao fim de cada evento processado, a não existência de mensagens com atraso 0 (*rvt=svt*). Jefferson observa que em sua definição do protocolo não admite a existência de eventos de múltiplas mensagens (mensagens para um mesmo destinatário estampadas com um mesmo *rvt*). Esta restrição é, no entanto, muito severa e impõe um sérias dificuldades à programação de simulações. Neste trabalho, não assumimos esta postura e permitimos este tipo de evento, o que causa algumas dificuldades de implementação mas que facilita em muito o desenvolvimento de aplicações.

A implementação deste protocolo requer sérios cuidados. Suponhamos que no momento em que um processador está devolvendo uma mensagem a outro processador, o remetente sofre *rollback* e envia para o destinatário uma anti-cópia da mensagem que está sendo devolvida. Algumas variantes deste tipo de situação podem ocorrer, devido uso do *Cancelback*, e permitir que mensagem e anti-mensagem se cruzem sem aniquilação. Como isso violaria regras básicas de operação do *Time Warp* comprometendo a correção da simulação, algum tipo de protocolo para evitar estas occorências deve ser implementado em um nível inferior ao do *Cancelback*.

Para esta implementação foi utilizado um protocolo simples que garante que mensagens e anti-mensagens recebam sempre o tratamento adequado em todas as situações possíveis. Existem algumas situações a considerar para garantir a aniquilação de um par de mensagens de sinais opostos, que podem ser descritas por três valores booleanos embutidos no cabeçalho de qualquer mensagem: *direção*, *rollback* e *cancelback*. Quando *direção* = `TRUE`, a mensagem está sendo enviada do remetente para o destinatário em uma direção dita *direta* e em caso contrário a mensagem está sendo devolvida pelo destinatário e trafega em uma direção dita *reversa*. O campo *rollback* indica que a mensagem está sendo enviada para cancelar uma computação incorreta quando contém o valor `TRUE` ou que é uma mensagem normal da simulação, quando contém `FALSE`. O campo *cancelback* contém o valor `TRUE` quando a mensagem está sendo enviada pelo protocolo *Cancelback* ou `FALSE` em caso contrário.

Com estes três campos é possível identificar oito combinações de valores das quais somente quatro são relevantes e indicam como a mensagem deve ser tratada:

1. *direção* = *direta*, *rollback* = `FALSE`, *cancelback* = `FALSE`;
   Esta é uma mensagem normal da simulação e não requer nenhuma forma especial de tratamento.

2. *direção* = *direta*, *rollback* = `FALSE`, *cancelback* = `TRUE`;
   Esta é uma anti-mensagem enviada na direção direta pelo protocolo *Cancelback* no

processador do remetente. Ao chegar ao destinatário, esta mensagem causa uma busca na fila de entrada para verificar se ali está sua anti-cópia. Caso a anti-cópia seja encontrada ambas são descartadas como em uma aniquilação normal. Caso contrário, a anti-cópia foi devolvida ao remetente pelo *Cancelback* no processador do destinatário e a mensagem recém recebida deve ser descartada. A anti-mensagem devolvida será tratada convenientemente no remetente e o efeito aparente será o de uma aniquilação normal.

3. *direção* = *direta*, *rollback* = `TRUE`, *cancelback* = `FALSE`;
   Esta é uma anti-mensagem enviada pelo mecanismo de *rollback* para cancelar os efeitos colaterais de uma computação que se descobriu ser incorreta. Uma busca na fila de entrada do destinatário é realizada e caso sua anti-cópia seja encontradas ambas são descartadas. Caso contrário, a anti-cópia foi devolvida ao remetente pelo *Cancelback* no processador do destinatário e a mensagem recém recebida deve ser descartada. A anti-mensagem devolvida será tratada convenientemente no remetente e o efeito aparente será o de uma aniquilação normal.

4. *direção* = *reversa*, *rollback* = `FALSE`, *cancelback* = `TRUE`;
   Esta é uma mensagem enviada pelo protocolo *Cancelback* no processador do destinatário. Uma busca por sua anti-cópia é realizada na fila de saída do remetente e caso seja encontrada ambas são descartadas. Caso contrário, a anti-cópia foi enviada na direção direta pelo protocolo *Cancelback* no remetente e deve ser descartada para simular uma aniquilação normal.

A implementação para máquinas de distribuída, empregada nestre trabalho, é apresentada na Figura 5.9. Esta implementação difere da proposta original em alguns aspectos. No algoritmo de Jefferson, o protocolo é ativado sempre que a quantidade de páginas de memória livre chega a zero e entra em ciclo terminando quando pelo menos uma página tiver sido liberada. Este limite inferior de páginas é definido em função da quantidade de memória necessária para armazenar um novo item, estado ou mensagem. Como nesta implementação uma mensagem está sempre associada a um registro de causalidade (que também ocupa uma página de memória), no pior caso, o espaço necessário para armazenar um novo item será duas páginas e por isso o limite para ativação do protocolo foi alterado.

A diferença mais importante está no modo de ativação do protocolo. Ao término de toda operação que aloca uma página de memória no processador, o procedimento `PreCancelback()` é invocado. Este procedimento verifica se o número de páginas livres é inferior ao limite pré-definido, invocando a atualização dos valores de GVT e PVT através da chamada a `Compute.GVT(cancelback)`. Esta chamada inicia a computação de um *snapshot* usando *tokens* tipo C, que determinam que ao término do algoritmo o valor de PVT terá sido atualizado. Quando o algoritmo de *snapshot* termina, o valor da variável `cancelback.on` é verificado e, caso seja `TRUE`, o procedimento `Cancelback()` é invocado. Este procedimento realiza as operações do protocolo descrito acima.

O procedimento `Cancelback()` implementa o protocolo e consiste basicamente de um ciclo que é executado até que o número de páginas livres seja no mínimo igual ao limite ou que se determine que deve ser interrompido para que uma outra computação seja efetuada. No ciclo do protocolo, inicia-se buscando o estado ou mensagem com maior

```
PROC Cancelback()
  SEQ
    escape.cancelback := FALSE
    WHILE (livres < 2) AND (NOT(escape.cancelback))
      SEQ
        ...  Busca item para ser cancelado
        IF
          (item.svt > pvt)
            -- retira item da fila e cancela
            CASE item.fila
              fila de entrada
                ...  devolve mensagem ao remetente e descarta item
              fila de saída
                ...  envia anti-mensagem ao destinatário, descarta item
                  e executa rollback
              fila de estados
                ...  descarta e estado e executa rollback
          TRUE
            IF
              (gvt < pvt)
                SEQ
                  Compute.GVT(cancelback)
                  escape.cancelback := TRUE
              TRUE
                  -- aborta simulação:  falta de memória
                  STOP



PROC PreCancelback()
  SEQ
    IF
      ((livres < 2) AND (NOT(cancelback.on)))
        SEQ
          cancelback.on := TRUE
          Compute.GVT(cancelback)
      TRUE
        SKIP
```

Figura 5.9: Algoritmo do protocolo *Cancelback*

tempo virtual associado em todas as filas do processador. Se o item encontrado estiver marcado com um tempo virtual superior ao valor corrente de PVT então pode ser cancelado conforme descrito acima. Se um elemento que satisfaça a esta condição não for encontrado, então existem duas alternativas:

- se o valor atual de GVT for inferior ao de PVT então é possível que esteja desatualizado. O ciclo deve ser interrompido e o procedimento terminado para que se possa iniciar a computação do GVT. Calculado um novo valor, algumas páginas de memória podem vir a ser liberadas através de *coleta de fósseis* e a simulação prossegue normalmente.
- se o valor de PVT for superior ao de GVT então não há mais chances de liberar memória para que a simulação prossiga neste processador e o sistema todo deve ser abortado.

Capítulo 6

Avaliação Experimental

Na avaliação de desempenho de um mecanismo de simulação duas abordagens podem ser seguidas. A alternativa mais comum, porém mais trabalhosa e demorada, consiste da execução de diversas aplicações reais sobre o mecanismo atentando para os parâmetros indicadores de seu desempenho. Diferentes aplicações podem exercitar aspectos diferentes do mecanismo e, portanto, tão mais precisa será a análise do seu desempenho quanto maior for a quantidade de aplicações empregadas nos ensaios.

Uma outra alternativa é a utilização de uma *carga sintética*, ou seja, uma aplicação fictícia que é a realização de um modelo parametrizado de avaliação de desempenho. Com a execução de um único modelo, facilmente construído, torna-se possível, através da variação de seus parâmetros, obter resultados que permitam prever o desempenho da execução de uma série de aplicações sobre o mecanismo avaliado. Assim, ao invés de despender tempo e esforço na programação de múltiplas aplicações pode-se, apenas, produzir conjuntos de parâmetros que reflitam características de aplicações reais.

Neste capítulo, apresentamos um modelo de carga sintética que possibilita a realização de experimentos profundos com o mecanismo Time Warp. Em seguida, expomos os resultados obtidos com o uso deste modelo tentando relacioná-los com aspectos de simulações reais de eventos discretos.

6.1 O modelo de avaliação de desempenho

Empregamos neste estudo o modelo PHOLD, construído por Fujimoto [8] como uma extensão do modelo *hold*, introduzido por Jones [31] e usado na avaliação de estruturas de dados empregadas em simuladores seqüenciais.

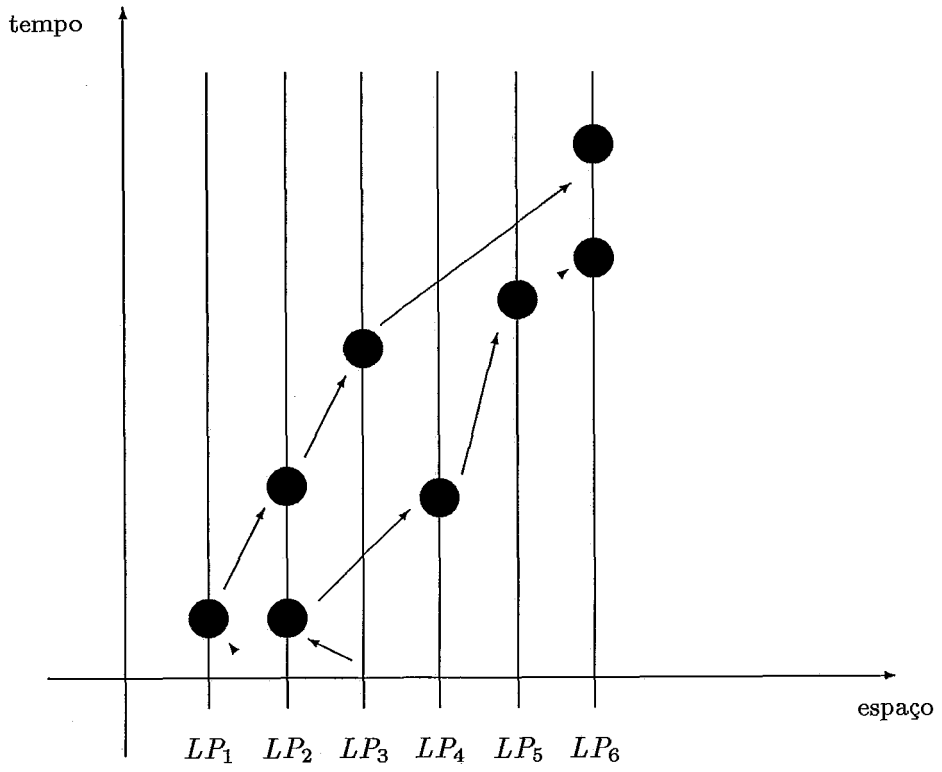

Figura 6.1: Exemplo de um diagrama espaço–tempo

Para descrever o modelo e seu comportamento usaremos um recurso gráfico comumente empregado para ilustrar o comportamento de uma simulação qualquer: o diagrama *espaço–tempo*. Este diagrama consiste de um plano definido pelo eixo espaço, de valores discretos onde cada ponto representa um processo lógico, e pelo eixo de tempo, de valores contínuos dados pelo tempo simulado.

Em um diagrama como da Figura 6.1, as linhas verticais representam a evolução de um processo lógico ao longo do tempo e cada círculo sobre estas linhas representa a ocorrência de um evento. Cada arco representa o escalonamento de um evento em conseqüencia do processamento de outro; um arco partindo de um ponto (s, T_s) para um ponto (r, T_r) indica que um evento ocorrido no processo lógico s em um tempo T_s causa o envio de uma mensagem para r a ser processada em um tempo T_r, onde $T_r > T_s$.

O modelo PHOLD assume que o processamento de em evento resulta no escalonamento de, exatamente, um outro evento. Isto corresponde a dizer que seu diagrama espaço–tempo é formado por um número fixo de seqüências de eventos, ou *threads*, começando

nos eventos escalonados no início da simulação.

As diferenças entre os os tempos de envio e recepção de cada evento escalonado determinam a localidade temporal da simulação, ou seja, como se distribuem os eventos ao longo do tempo simulado. Analogamente, a movimentação de uma *thread* de um processo lógico para outro determina a localidade espacial da simulação. Simulações diferentes possuem maneiras próprias de calcular incrementos ou de determinar o recipiente de uma mensagem.

Para representar uma simulação genérica, o modelo PHOLD contém uma série de parâmetros que permitem definir comportamentos bastante diversos em sua execução:

número de processos lógicos; oferece um limite superior para a quantidade de paralelismo disponível;

população de mensagens; que indica a quantidade de seqüências causais, ou *threads*, existente no diagrama espaço–tempo;

função de incremento de timestamp; determina a diferença entre tempo de envio e tempo de recepção para eventos criados por cada processo lógico;

função de movimento; determina o processo lógico j, destino de um evento processado no processo lógico i;

granularidade; representa o tempo necessário ao processamento de um evento;

configuração inicial; define a distribuição inicial dos eventos da simulação.

6.1.1 Localidade Espacial

A função de movimento define a localidade espacial da simulação e através deste parâmetro é possível construir as mais diversas topologias para a rede de processos lógicos. Cada processo nesta rede possui um conjunto de processos mais próximos que é denominado *vizinhança*. O tamanho deste conjunto controla o grau de localidade espacial exibido pela simulação.

Como dito anteriormente, a cada evento processado corresponde a criação de um novo evento escalonado em um outro processo. A seleção do processo para onde se destina o novo evento é feita através da função de movimento, que escolhe o recipiente dentro da vizinhança do remetente. Todos os elementos dentro de uma vizinhança possuem a mesma probabilidade de serem selecionados.

Os ensaios realizados por Fujimoto [8], utilizam redes que variam desde uma topologia toroidal (Figura 6.2) até um grafo completo. Na rede toroidal, um processo lógico qualquer possui como vizinhos apenas os quatro processos lógicos adjacentes. No grafo completo, um processo lógico possui como vizinhos todos os outros processos da rede. É fácil verificar que estes dois casos representam extremos de localidade espacial; a rede toroidal possui a máxima localidade enquanto o grafo completo a mínima.

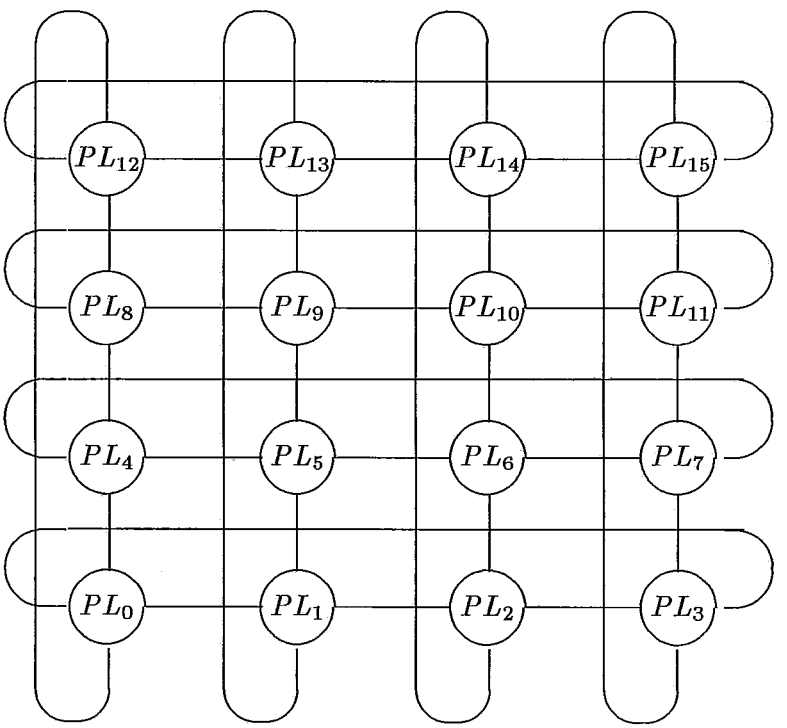

Figura 6.2: Rede toroidal usada na avaliação de desempenho

Nestes experimentos, ficou comprovado que quanto maior localidade espacial exibe uma aplicação, maior *speed-up* pode ser conseguido. Neste trabalho, não variamos continuamente a localidade espacial analisando seu efeito sobre *speed-up* ou eficiência. Os experimentos aqui realizados consideram apenas os casos extremos deste parâmetro. Como vimos, o novo esquema de cancelamento de mensagens aqui proposto, permite alcançar maiores *speed-ups* em aplicações de alta localidade espacial por minimizar não só os custos de comunicação entre processadores. O esquema de *Cancelamento Direto Local* minimiza também os custos inerentes aos ajustes de filas nas aniquilações entre mensagem e anti–mensagem, além de reduzir o tempo total envolvido na correção de erros de causalidade.

A rede escolhida para avaliar o desempenho desta versão de *Time Warp*, difere das redes de Fujimoto, mas busca submeter o sistema a uma situação de pior caso e analisar seu comportamento. Definiram-se dois anéis interligados com 8 processos cada um, alocando 2 processos a cada processador do hipercubo. O grau de localidade espacial é razoavelmente grande e através da uma distribuição inicial de mensagens nos anéis é possível criar situações que levam a ocorrência de um grande número de *rollbacks*.

6.1.2 Granularidade

Variando a granularidade de eventos, Fujimoto mostra que a sensibilidade do mecanismo Time Warp em relação a este parâmetro é da ordem de 10% para valores de eficiência e de 5% para valores de *speed-up*. Sabemos que a granularidade de uma aplicação é definida como a razão entre o tempo despendido em processamento e o tempo despendido em comunicação e sincronização. O artigo de apresenta uma análise de sensibilidade sem, no entanto, indicar medidas dos custos de comunicação e sincronização da arquitetura. Por isso, não é simples tentar inferir os resultados de experimentos semelhantes com outras arquiteturas a partir dos dados apresentados.

Na discussão do impacto da granularidade sobre uma computação paralela, Stone [46] propõe a seguinte questão:

> Dado um sistema com N processadores e uma razão R/C que reflita a taxa alcançável de tempo de execução por unidades de custo de comunicação, como pode uma aplicação ser particionada e executada na arquitetura multiprocessada, fazendo o melhor uso possível de seus recursos ?

O desempenho do Time Warp em uma arquitetura como o hipercubo de *transputers*, bastante diversa da arquitetura do BBN Butterfly dos testes de Fujimoto, pode apresentar sensibilidades diferentes em relação à granularidade. Não é possível, portanto, prever o comportamento desta versão de Time Warp por simples comparação. Não se dispõe de dados como o valor de R/C para o Butterfly, ou mesmo para o hipercubo de *transputers*, nem da dismensão dos dados comunicados em uma mensagem nos experimentos mencionados. Para reavaliar, então, o comportamento do mecanismo nesta arquitetura de memória distribuída, novos testes devem ser realizados.

6.1.3 Localidade Temporal

Um programa executando sobre um sistema de memória virtual, para sofrer degradação de desempenho, deve concentrar os acessos que faz à memória em uma região limitada e bem definida para minimizar a quantidade de *page faults*. Um programa com padrão de acessos bem definido exibe uma propriedade que se convencionou chamar de *localidade*.

Jefferson [1] enuncia uma propriedade semelhante a ser aplicada a sistemas de tempo virtual, que também é fator determinante de desempenho. A propriedade de *localidade temporal* é exibida por um programa que executa sobre um sistema de tempo virtual em que os processos tendem a enviar mensagens com tempos de recepção que não estejam no passado dos processos a que se destinam. Portanto, quanto maiores forem os incrementos nos tempos virtuais de recepção calculados por uma aplicação, maior a possibilidade destas mensagens chegarem no futuro virtual de seus destinatários não causando *rollback*. A ocorrência infreqüente de *rollbacks* evita a degradação de desempenho do sistema.

No modelo PHOLD, podemos verificar que a função de incremento de *timestamp* tem efeito direto sobre a localidade temporal da simulação. Nas experiências de Fujimoto, diversas funções foram analisadas na determinação do tempo virtual de recepção (rvt) de uma mensagem a partir do valor de relógio virtual local de um processo. As funções apresentadas a seguir, com exceção da *determinística*, empregam uma função para geração de números aleatórios, *rand*, que retorna um valor uniformemente distribuído entre 0 e 1.

Definição 6.1 *Determinística.* $\delta rvt = 1.0$ □

Definição 6.2 *Polarizada.* $\delta rvt = 0.9 + 0.2\, rand$ □

Definição 6.3 *Uniforme.* $\delta rvt = rand$ □

Definição 6.4 *Exponencial.* $\delta rvt = -ln(rand)$ □

Definição 6.5 *Bimodal.* $\delta rvt = 0.95238\, rand +$ **if** $rand \leq 0.1$ **then** 9.5238 *else* 0 □

Os resultados obtidos em testes com cada uma destas funções reforça a hipótese de Jefferson de que melhores localidades temporais, em uma simulação sobre o Time Warp, levam a melhores desempenhos. Entre as funções testadas, a determinística e a polarizada, mostraram-se as mais eficientes por exibirem as melhores localidades temporais. Como os incrementos de *timestamp* calculados com estas funções tendem a projetar os tempos de recepção das mensagens no futuro dos processos receptores, os *rollbacks* tendem a diminuir.

6.1.4 Lookahead

Na criação de novos eventos, muitas vezes os processos lógicos são capazes de prever o futuro. Por exemplo, em uma simulação de redes de filas com *jobs* priorizados com preempção, quando um processo lógico modelando o servidor recebe um *job* de alta prioridade,

pode prever e enviar imediatamente um novo *job* escalonado para um tempo virtual futuro. Esta capacidade de determinar quão prontamente um processo pode escalonar novos eventos é chamada de *lookahead*. Um processo como o servidor descrito acima possui "bom" *lookahead*, pois pode criar um novo evento no futuro virtual assim que recebe o evento que o causa.

Na simulação de uma rede de filas com *jobs* priorizados e com preempção, quando um servidor recebe um *job* de baixa prioridade, não pode enviar um novo *job* até que o tempo de serviço do que recebeu se complete. A qualquer momento durante o tempo de serviço do *job* de baixa prioridade, o servidor poderia receber um *job* de alta prioridade que causaria sua preempção. Neste caso, o processo tem um *lookahead* "ruim" porque não pode prever com precisão os eventos que gera em função do primeiro.

Definição 6.6 Lookahead. *Se um processo, de conhecimento de todos os eventos que recebe até um tempo T, pode predizer os novos eventos que gera até um tempo $T + L$ ou menor, diz-se que possui* lookahead L. □

Segundo Fujimoto [12], *lookahead* é uma função complexa que varia com o tempo e tipo de evento, altamente dependente de detalhes do problema simulado e da maneira como foi programado. Um processo pode enviar um novo evento com *rvt* no futuro, se este valor de *timestamp* for menor ou igual ao valor de seu relógio virtual local somado ao seu *lookahead*.

Para introduzir *lookahead* como um parâmetro do modelo consideremos um evento com $rvt = T_{causa}$, que ao ser processado cria um novo evento com $rvt = T_{efeito}$. A diferença $T_{efeito} - T_{causa}$ representa um incremento de *timestamp* que pode ser considerado como dividido em duas partes: (1) um intervalo de tempo em que o processo não tem capacidade de prever com precisão sua saída, pois eventos que viria a receber como entrada poderiam modificá-la e (2) um intervalo de tempo em que pode prever sua saída independente do que venha a receber como entrada, ou seja, *lookahead*.

Na modelagem de localidade temporal define-se uma função probabilística que retorna valores usados como incremento de *timestamp*, ou seja, valores prontos para a diferença $T_{efeito} - T_{causa}$. Para introduzir *lookahead* no contexto deste modelo, define-se uma relação entre o incremento de *timestamp* e um valor numérico de *lookahead* capaz de indicar quanto um processo deve avançar no tempo virtual antes de enviar o novo evento. Assim, apresentamos abaixo a *Taxa Inversa de Lookahead* como definida por Fujimoto.

Definição 6.7 Taxa Inversa de Lookahead *ou* Inverse Lookahead Ratio (ILAR).

$$ILAR = \frac{lookahead}{T_{efeito} - T_{causa}}$$

□

Esta quantidade representa, em termos percentuais, a relação entre o intervalo de tempo determinado pelo *lookahead* e o intervalo maior determinado pelo incremento de

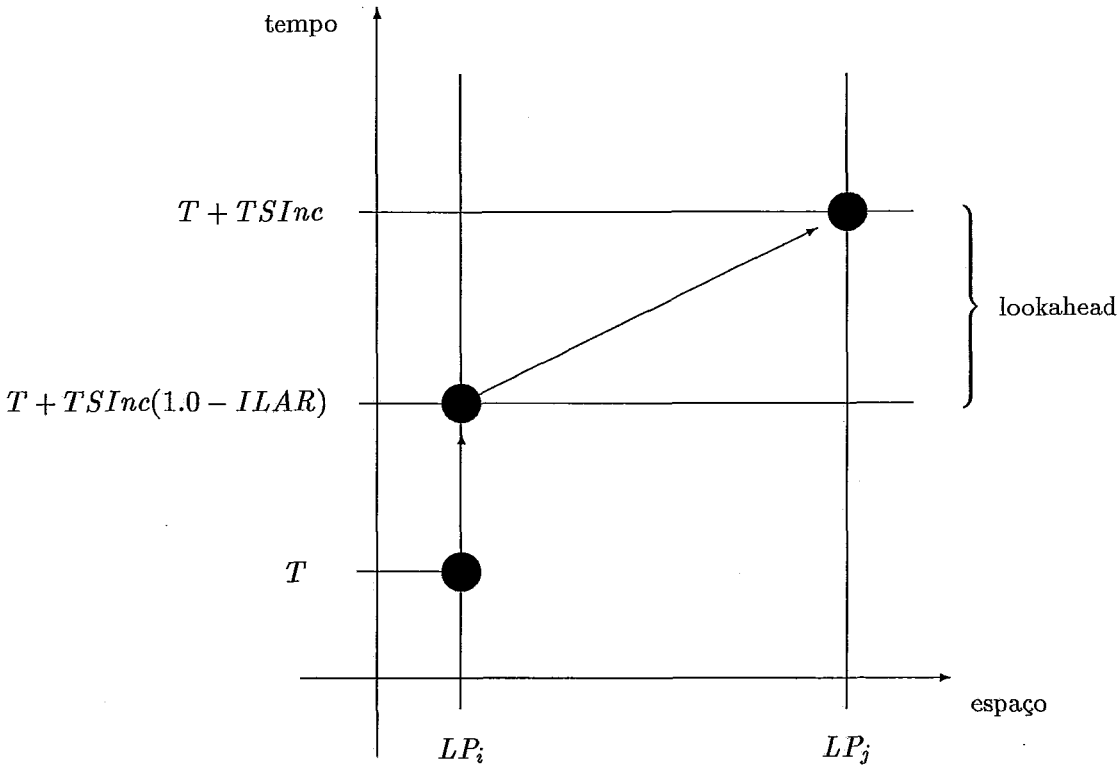

Figura 6.3: Modelagem de lookahead

timestamp. Valores próximos de 1 indicam a dominância do intervalo de tempo determinado pelo *lookahead* dentro do intervalo de tempo que abrange o incremento total. Valores mais p'roximos de 0 indicam um peso maior do intervalo em que predomina a incerteza no escalonamento do novo evento dentro do incremento total.

Arbitrando valores para *ILAR* é possível definir um percentual fixo de *lookahead* dentro dos intervalos variáveis retornados pela função de incremento de *timestamp*. Para cada evento processado, o modelo PHOLD força um processo lógico a enviar um evento intermediário para si mesmo, antes de enviar o evento que continue uma seqüência causal. Como se pode ver na Figura 6.3, para um valor de incremento *TSInc* e de *ILAR*, no processamento de um evento o processo lógico a avança seu relógio virtual até o valor de tempo virtual determinado pela expressão $T + TSInc(1.0 - ILAR)$ que calcula o *rvt* do evento intermediário. Após processar o evento enviado para si próprio, o processo pode escalonar o evento real simulando um valor de *lookahead*.

Com este artifício, o modelo pode criar uma distinção clara entre os intervalos menores de certeza e incerteza contidos nos incrementos de tempo de recepção, calcu-

lados a partir de uma função aleatória, mantendo uma determinada proporção.

Os experimentos de Fujimoto indicam que o mecanismo *Time Warp*, diferentemente dos paradigmas consevadores, apresenta pouca sensibilidade em relação a variações de *lookahead*. A não ser em casos com valores de *ILAR* muito próximos ou iguais a 0, eficiência e *speed-up* apresentam valores dentro de uma estreita faixa de variação.

6.2 Resultados Experimentais

Inicialmente foram realizados testes para determinar se alguns valores arbitrários de tempo de processamento de evento seriam capazes replicar o comportamento de simulações relativamente complexas nesta máquina. Após algumas experiências, chegou-se à conclusão que utilizando tempos fixos de 10ms por evento consegue-se um programa paralelo adequado à granularidade da arquitetura e uma carga sintética computacionalmente pesada para testar o mecanismo de simulação.

Os outros parâmetros do modelo de avaliação de desempenho receberam valores de acordo com o seguinte critério:

- $ILAR = 0.5$; como os experimentos de Fujimoto mostram pouca sensibilidade do *Time Warp* em relação a este parâmetro, arbitrou-se este valor por permitir a modelagem de um *lookahead* considerável levando a simulação a exibir um grau de paralelismo razoável para o número de processos executados;
- *horizonte da simulação* $= 10000$; com a granularidade de eventos selecionada este valor permite a realização de simulações seqüenciais em torno de uma hora de duração para maiores densidades de mensagens por processo lógico;
- *distribuição determinística de incrementos de timestamps*; por permitir grande localidade temporal não induzindo a ocorrência de uma maior freqüência de *rollbacks* que a determinada pela distribuição inicial de mensagens nos anéis .

As distribuições iniciais de eventos pelos processos lógicos foram geradas aleatoriamente para populações de mensagens variando desde 2 até 32, passando por valores intermediários correspondentes ao dobro do valor anterior. A densidade de mensagens correspondente assume valores entre 1/4 e 4 inclusive. Sobre estas distribuições espaciais de mensagens criaram-se duas distribuições de *timestamps*: uma impedindo totalmente a ocorrência de *rollbacks*, com a finalidade de prover bases para avaliação do *overhead* do mecanismo sobre a computação e outra aleatória, com tempos iniciais entre 0 e 1, favorecendo a ocorrência de múltiplos *rollbacks* nas redes em anel.

Nesta versão de *Time Warp*, a estimativa de *GVT* é calculada por demanda e não periodicamente a intervalos pré-determinados. No entanto, é necessário arbitrar um valor de tempo máximo que um processador pode ficar ocioso por não ter eventos a processar. Ao detectar o fim de um intervalo de tempo limitado por este valor, o processador dispara uma computação de *GVT* para verificar se a simulação foi terminada encerrando seus

processos e computando estatísticas caso todo o sistema tenha atingido o tempo virtual $+\infty$. O valor escolhido para este parâmetro exerce forte influência nos resultados da avaliação de desempenho sob dois aspectos:

- quanto maior este limite, maior o tempo entre o término da simulação e a detecção desta condição, podendo extender o tempo medido de execução muito além do tempo efetivamente gasto para completer a computação;
- valores muito pequenos podem permitir uma rápida detecção de terminação, no entanto, causam um aumento no *overhead* do mecanismo por determinar um número de execuções do algoritmo de *GVT* potencialmente bem maior do que o necessário.

Realizando algumas experiências chegou-se a uma solução de compromisso que minimiza *overheads* e permite uma detecção rápida de terminação utilizando um limite de 2s.

Com este conjunto de parâmetros realizaram-se experimentos produzindo resultados apresentados em forma de gráficos nas Figuras 6.4, 6.5 e 6.6. Analisando o gráfico de *speed-up* em 8 processadores em função da população de mensagens apresentado na Figura 6.4, verifica-se que somente para os maiores números de mensagens trafegando nos anéis obtém-se ganhos significativos na simulação paralela, como se poderia prever. Quanto menor a quantidade de eventos processada, menor é o número de cadeias causais e menor o paralelismo inerente à simulação. Para simulações oferecendo maior paralelismo obtiveram-se valores de *speed-up* de 2.00 para o caso de distribuição de eventos iniciais que maximiza a taxa de em rollbacks e de 4.6 para o caso de uma distribuição de eventos inicias evitando a ocorrência de *rollbacks*. Este último valor indica uma eficiência de 57% na simulação, enquanto o pior caso indica uma eficiência de aproximadamente 13%. Estes valores tão díspares podem ser explicados pelo fato de a natureza de cada experimento explorar casos extremos. Espera-se que para aplicações reais que produzam taxas intermediárias de *rollbacks* os resultados situem-se dentro destes limites.

Observando os gráficos nas Figuras 6.5 e 6.6 pode-se verificar que enquanto o tempo de processamento das simulações seqüenciais tende a aumentar muito rapidamente com a população de mensagens, nas simulações paralelas este crescimento é muito mais lento. Espera-se que para valores de número de cadeias causais ainda maiores que os avaliados, que o tempo de processamento continue a aumentar na mesma proporção até que se atinja o limite máximo de paralelismo que esta quantidade de processadores permite explorar. Acima desta limite, espera-se que a curva passe a seguir maiores inclinações no entanto mantendo uma diferença cada vez maior em relação ao caso seqüencial e determinando maiores valores de *speed-up*.

No pior caso, o algoritmo de cálculo de *GVT* juntamente com o procedimento de *coleta de fósseis* foi executado 12 vezes para um horizonte de 10000 unidades de tempo virtual, dispondo de 6250 páginas de memória de 256 bytes. O tempo total gasto nestas atividades foi medido em, aproximadamente, 26s, o que em relação a um tempo de simulação de 680s representa uma fração de 3.8se esgotou em algum processador, um grande número de páginas pode ser liberado através de simples *coleta de fósseis* sem a ativação do protocolo *Cancelback*. Reduzindo a quantidade de páginas em cada processador para 1000, pode-se observar que este protocolo era constantemente requisitado

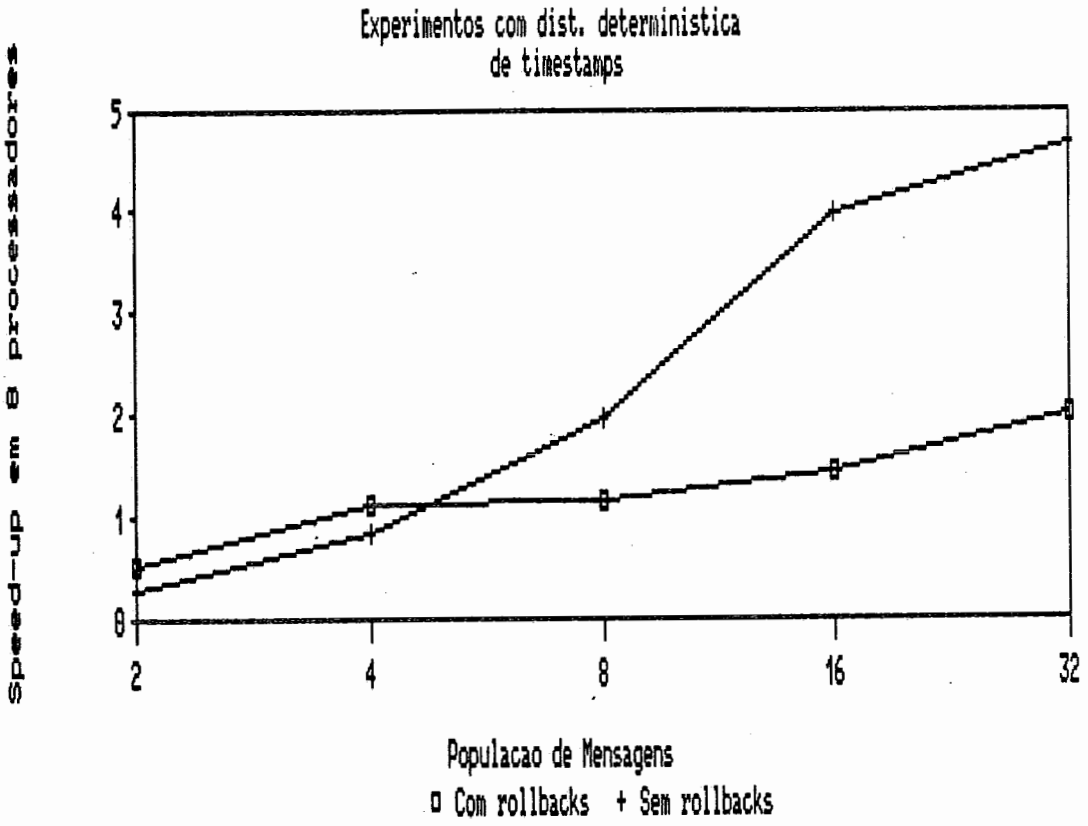

Figura 6.4: *Speed-up* em um hipercubo com 8 processadores

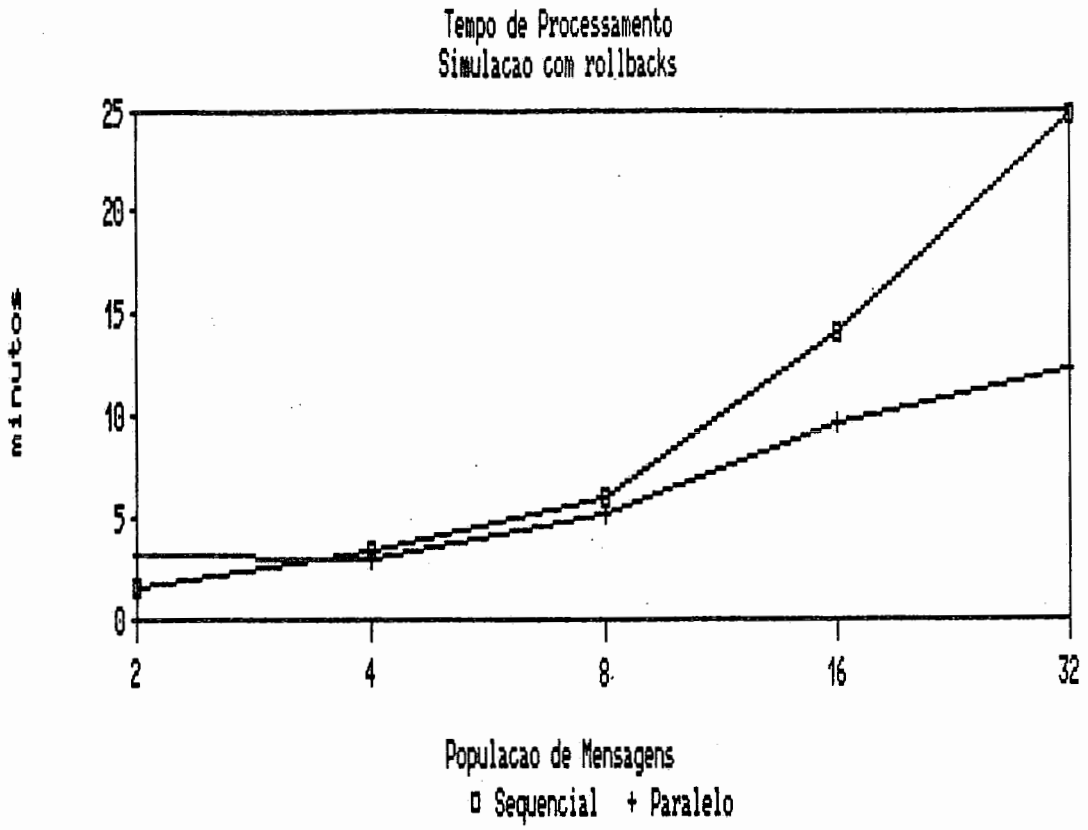

Figura 6.5: Tempo de processamento para simulação sem *rollbacks*

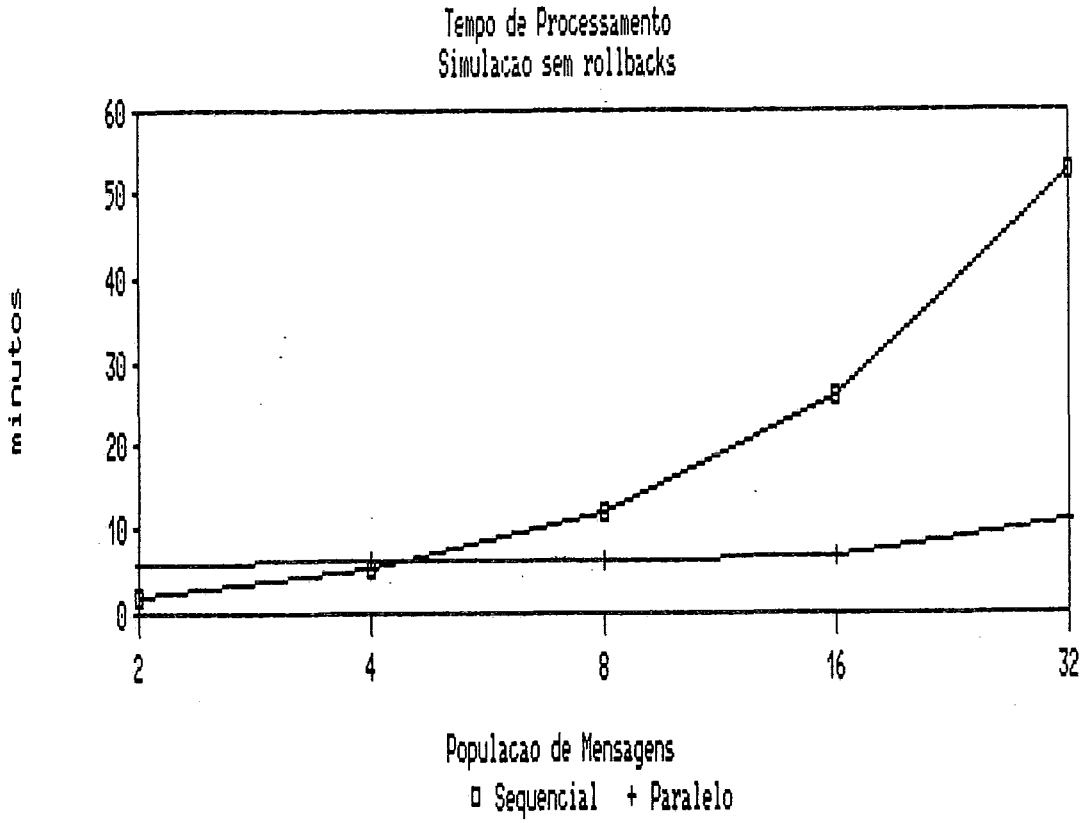

Figura 6.6: Tempo de processamento para simulação com *rollbacks*

degradando absurdamente a taxa de processamento de eventos por segundo até o valor de 2.0. Destes resultados pode-se conlcuir que mesmo com este mecanismo de gerência de memória e controle fluxo, a simulação paralela só pode ser executada eficientemente quando dispõe de um espaço de memória bastante grande. Para as mesmas simulações executadas seqüencialmente, a quantidade de memória necessária foi sempre muito pequena: da ordem do número de cadeias causais simuladas. A afirmação de que com *Cancelback* é possível executar simulações paralelas sob *Time Warp* ocupando a mesma quantidade de memória empregadas nas simulações seqüenciais, enquanto teoricamente verdadeira, representa resultados práticos desastrosos que inviabilizam a sua realização. Para casos menos drásticos de redução de memória pode-se verificar que quando o protocolo não é requisitado repetidamente, a simulação paralela pode avançar sem degradações exageradas de desempenho. Não foram obtidas medidas precisas para estes casos a ponto de permitir alguma quantificação dos resultados.

Capítulo 7

Conclusões

Os resultados obtidos com este trabalho podem ser considerados um ponto de partida no caminho para uma avaliação mais extensa do desempenho do *Time Warp* para um hipercubo de *transputers*. As principais modificações introduzidas no mecanismo, tais como o método de *Cancelamento Direto Local*, o novo algoritmo de *GVT* e a versão para memória distribuída do protocolo *Cancelback* só podem ser avaliadas através da comparação com experimentos realizados na mesma máquina com o sistema original. Como não se dispos de uma tal versão, não foi possível avaliar quantitativamente o efeito das modificações propostas por falta de parâmetro de comparação.

No entanto, considerando os testes sob condições extremas realizados, é possível perceber que o mecanismo na forma como foi aqui implementado, permite a obtenção de resultados compatíveis com os publicados na literatura. Extrapolando os dados apresentados sob a forma de gráficos, pode-se inferir que o potencial do mecanismo como simulador de uso genérico é bastante promissor.

Devido a grande quantidade de parâmetros existentes, tanto na implementação do mecanismo, como no modelo de avaliação de desempenho, as situações apresentadas nos testes representam um subconjunto ínfimo das possibilidades. Testes mais completos deveriam incluir variações em todos os parâmetros do modelo PHOLD, outros algoritmos para cálculo de *GVT*, experimentos com *Cancelamento Preguiçoso* complementando os resultados aqui obtidos com *Cancelamento Agresssivo* e observação do desempenho em função da quantidade de páginas de memória disponível. Estes testes requeririam um esforço de programação muito grande, ao mesmo tempo em necessitariam de um tempo muito grande de utilização do hipercubo e por isso não puderam ser realizados.

As experiências com o modelo de carga sintética permitem uma avaliação ampla das condições que o mecanismo oferece às mais diferentes aplicações. No entanto, verifica-se que é preciso um conhecimento muito grande das características de diversos tipos de sim-

ulação para que se produza um conjunto de parâmetros do modelo que descreva bem uma desta classes. Obtidos os resultados com o modelo nas condições dos testes realizados, é muito difícil prever o comportamento do mecanismo sob uma ou outra aplicação específica.

O mecanismo, sem contar com o espaço necessário para armazenar as filas de todos os processos lógicos, ocupa um espaço de memória inferior a 500 Kbytes. Considerando que na atual versão do hipercubo NCP-I, cada processador conta com um espaço de 2 Mbytes de memória, isto representa apenas 25% do total deixando todo o restante para código e dados de processos lógicos. Espera-se que neste espaço seja possível abrigar um número grande de processos lógicos permitindo uma grande exploração do paralelismo da máquina com *overheads* relativamente baixos introduzidos pelo mecanismo.

A técnica de escalonamento preemptivo de processos aqui apresentada pode ser adaptada para outros programas que tenham necessidade de fugir aos métodos de escalonamento impostos pelo *transputer*. Ficou comprovado com os testes realizados na avaliação de desempenho deste trabalho, que o mecanismo de preempção é robusto, apesar de fazer uso de técnicas não recomendadas na documentação do *hardware* ou do ambiente de desenvolvimento para programas **occam**.

Como extensões ou melhoramentos deste trabalho pode-se propor:

- um pré-processador para processos lógicos que seja capaz de determinar sem intervenção do programador o identificador de processo local ao processador e a área de variáveis de estado do sistema ocupada pelos dados do processo. Caberia também ao pré-processador fazer a replicação de código no caso de existirem processos idênticos em um mesmo processador;
- a definição de uma linguagem para programação de processos lógicos;
- um mecanismo de distribuição de processos lógicos entre os processadores do hipercubo que procure balancear a carga computacional em cada nó;
- novas técnicas de *checkpointing* que permitam uma menor freqüência de salvamento de estados e/ou uma redução da quantidade de dados registrados (possivelmente salvando somente os dados alterados entre a gravação de dois estados);
- uma adaptação no critério de seleção dos elementos a serem cancelados pelo protocolo *Cancelback* que permita a utilização de itens de memória de tamanho varia'vel;
- uma extensão do algoritmo de *GVT* para considerar redes de comunicação com canais não FIFO.

Bibliografia

[1] D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[2] D. Jefferson *et al.*, "Distributed simulation and the time warp operating system," em *Proceedings of the 11th Annual ACM Symposium on Operating Systems Principles*, November 1987.

[3] D. Jefferson and F. Wieland, "Case studies in serial and parallel simulation," em *Proceedings of the 1989 International Conference on Parallel Processing*, 1989.

[4] D. Jefferson *et al.*, "The status of the time warp operating system," in *The Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[5] D. Jefferson, "Virtual time II: The cancelback protocol for storage management in time warp," tech. rep., UCLA, October 1989.

[6] D. Jefferson and H. Sowizral, "Fast concurrent simulation with the time warp mechanism," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1985.

[7] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, October 1990.

[8] R. M. Fujimoto, "Performance of time warp under synthetic workloads," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1990.

[9] R. M. Fujimoto, "Time warp on a shared memory multiprocessor," em *Proceedings of the 1989 International Conference on Parallel Processing*, 1989.

[10] R. M. Fujimoto, "Parallel discrete event simulation," em *Proceedings of the 1989 Winter Simulation Conference*, 1989.

[11] R. M. Fujimoto *et al.*, "The roll back chip: Hardware support for distributed simulation using time warp," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1989.

[12] R. M.Fujimoto, "Lookahead in parallel discrete event simulation," em *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[13] F. Wieland and L. Hawley, "Implementing a distributed combat simulation on the time warp operating system," em *The 8th International Conference on Distributed Computing Systems*, June 1988.

[14] F. Wieland *et al.*, "Distributed combat simulation and time warp: The model and its performance," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1989.

[15] F. Wieland *et al.*, "The performance of a distributed combat simulation with the time warp operating system," *Concurrency: Practice and Experience*, vol. 1, pp. 35–50, September 1989.

[16] P. L. Reiher, "Parallel simulation using the time warp operating system," in *Proceedings of the 1990 Winter Simulation Conference*, 1990.

[17] P. L.Reiher *et al.*, "Cancellation strategies em optimistic execution systems," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1990.

[18] P. L. Reiher, D. Jefferson, and S. Bellenot, "Temporal decomposition of simulations under the time warp operating system," em *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, 1991.

[19] P. L. Reiher, S. Bellenot, and D. Jefferson, "Debugging the time warp operating system and its application programs," em *SEDMS II, Distributed & Multiprocessor Systems*, USENIX Association, 1990.

[20] M. Ebling *et al.*, "An ant foraging model implemented on the time warp operating system," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1989.

[21] B. Beckman *et al.*, "Distributed simulation and time warp - part 1: Design of colliding pucks," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1988.

[22] P. Hontalas *et al.*, "Performance of the colliding pucks simulation an the time warp operating system (part 1: Asynchronous behaviour & sectoring," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1989.

[23] M. T. Presley, P. L. Reiher, and S. Bellenot, "A time warp implementation of sharks world," em *Proceedings of the 1990 Winter Simulation Conference*, 1990.

[24] M. T. Presley *et al.*, "Benchmarking the time warp operating system with a computer network simulation," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1989.

[25] O. Berry and D. Jefferson, "Critical path analysis of distributed simulation," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1985.

[26] A. Thesen and L. E. Travis, "Introduction to simulation," em *Proceedings of the 1990 Winter Simulation Conference*, 1990.

[27] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, vol. 24, pp. 198–205, April 1981.

[28] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, February 1985.

[29] J. Misra, "Distributed discrete event simulation," *Computing Surveys*, vol. 18, March 1986.

[30] K. M. Chandy, V. Holmes, and J. Misra, "Distributed simulation of computer networks," *Computer Networks 3*, pp. 105–113, 1977.

[31] D. W. Jones, "An empirical comparison of priority-queue and event-set implementations," *Communications of the ACM*, vol. 29, April 1986.

[32] R. E. Bryant, "Simulation of packet communication architecture computer systems," Tech. Rep. TR-188, M.I.T., 1977.

[33] R. J. Lipton and D. W. Mizell, "Time warp vs. chandy-misra: A worst case comparison," em *Proceedings of the SCS Multiconference on Distributed Simulation*, January 1990.

[34] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, July 1978.

[35] V. C. Barbosa, C. L. Amorim, and E. S. T. Fernandes, *Uma Introdução à Computação Paralela e Distribuída.* VI Escola de Computação, Campinas, 1988.

[36] A. Gupta, R. M. Fujimoto, and I. F. Akyldiz, "Performance analysis of time warp with homogeneous processors and exponential task times," Tech. Rep. GIT-ICS-90-48, College of Computing, Georgia Institute of Technology, October 1990.

[37] R. E. Felderman and L. Kleinrock, "Two processor time warp analysis: Some results on a unifying approach," em *Proceedings of the SCS Multiconference on Distributed Simulation*, January 1991.

[38] Y.-B. Lin and E. Lazowska, "Determining the global virtual time in a distributed simulation," Tech. Rep. 90-01-02, University of Washington, December 1989.

[39] L. M. A. Drumond, "Projeto e implementação de um processador virtual de comunicação," Tese de Mestrado, COPPE–UFRJ, Programa de Eng. de Sistemas e Computação, 1990.

[40] B. Samadi, *Distributed Simulation, Algorithms and Performance Analysis.* PhD thesis, Computer Science Department, University of California, Los Angeles, 1985.

[41] S. Bellenot, "Global virtual time algorithms," em *Proceedings of the SCS Multiconference on Distributed Simulation*, 1990.

[42] A. Gafni, "Rollback mechanisms for optimistic distributed simulations," in *Proceedings of the SCS Multiconference on Distributed Simulation*, February 1988.

[43] A. Burns, *Programming in occam 2.* Addison-Wesley Publishing Company, 1988.

[44] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 23, pp. 666–677, August 1978.

[45] J. Marti and C. Burdorf, "Non-preemtive time warp scheduling algorithms," *Operating Systems Review*, April 1990. ACM Press.

[46] H. S. Stone, *High–Performance Computer Architecture.* Addison–Wesley Publishing Company, 1987.

[47] INMOS Limited, *A Tutorial Introduction to occam 2*, March 1988.

[48] INMOS Limited, *occam 2 Reference Manual*, 1988.

[49] INMOS Limited, *The Transputer Instruction Set – A Compiler Writer's Guide*, June 1987.